Exploring Performance-Correctness Explicitly-Decoupled Architectures

by

Alok Garg

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Michael Huang

Department of Electrical and Computer Engineering
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2011

To my family and friends

Curriculum Vitae

Alok Garg was born in 1979 in Bayana, a small town in the colorful state of Rajasthan in western India. He graduated from Indian Institute of Technology Kharagpur, India in 2002, with a Bachelor of Technology degree in the area of Electrical Engineering. Interested in circuits and logic design, he joined Paxonet Communications and worked there for two years. While working on a MIPS based microprocessor at Paxonet, Alok developed interest in computer architecture. He entered graduate studies at the University of Rochester in the Fall of 2004, pursuing research in computer architecture, under the direction of Professor Michael Huang. He received the Master of Science degree in Electrical and Computer Engineering from the University of Rochester in 2005. During his PhD, Alok has contributed to 10 original articles in international peer-reviewed journals and conferences. Since August 2010, Alok has been working at AMD as a Senior Design Engineer. Besides work, Alok enjoys adventure sports.

List of Publications and Articles Accepted for Publication:

- **A. Garg**, R. Parihar, and M. Huang, "Speculative Parallelization in Decoupled Look-ahead," in 20^{th} *International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011.

- R. Parihar, **A. Garg**, and M. Huang, "Speculative Parallelization in Decoupled Look-ahead Architectures," accepted at *ACM Student Research Competition held in conjuction with the* 20^{th} *International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011.

- B. Ciftcioglu, R. Berman, J. Zhang, Z. Darling, S. Wang, J. Hu, J. Xue, **A. Garg**, M. Jain, I. Savidis, D. Moore, M. Huang, E. Friedman, G. Wicks, and H. Wu, "A 3-D Integrated Intra-Chip Free-Space Optical Interconnect for Many-Core Chips," in *IEEE Photonics Technology Letters*, Nov 2010.

- B. Ciftcioglu, R. Berman, J. Zhang, Z. Darling, **A. Garg**, J. Hu, M. Jain, P. Liu, I. Savidis, S. Wang, J. Xue, E. Friedman, M. Huang, D. Moore, G. Wicks, and H. Wu, "Initial Results of Prototyping a 3D Integrated Intra-Chip Free-Space Optical Interconnect," in *WINDS: Workshop on the Interaction between Nanophotonic Devices and Systems held in conjuction with the* 43^{rd} *International Symposium on Microarchitecture*, Nov 2010.

- J. Xue, **A. Garg**, B. Ciftcioglu, S. Wang, J. Hu, I. Savidis, M. Jain, M. Huang, H. Wu, E. Friedman, G. Wicks, and D. Moore, "An Intra-Chip Free-Space Optical Interconnect," in 37^{th} *International Symposium on Computer Architecture*, Jun 2010.

- J. Xue, **A. Garg**, B. Ciftcioglu, S. Wang, J. Hu, I. Savidis, M. Jain, M. Huang, H. Wu, E. Friedman, G. Wicks, and D. Moore, "An Intra-Chip Free-Space Optical Interconnect," in *CMP-MSI:* 3^{rd} *Workshop on Chip Multiprocessor Memory Systems and Interconnects held in conjunction with the* 36^{th} *International Symposium on Computer Architecture*, Jun 2009.

- F. Castro, R. Noor, **A. Garg**, D. Chaver, M. Huang, L. Pinuel, M. Prieto, and F. Tirado, "Replacing Associative Load Queues: A Timing-Centric Approach," in *IEEE Transactions on Computers*, 58(4):496-511, Apr 2009.

- **A. Garg** and M. Huang, "A Performance-Correctness Explicitly-Decoupled Architecture," in 41^{st} *International Symposium on Microarchitecture*, Nov 2008. (**Nominated for Best Paper Award**)

- L. Zhang, A. Carpenter, B. Ciftcioglu, **A. Garg**, M. Huang, and H. Wu, "A Low-Power Clock Distribution Scheme for High-Performance Microprocessors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(9):1251-1256, Sep 2008.

- H. Wu, L. Zhang, A. Carpenter, **A. Garg**, and M. Huang, "Injection-Locked Clocking: A Low-Power Clock Distribution Scheme for High-End Microprocessors," in 3^{rd}*Watson Conference on Interaction between Architecture, Circuits, and Compilers*, Oct 2006.

- **A. Garg**, F. Castro, M. Huang, D. Chaver, L. Pinuel, and M. Prieto, "Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors," in *International Symposium on Low-Power Electronics and Design*, Oct 2006.

- **A. Garg**, M. W. Rashid, and M. Huang, "Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification," in 33^{rd}*International Symposium on Computer Architecture*, Jun 2006.

- R. Huang, **A. Garg**, and M. Huang, "Software-Hardware Cooperative Memory Disambiguation," in *International Symposium on High-Performance Computer Architecture*, Feb 2006.

Acknowledgments

It is a pleasure to thank many the people who made this thesis possible.

It is difficult to overstate my gratitude to my Ph.D. supervisor, Prof. Michael Huang. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without him.

I would like to thank my thesis committee members, Prof. Eby G. Friedman, Prof. Sandhya Dwarkadas, Prof. Chen Ding, and my graduate teachers (especially Prof. Martin Margala and Prof. Wendi Heinzelman). I am indebted to my many student colleagues for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Rajeev Garg, Wasiur Rashid, Arrvindh Shriraman, Xin Li, Aaron Carpenter, Lin Zhang, Jing Xue, Berkehan Ciftcioglu, and Raj Parihar.

I wish to thank my best friend in high school (Anuj Kushwaha), my best friend as an under graduate (Hetul Sanghvi), and my best friend as a graduate student (Ram Kumar Sabesan), for helping me get through the difficult times, and for all the emotional support, comraderie, entertainment, and caring they provided. My thanks to my sister, Deepti, for her continuous support and interest in what I do.

Lastly, and most importantly, I wish to thank my parents, Hari Charan Garg and Veena Garg. They bore me, raised me, supported me, taught me, and loved me. To them, I dedicate this thesis.

Abstract

Optimizing the common case has been an adage in decades of processor design practices. However, as the system complexity and optimization techniques' sophistication have increased substantially, maintaining correctness under all situations, however unlikely, is contributing to the necessity of extra conservatism in all layers of the system design. The mounting process, voltage, and temperature variation concerns further add to the conservatism in setting operating parameters. Excessive conservatism in turn hurts performance and efficiency in the common case. However, much of the system's complexity comes from advanced performance features and may not compromise the whole system's functionality and correctness even if some components are imperfect and introduce occasional errors. In this thesis, we propose to separate performance goals from the correctness goal using an *explicitly-decoupled* architecture.

As a proof-of-concept, we discuss two such incarnations for an out-of-order microprocessor. First, we discuss how explicitly-decoupled architecture can be used to implement an efficient mechanism to track and enforce memory dependences. Later, we discuss enhancements to improve traditional ILP (instruction-level parallelism). In both the designs a decoupled performance enhancement engine performs optimistic execution and helps an independent correctness engine by passing high-quality predictions. The lack of concern for correctness in the performance domain allows us to optimize its execution in a more effective fashion than possible in optimizing a monolithic design with correctness requirements. In this thesis we show that such a decoupled design allows significant optimization benefits and is much less sensitive to conservatism applied in the correctness domain.

Table of Contents

List of Figures

List of Tables

Foreword

While I am the author of this dissertation, the work in this dissertation would not have been possible without the collaboration of various students and professors. Specially, I would like to thank my advisor, Prof. Michael Huang, for providing valuable suggestions and technical guidance throughout the thesis.

In the early stages of this thesis, when the idea of *Explicitly Decoupled Architecture* was still in it's infancy, the work on *Decoupled Memory Dependence Enforcement*, presented in Chapter 3, was seeded and developed after numerous discussions with Michael Huang, Wasiur Rashid, and Ruke Huang. I would like to thank Wasiur Rashid and Ruke Huang for their valuable support in the initial modeling and evaluation of ideas. Our collaboration resulted in a paper at ISCA06.

The initial work on Decoupled Memory Dependence Enforcement lay the foundation of this thesis and future work. Principles behind *Explicitly Decoupled Architecture* were later framed with significant contributions from Michael Huang and valuable inputs from Professor Jose Renau from University of California, Santa Cruz. This framework along with a proof-of-concept to improve traditional ILP lookahead (presented in Chapter 1 and Chapter 4) was published at MICRO'08. This work was rewarded with a nomination for the best paper award at MICRO'08.

The speculative parallelization for decoupled lookahead, described in Chapter 5, was designed and developed by me with advice from Michael Huang. I would like to thank Raj Parihar for valuable debugging support in the later stages of this work.

Chapter 1

Introduction and Motivation

1.1 The Problem: targeting performance and correctness goals at the same time

Achieving high performance is a primary goal of processor microarchitecture design. While designs often target the common case for optimization, they have to be correct under all cases. Consequently, while there are ample opportunities for performance optimization and novel techniques are constantly being invented, their practical application in real product designs faces ever higher barriers and costs, and diminishing effectiveness. Correctness concern, especially in thorny corner cases, can significantly increase design complexity and dominate verification efforts. The reality of microprocessor complexity, its design effort, and costs [BMMR05] reduces the appeal of otherwise sound ideas, limits our choice, and forces suboptimal compromises. Furthermore, due to the tightly-coupled nature of monolithic conventional microarchitecture, conservatism or safety margin necessary for each component and each layer of the design stack quickly accumulates and erodes common-case efficacy. With mounting PVT (process, voltage, and temperature) variation concerns [BKN+03], the degree and extent of conservatism will only increase. The combination of high cost and low return makes it increasingly difficult to justify implementing a new idea and we need to look for alternative methodology that allows us to truly focus on the common case. One promising option is to explicitly decouple the circuitry for performance and correctness goals, allowing the realization of each aspect to be more ef-

ficient and more effective. Decoupling is a classic, time-tested technique and seminal works on various types of decoupling in architecture [Aus99; MHW03; ZS02; Smi84; SPR00] have attested its effectiveness and advanced the knowledge base. Building on this foundation, we propose to explore *explicitly-decoupled* architecture.

By "explicit", we mean two things. First, the decoupling is not simply providing a catch-all mechanism for a monolithic high-performance microarchitecture to address rare-case correctness issues. Rather, from ground up, the design is explicitly separated into a performance and a correctness domain. By design, the performance domain only *enables* and *facilitates* high performance in a probabilistic fashion. Information communicated to the correctness domain is treated as fundamentally speculative. Therefore, correctness issues in the performance domain will only affect its performance-boosting capability and become a performance issue for the whole system. This allows true focus on the common case and reduction of design complexity, which in turn permits the implementation of ideas previously deemed impractical or even incorrect. An effective performance domain allows designers to use simpler, throughput-oriented designs for the correctness domain and focus on other practical considerations such as system integrity.

Second, the architecture design is not just conceptually but also physically partitioned into performance and correctness domains. The physical separation extends to the whole system stack from software and microarchitecture down to circuit and device. Physical separation ① allows the entire system stack to be *optimistically* designed; ② conveniently and economically provides the same mechanism for ultimate correctness guarantee; and ③ permits custom software-hardware interface in the performance domain, which opens up more cross-layer cooperative opportunities to implement ideas difficult to accomplish within a single layer.

Explicitly decoupled architecture represents a very broad design space. What aspect of performance improvement is best achieved in such a decoupled fashion, and how to exploit the lack of concern for correctness to design novel optimization techniques and indeed synergistic techniques from different layers are but a small set of questions that need to be addressed. To narrow down the exploration in this thesis, we explored a few techniques, *e.g.*, we used the explicitly decoupled architecture principle to improve traditional ILP (instruction-level parallelism) lookahead, and study the effect of using practical, complexity-effective techniques to

manage long, and more importantly, unpredictable latencies associated with branch and load processing.

1.2 Thesis Statement

Explicitly decoupled architecture offers an effective and efficient design paradigm where performance and correctness goals can be pursued independently while keeping overall design complexity under control.

1.3 Contributions

This thesis makes the following major contributions:

- A Framework for Explicitly Decoupled Architecture is presented. In particular, probabilistic lookahead techniques for performance improvement can be more efficiently realized when correctness constraints on these techniques is small, hence, we press for greater autonomy and high tolerance against deviations for the lookahead agent. In this thesis, we also quantify the effect of deviances in performance domain on the system.

- With correctness, semantic, and compatibility constraints removed from the performance domain, the whole system stack from software and microarchitecture down to circuit and device can take advantage of the flexibility to speedup the common case. We presented and proposed a few such software and microarchitectural optimizations in this thesis.

- With the focus on achieving high performance moved to the performance domain, correctness domain can have a simple and throughput oriented design. Additionally, it is easier to make the implementation more robust against various emerging concerns such as PVT variation. We evaluated a few simplification opportunities in the correctness domain.

- Since energy impact of any new architecture is of paramount importance, we also evaluate the energy efficiency of such a decoupled architecture.

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 discusses background and related work; Chapter 3 discusses how concepts of explicitly decoupled architecture can be used to implement an efficient mechanism to track and enforce memory dependences; Chapter 4 takes the design discussed in Chapter 3 a step further by using explicitly decoupled architecture to improve traditional ILP (instruction-level parallelism) lookahead; Chapter 5 explore speculative parallelization in a decoupled look-ahead agent ; and Chapter 6 summarizes and discusses future work.

Chapter 2

Background and Related Work

In this thesis, we attempt to explicitly separate the complexity of performance optimizations from the concerns for correctness. By explicit decoupling, we hope to target the common case more effectively in the performance domain. This can allow ideas that often remain in the realm of literature due to cost of implementing them on a monolithic microprocessor. In fact, we explored a few such insights in the context of performance domain. To give an overview of these ideas, we discuss the state of the art in microprocessor and summarize other optimizations presented in the literature. Most of these performance improvement techniques either try to improve lookahead to mitigate future performance bottlenecks early enough; or find ways for parallel execution of a program.

Due to long memory latencies of main memory, a significant part of the processor execution time is consumed in stalls waiting for data from the off-chip memory. A state-of-the-art microprocessor tries to hide these long latencies by prefetching data employing lookahead techniques. These lookahead techniques can be roughly classified into three main categories: ① state-machine controlled prefetching, ② program execution driven prefetch agent, and ③ out-of-order processing. We will discuss these techniques in the next few sections.

Executing a program in parallel is also one of the ways to improve performance. The performance improvement is partially determined by the amount of parallelism that can be exploited from a program. However, many programs are inherently sequential, or even if a degree of parallelism exists, it is hard to exploit using state-of-the-art compilation techniques. Speculative

parallelization is proposed for such programs. Although, speculative parallelization is not directly related to this thesis, in Chapter 5 we use certain insights and techniques from speculative parallelization in the performance domain to improve lookahead. We will discuss a few aspects of this work related to thesis in Section 2.4.

Finally, decoupling is a classic, time-tested technique and seminal works on various types of decoupling in architecture have been proposed. Building on this foundation, in this thesis we propose to explore explicit decoupling of performance and correctness aspects of a microprocessor. To provide background, we will summarize other decoupled architectures in Section 2.5.

2.1 State-Machine Controlled Prefetching

State-machine controlled prefetching often uses memory accesses that have been observed to predict and prefetch potential future accesses. Studies show that many memory references are predictable and can be successfully exploited using software or hardware mechanisms. Some of these advanced predictors are already being used in microprocessors.

2.1.1 Software prefetcher

Special code for data prefetching can be added to the program. An expert programmer can understand the memory behavior of a program and write additional code to prefetch future memory references. The compiler can also insert special prefetch instructions by analyzing the application's memory access behavior [Por89; CKP91; CMCWm91; KL91; MLG92; LM96; RS99]. Array accesses inside a loop or pointer-chasing accesses are a few examples where memory reference analysis is relatively easy.

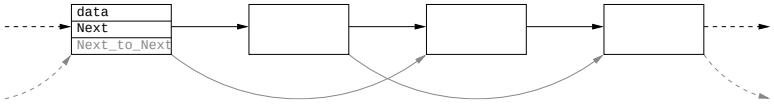
An array of data is often accessed linearly inside a loop using an induction variable. As shown in Figure 2.1-(a), compilers can recognize such loops and understand the induction variable computation relatively easily. The address computation of a memory reference can be advanced to launch prefetches. Scheduling algorithms are used to determine the minimum loop iterations needed to hide most of the memory access latency [Por89; CKP91; KL91; MLG92]. For example, in Figure 2.1-(a), prefetches are scheduled 50 loop iterations ahead.

```
for(i=0; i < 20000; i++) {
    a[i] = a[i] + b[3*i];
    prefetch (a[50+i]);
    prefetch (b[3*(50+i)]);
}
```

(a)

```
class LinkedList {
    int data;
    LinkedList Next;
    LinkedList Next_to_Next;

    int Sum () {
        prefetch (Next_to_Next);
        if(next != NULL)
            return data + Next.sum();
        return data;
    }
}
```

(b)



(c)

Figure 2.1: Examples of software prefetching.

Another class of memory accesses involve traversal through a linked chain to reach the target location in pointer-based data structures such as linked-lists (Figure 2.1-(b,c)), trees, and other graph structures. Compiler algorithms try to prefetch these links ahead of time [LSKR95; LM96; Sel92; RMS98]. In this type of code, data is accessible through pointers that are saved in any of the neighboring data structures. Pointers create an arbitrary level of indirection, sequentializing the access to the target data structure. This makes prefetching pointer-chasing code challenging. To address this problem, compilers insert additional pointers to the data structure to connect non-consecutive elements, cutting the amount of indirections needed to launch prefetches deep down the chain [RS99]. This technique is illustrated in Figure 2.1-(b,c).

There are many challenges in software prefetching. The effectiveness of software prefetching relies on the ability of compiler to advance the address computation of memory references sufficiently ahead of time, which is usually difficult. Further, it is hard to predict the cache behavior of an application using only static analysis techniques. Application profiling can reveal some of the potential prefetch targets [ML00].

2.1.2 Hardware prefetching

Hardware prefetcher, usually located along with a cache controller, uses run-time information to accumulate the dynamic hit and miss profile of the executing application. It detect patterns in the miss profile, and exploits their temporal locality to launch prefetches. For example, in a typical implementation, lines at constant offsets from an address A are prefetched as soon as a reference to address A is detected. In general, these prefetchers exploit regularity of memory accesses to detect streams with constant strides or repetitions of memory references.

Stride based prefetching

These prefetchers detect and lock on to a stream of memory references that show a constant stride behavior. Once a stream with the constant stride is detected, prefetches are generated using the base address and offset recorded for the stream. To minimize potential stalls, these prefetches are launched several steps ahead of current access. Prefetches are considered successful if a demand request hits the prefetched cache lines. Successful prefetch updates the base address and trigger new prefetches. Many such streams can exist in the dynamic miss profile of an application, and hardware support is needed to disambiguate them. Recent proposals differ in ways to efficiently detect and track multiple streams [BC91; CB95; JG99; HMK91; KS02; KV97; LFF01; PK94; SJT02; NS04]. For example, Nesbit *et al.* proposed to maintain an ordered history of addresses to detect streams with constant stride [NS04].

Non-stride based prefetching

These prefetchers are based on the observation that many applications utilize data structures with repetitive layouts and access patterns. Memory references in these applications are often non-contiguous and irregular. However, they recur in the same order and with the same relative offset from a previous access [JMH97; KW98; CYFM04; SWA$^+$06]. Traversing a binary-tree is one such example that does not follow a constant stride. However, the local access pattern may repeat or follow a similar access pattern. To exploit this behavior, the hardware mechanism proposed by Somogyi *et al.* divides memory into regions [SWA$^+$06]. First access to a region

triggers pattern detection in this region. Patterns are stored as an offset to the address of trigger access using a bit vector. These patterns are tagged to the PC and address of the trigger access, and are used to launch prefetches whenever this tag is observed.

2.1.3 Summary

Although practical implementations of a state-machine controlled prefetching algorithm can eliminate misses by targeting common types of memory reference patterns, it cannot hide certain types of memory stalls due to the inability of these techniques to prefetch irregular patterns without adding too much hardware complexity or overhead on the application. Even if prefetching is used aggressively, they may be issued either too late or too early. Further, these prefetches might not always be accurate, and run into the risk of polluting cache and wasting memory bandwidth.

2.2 Program Execution Driven Lookahead Agent

To extract better and more accurate dynamic information of future program behavior, a class of techniques execute a future code section of the program. Often, part of the program, or a version derived from it, is executed ahead of time before the actual execution. The information gained from this execution is used to improve the cache hit rate of the original execution. For discussion purposes we refer to this pre-executed code as the *lookahead agent*. The core assumption behind this technique is that the lookahead agent would be able to capture memory access patterns, which state-machine controlled prefetching cannot target accurately or in a cost effective way. The way the lookahead agent(s) are constructed and triggered differs among various proposals.

2.2.1 Future Execution on a Conventional Stall

Recent proposals use idle resources during conventional stalls to trigger execution of program code itself to perform prefetching [DM97; BDA01a; MSWP03; CST+04; KKCM05]. In a typical implementation, the processor, instead of idling, saves the architectural state and continues executing instructions speculatively. Hence, the processor may be able to overlap future

long-latency memory accesses in the shadow of a stall. When the stall is over, the speculation is verified and/or recovery is performed from the checkpoint. The subsequent instructions are dispatched all over again, making speedier progress as the loads might have been prefetched. However, this also means that the scheme is reactive. Engaged only during the stalls, the scope is limited, since it cannot effectively hide the latency of the stall that triggered this execution mode.

2.2.2 Helper threading

In another class of work, the main program forks out a snippet of code to issue prefetches or resolve branches ahead of their actual execution in the main thread [FTEJ98; DS98; CSK+99; ZS00; APD01; Luk01; ZS01; CWT+01; CTWS01; RS01; MPB01; CTYP02a; CTYP02b; CSCT02; KY02; ZTC07]. These snippets of code are either hand optimized or automatically generated by the compiler from the main program. The top several targets of performance degrading events are first selected either from profiling run or from static analysis of the program. Performance degrading events are branches that are hard to predict correctly by branch predictors or loads that frequently miss in the cache. Starting from the target, instructions on its reverse data flow graph (DFG) are selected as a part of its slice. The DFG traversal is bounded by the maximum size of the slice and trigger selection point. Trigger point is selected such that the target events are resolved (hopefully) before being used by the main thread, and at the same time, close enough to receive accurate live-in values from the main thread needed by the slice for correct progress and to remain useful. Helper threads are terminated once execution reaches the end of the slice. Special mechanisms are required to detect and terminate helper threads that become run-away due to speculative execution.

The biggest advantage of the helper threads is that they can be used to target irregular long-latency memory references. However, there are several challenges in implementing cost-effective helper-threading. First, systematic and automatic generation of high-quality helper thread code is difficult. The code has to balance efficiency with the success rate. Recall that these code slices have to compute irregular addresses and are invoked much higher up in the control flow when some necessary input may not be available. Duplicating and adding the code

to generate this input data may be inefficient. Second, triggering helper threads at the right moment is also challenging. Both early and late triggering reduce a helper thread's effectiveness. Finally, and perhaps most importantly, helper-threading needs the support to quickly and frequently communicate initial values at the register level from the main thread to the helper threads, which dictates that the hardware support for helper threads have to be tightly coupled to the core.

In a somewhat different class of helper threading design, the helper thread does not attempt to get ahead of the main thread in the control flow, but instead infers the right access pattern by operating "downstream" of the main thread [GB05a; GB06]. This access pattern can be applied to the future code sections with different base address.

2.2.3 Two-pass architectures

In a two-pass architecture, the program is processed in two passes using an extra thread of execution. One of the two threads uses a speculatively generated reduced version of the program's execution stream to remain ahead of the main thread. It launches prefetches in advance and passes other predictions to speed up the main thread. Recent proposals, like slip-stream [PSR00; SPR00], flea-flicker [BNS+03b; BNS+03a], dual-core execution [Zho05], tandem [MMR07], and paceline [GT07], differ in how the lead/lookahead thread is accelerated.

In slip-stream, the lookahead thread is constructed by eliminating ineffectual instructions such as highly predictable branches and silent stores. Non-essential instructions usually constitute only a small percentage of all the instructions, and tend not to be on the critical path. Consequently, eliminating them does not speed up the program significantly. Dual-core execution, on the other hand, uses a full version of the program. To achieve lookahead, it performs speculation during stalls due to long-latency memory accesses. When a long-latency memory access blocks the pipeline, the instruction responsible for the stall and all its dependent instructions are flushed from the pipeline. This allows the lookahead thread to run ahead and issue prefetches. Tandem and paceline achieves acceleration of the lookahead thread by improving the operating clock frequency of the lookahead agent. Increase in frequency by as much as 25% is possible in Tandem by systematic pruning of infrequently used functionality, only im-

plemented to satisfy the rare corner cases. Paceline exploits extra timing margins built in the design to overclock the lookahead agent by 30%. Note that in all these proposals the lookahead agent is still conservatively designed and tries to produce almost 100% correct program output, limiting lookahead capabilities. This level of accuracy, however, is required because of heavy dependence of main thread on the information it receives from the advanced thread.

2.2.4 Summary

In Figure 2.2, various program execution driven lookahead mechanisms are compared to an out-of-order processor execution. Bars from left to right show the progress (time line). Thick bars show the progress of the processor core and thinner bars shows the progress of the L2 cache miss event. A modern out-of-order processor could not exploit far-flung instruction level parallelism to hide processor stalls due to long latency memory references. The runahead execution, proposed by Mutlu *et al.*, starts a special lookahead mode (referred to as runahead mode) in the shadow of stall due to "load A" [MSWP03]. This runahead mode launches prefetch for "load B" to hide its memory access latency. However, the processor still suffers the full impact of the first L2 miss. In helper threading or two-pass architectures, the lookahead agent launches prefetches to partially hide the memory access latency of both "load A" and "load B" to achieve a significant improvement in execution time. These lookahead mechanisms are also compared with a large instruction window processor (discussed in the next section). A large instruction window processor is able to exploit far-flung instruction level parallelism and execute "load B" before "load A" cache miss stalls the pipeline, allowing partial overlap of its L2 miss latency.

2.3 Out-of-Order Processing for Lookahead

Modern processors exploit instruction level parallelism to overlap and hide latencies. To dynamically search for parallelism, a processor buffers a stream of instructions, and allows independent instructions to be executed out of their semantic program order. As discussed in Figure 2.2, hiding longer latencies requires larger instruction buffering capabilities to exploit far-flung instruction level parallelism. In a processor, therefore, lookahead is tightly integrated with the out-of-order execution engine [Tom67] and is achieved via a set of heavy-weight mech-

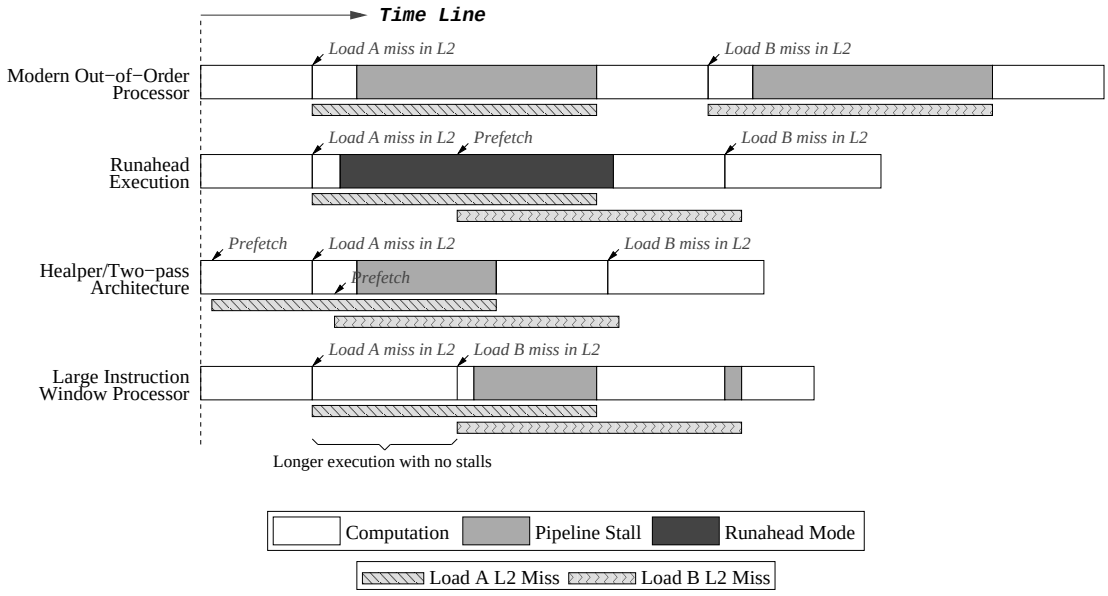


Figure 2.2: Execution time-line for various lookahead architectures under two L2 misses for Load A and Load B. (Part of this figure is taken from [MSWP03]). The thick bars show the progress of the processor core and the thinner bars show the progress of the L2 cache miss event.

anisms: ① sophisticated branch prediction and complex rollback support, ② register renaming, ③ out-of-order issue logic to maintain register-based dependences, and ④ load-store queues to buffer speculative memory updates and to enforce memory-based dependences. Figure 2.3 shows the block diagram of an out-of-order processor. The size and capabilities of these structures determine the amount of lookahead that can be extracted from the program. However, many of these structures are very expensive to scale. As a result, lookahead beyond several tens of instructions quickly becomes impractical. Proposed solutions include better utilization and effective ways to scale up these structures.

Optimized utilization of processor resources

Careful and efficient utilization of hard-to-scale structures is one of the solutions to improve buffering capabilities of the processor. Recent proposals allow early release of resources by speculatively detecting their last use [MRH+02; ARS03; TNN+04]. Resource allocation can also be delayed until their actual use. For example, allocation of a physical register can be delayed until the value is actually ready to be written [WB96; GGV98; TNN+04].

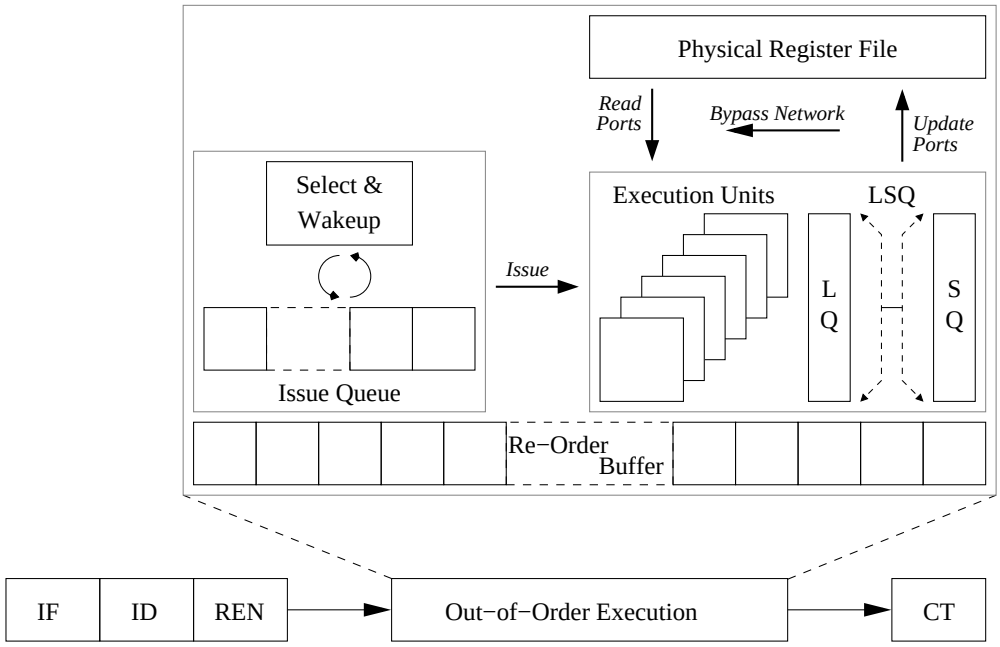


Figure 2.3: Simplified block diagram of an out-of-order processor (IF - Instruction Fetch, ID - Instruction Decode, REN - Renaming, CT - Instruction Commit, LSQ - Load and Store Queue).

A body of work frees critical resources during processor stalls by draining the instruction responsible for stall (and its dependency chain) from the processor pipeline [LKL+02; SRA+04; PCG+06]. This instruction slice is saved separately along with source register and memory values for later execution. Proposal by Cristal *et al.* allow completed instructions to be removed speculatively from the pipeline while older instructions are still pending completion, hence freeing up the resources early [COLV04]. Hardware mechanisms are required to enforce correctness and state recovery during mis-speculation.

In a separate line of work, to make the small instruction window that exist in the current processors amenable for lookahead, Pai *et al.* uses compiler analysis to cluster long-latency accesses [PA99]. The hope is that the out-of-order execution engine does not have to exploit far-flung ILP to perform these long-latency accesses in parallel.

Building large instruction window

Building large structures is a key to buffer a large number of instructions. However, it is challenging because of negative effect on their access time. These structures are often on the critical

path, affecting processor's cycle time or pipeline depth or both. Bigger structures also incur energy penalties, which undermine the overall design goal. Many proposals employ a two-level approach to scale up buffer sizes: to make the first-level (L1) structure small (thus fast and energy efficient) and still able to perform a large majority of the work. This L1 structure is backed up by a much larger second-level (L2) structure to correct/complement the work of the L1 structure.

We will next discuss individual components and proposed enhancements.

2.3.1 Branch prediction

To buffer instructions, the fetch unit has to supply a continuous stream of instructions past multiple control flow divergence points. However, it takes many cycles after fetch to execute the branch and resolve the correct direction. To avoid stalling until the branch is resolved, the fetch unit predicts the direction and speculatively supplies instructions along the predicted path. This speculation is later verified when the branch is executed. Mis-speculation results in a costly pipeline flush and state recovery.

To achieve a high accuracy of branch prediction, sophisticated multi-level branch predictors are used [YP92b; YP92a; McF93]. These predictors track the past behavior of a branch (using bimodal predictor) as well as the behavior of the different branches (using global predictor). The bimodal predictor successfully predicts those branches that show a strong bias towards either taken or not taken, *e.g.*, loop branches. Direction of other branches depends on the shared history of many branches. A global branch predictor uses correlation between different branches in the prediction making. A separate predictor is used to choose the best among two for each branch [McF93].

Conventional branch predictors, although very accurate, are still imperfect for deep lookahead. Researchers try to address this issue by exploring instruction execution on both paths to hedge the risk of losing lookahead on to the alternate path.

Predicated execution

Execution of one of the two diverging paths usually depends on the resolution of the control dependency. As shown in Figure 2.4, compiler can convert a control dependency into a data dependency by allowing instructions on both paths to either perform an operation or do nothing. This technique is usually referred to as predication. Since executing instructions on both paths puts extra pressure on the processor resources, it is usually performed on short branches. A limited form of predication is supported by current generation of compilers and processors. Kim *et al.* proposed to use a form of dynamic predication only for hard to predict branches [KMSP05; KJM+06].

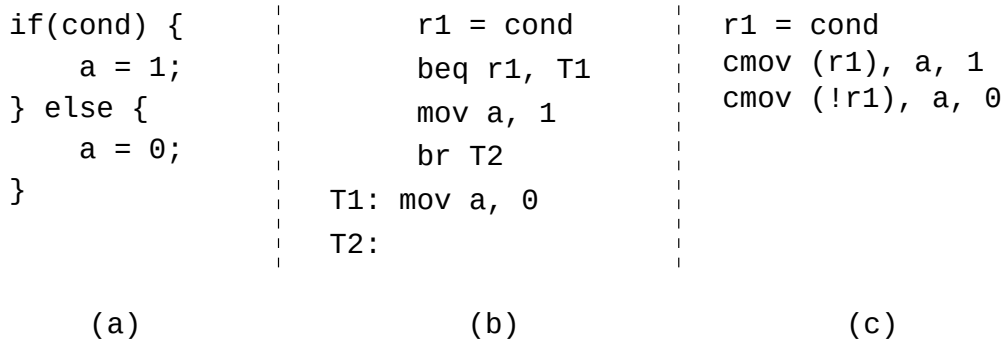


Figure 2.4: Example of predication - (a) source code, (b) normal assembly, and (c) predicated assembly.

Multi-path execution

An alternative to single path speculative execution is to conservatively execute on both paths, and flush the wrong path once the branch is resolved. In this approach, referred as *Eager Execution*, the number of unresolved paths a processor has to process increases exponentially with the increase in its instruction window size [WCT98]. To reduce the cost of processing a large number of paths, proposals like disjoint eager execution [USH95] and adaptive branch trees [Che98] executes only a subset of paths. Execution probability of eligible paths is often used in the selection criteria. *Selective Eager Execution* (SEE), on the other hand, uses eager execution only for hard to predict branches [KPG98]. Like predication, these approaches need high execution bandwidth to simultaneously handle multiple paths.

2.3.2 Register renaming

Compilers use a fixed number of architectural registers to communicate values between instructions. These registers are recycled and reused often. The false dependencies created by the reuse of registers artificially limit the amount of ILP that can be exploited in a program. An out-of-order processor uses register renaming to eliminate output and anti dependences among in-flight instructions by providing multiple physical registers (from a physical register file) mapped to multiple instances of a logical register [Tom67; MPV93].

There are many challenges in scaling up the size of physical register file. A wide superscalar processor, executing many instructions every cycle, needs a heavily ported register file to support more than one reference per instruction. At the same time, a larger, heavily ported register file has to support minimal read and update latency to keep the complexity of the bypass logic at a manageable level. Bypass logic is used to feed results directly from just executed instructions until their availability from the register file. To achieve lower access time, Alpha 21264 used clustered organization to reduce the number of ports per cluster [Kes99].

A class of work exploits value locality to reduce resource pressure on physical register file [JRB+98; BS03; TNN+04]. The value produced by an instruction is often the same as value produced by another recently executed instruction, resulting in multiple physical registers containing the same value. Typically, of the values that are often repeated, 0 and 1 occur with highest frequency. Allocating shared physical registers for some of the frequent values can reduce the pressure on register file.

To support a larger register file, a body of work proposed variations of two-level register file [SP88; YR95; CGVT00; BDA01b; TA03; BS04]. Typically, a small level-1 register file provides a fast path to registers that are going to be accessed soon. In the proposal by Balasubramonian *et al.*, registers are allocated from the level-1 at dispatch [BDA01b]. After the last use of the level-1 physical register, it is de-allocated and the value is moved to a level-2 physical register. Values from the level-2 are used infrequently, and hence, can withstand a longer access latency. Since values stay in the level-1 register file only for a very short duration compared to their entire life span, a small number of registers are sufficient to support a larger instruction window.

2.3.3 Issue logic

To exploit instruction level parallelism, the issue logic aggressively finds ready instructions to fill available execution slots. This logic is divided into two mutually-dependent steps: select and wakeup. Select logic checks all the ready instructions that can be executed in the next cycle. At the same time, the dependents of selected instructions have to be woken-up for selection. To avoid bubbles in the execution engine, these two functions have to be performed in the same cycle, forming a tight scheduling loop. The implementation of issue logic not only has to support coupled single cycle select and wakeup, but also aggressively utilizes broadcasting, compaction of queues, and heavily-ported structures - making the design hard to scale [Tom67; PJS97].

A decoupled implementation of select-wakeup loop has been proposed to remove it from the critical path [GNK+01; BSP01; MMHR06]. The insight behind the proposals is the observation that usually no more than one instruction becomes ready per cycle. Brown *et al.* takes advantage of this common case by speculatively selecting all waking instructions for execution [BSP01]. The select logic, which is not any more on the critical path, is only used to validate the speculative scheduling. In a separate work by Mesa-Martinex *et al.*, the wakeup logic allows instructions to wake up their dependents as long as they are woken up [MMHR06]. Selection is performed only on the woken-up instructions, decoupling it from the wakeup logic and removing it from the timing critical path. This is a significant difference from the conventional design, where only selected instructions can wake up their dependents.

Brekelbaum *et al.*, on the other hand, proposed to use a hierarchical issue window design [BIWB02]. A small and fast scheduling window is used to issue latency-critical instructions. While a large but slow scheduling window is used to store instructions that cannot execute soon – the chain of dependent instructions waiting for long-latency memory reference to resolve. These instructions are identified by using a built in heuristic and are steered towards the large scheduling window.

2.3.4 Load store queue

While it is straightforward to establish dependencies between instructions using register names, the same is not possible for memory instructions. This is because the address, required for memory reference, is not immediately available at the time of dispatch. The processor employs load-store queue (LSQ) to keep track of memory instructions to make sure that the out-of-order execution of these instructions do not violate the program semantics.

A number of factors hinder the scalability of load-store queue. First, the queues are are typically implemented as priority CAMs. Not only is an address used to search through the queue in a fully associative manner, the match results also have to go through a priority logic to pin-point the oldest or the youngest instance for precise forwarding of data. Second, searches in the store queue are on the timing critical path of load execution, limiting the allowed store queue search latency. This makes store queue even harder to scale. In Chapter 3, we will discuss how the principles of explicitly decoupled architecture can be applied to perform memory disambiguation more effectively using scalable structures. Next, we will discuss some of the related work, and postpone the detailed comparison with our work until Section 3.1.3.

A large body of work adopts a two-level approach to disambiguation and forwarding [ARS03; BZ04; Rot04; GAR+05; TIVL05]. In two level store queue organization proposed by Akkary *et al.*, a fast and small level-1 queue is backed up by a much larger and slower level-2 store queue [ARS03]. Stores are inserted into the level-1 at the time of dispatch and are moved to level-2 to make way for new stores. In the common case, loads do not receive forwarding from the distant stores, residing in level-2. To improve the critical path, a fast membership test is performed on the level-2 to rule out forwarding. Exploiting similar insights, Torres *et al.* proposed to support forwarding (speculative) from a subset of stores using a level-1 store queue [TIVL05]. To verify the speculation, level-2 buffers all the stores, but does not support forwarding. Since level-2 is not on the timing critical path anymore, it is relatively easy to scale.

Another body of work proposed filtering mechanisms to reduce check frequency [SDB+03; POV03; GCH+06; CNG+09]. Sethumadhavan *et al.* proposed to physically partition load-store queue into banks [SDB+03]. A conservative membership using bloom filter can quickly filter out banks that will not find a match. In our previous work, we proposed to use hash tables to

eliminate the load queue [GCH+06; CNG+09]. Load queue is primarily used to detect stores-load ordering violations. Typically, stores search through all the loads to detect if any younger load has executed prematurely. We proposed to use an address-indexed hash table to track the youngest executed load for a group of addresses. A store hitting in the table conservatively hints a potential violation.

To eliminate the requirement of a load queue, Cain *et al.* proposed to re-execute the loads in the program order to validate the value returned from prior execution [CL04]. Re-execution of loads, however, can slow down the retirement stage and may put extra pressure on the memory sub-system. Various filtering mechanisms are proposed to reduce re-execution frequency [Rot05].

To better utilize the store queue resource, silent stores need not consume any entry [LL00]. The processor can speculatively detect the silent stores and prevent them from occupying a store queue entry. Similarly, load instructions that are guaranteed not to overlap with earlier pending stores in store queue also do not have to reside in load queue. Huang *et al.* uses software assistance to identify these loads and prevent them from competing for the resources in the load queue [HGH06].

2.3.5 Summary

Achieving deep lookahead by increasing the instruction window size of the processor is promising, but a practical implementation requires increasing the buffering capabilities of several structures. Efficient utilization of some of these structures without any improvement in their sizes is not enough to provide significant lookahead capabilities. Despite proposals to improve the scalability of some of these structures, it is difficult to have a practical implementation without unreasonably increasing the cycle time, power/energy, design complexity, and verification effort of the current processors.

2.4 Speculative Parallelization

Up to now we have discussed techniques to achieve lookahead by exploiting instruction level parallelism. Another way to improve performance is by executing a program in parallel on multiple processing cores in the system. The achievable performance improvement depends on the amount of parallelism that can be exploited from a program. However, many programs are inherently sequential or even if a degree of parallelism exists, it is hard to exploit using state-of-the-art compilation techniques. Speculative parallelization is proposed for such programs. Insights and techniques from the past work, though not directly related to this thesis, are used to improve lookahead in the performance domain.

A typical implementation speculates that a "task" is independent, spawn another thread, and executes it in parallel with the rest of the program. To detect miss-speculations, the primary and the spawned thread monitor accesses to register file and memory. In case of a dependence violation, *e.g.*, when the primary thread (executing older instructions) writes to a location already read by the spawned thread (executing younger instructions), the speculative thread is aborted and the architectural state is recovered. Finally, success of speculative parallelization depends on capability of software or hardware mechanisms to find out tasks with little inter-task dependencies. Recent proposals differ in how the violations are detected and how updates from the parallel tasks are merged [SBV95; AD98; HWO98; CMT00; ZS02; BS06; AMW+07]. Using such a technique only to improve lookahead in the performance domain has fundamental advantages. By speculatively parallelizing the lookahead thread, some dependence will also be violated, but the consequence is (much) less severe: (e.g., there is no impact on correctness). This makes tracking of dependence violations potentially unnecessary. We will discuss these advantages and new challenges in Chapter 5.

2.5 Decoupled Architectures

Decoupling in micro-architecture is a technique where two mutually dependent design goals are separated to achieve a variety of architectural improvements, *e.g.*, performance, reduction in design/verification complexity, or reliability. In one of the earliest work, the concept of decoupling

was exploited by Smith *et al.* to separate memory accesses (referred to as access processor) from the execution (referred to as execute processor) to achieve better lookahead [Smi84]. Both processors execute a separate stream of programs performing two different functions. Front-running access stream is only responsible for lookahead and fetch memory operands that can be used by the execute stream. By using an independent control flow separated from the main program thread, the access stream is different from other architectures like helper-threading and two-pass architecture. The access stream is independent and continuous, not spawned as a reaction to cache misses. The proposed architecture was further extended by Tyagi *et al.* to decouple control flow from memory access and computation [TNM99]. The control flow stream is responsible for executing the control flow graph of the program and sending direction to the fetch unit of access and execute streams. Since the branches are independently resolved much earlier, the other two streams are non-speculative. However, difficulty in effective separation of the program is one of the prominent problems associated with this form of a decoupled architecture.

Reliability and functional correctness

Recent proposals used decoupling to improve the reliability and functional correctness of the processors. In a conventional design, the reliability against errors due to particle strikes or circuit glitches is conservatively built in the hardware. Dynamic Implementation Verification Architecture (DIVA) is one of the first major proposals that decouple the implementation of correctness guarantee circuitry from each individual hardware structures to a separate functional checker unit (as shown in Figure 2.5) [Aus99]. An out-of-order processor pipeline (referred to as DIVA core) is used to execute the instructions. All the completed instructions along with their operands, input values, and results are sent to the functional checker in the program order. The functional checker re-executes these instructions and validates the results passed by the DIVA core with the new results. Only the correct results commit to the architectural state. Any error detected in the result is corrected along with full recovery of DIVA core pipeline from the next instruction onward. Since by the time the instruction is ready for functionality check, all the control and data dependences for that instruction had already been resolved and all the inputs needed for the execution of an instruction are available, the checker unit is both faster and

simpler at the same time. This is because; it does not require a complex out-of-order execution engine (as shown in Figure 2.3) for performance improvement. This allows the design effort to be concentrated on a single smaller and simpler checker unit to make it more robust and reliable, lifting the burden of ensuring correctness from the complex out-of-order DIVA core. This can reduce the overall design effort, because it is difficult and more time consuming to design and verify large complex structures for 100% correctness.

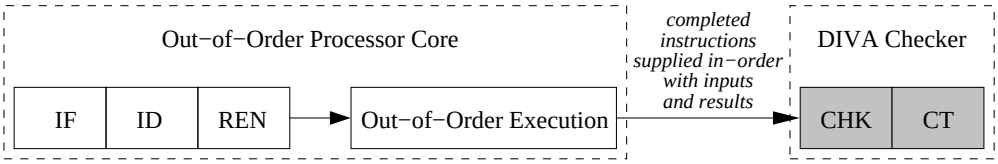


Figure 2.5: Block diagram of an out-of-order processor core, as shown in Figure 2.3, modified with a robust DIVA checker unit.

2.6 Summary

In explicitly decoupled architecture, our aim is to explicitly decouple the requirement for correctness from a design only for performance. In this thesis, we try to achieve this level of decoupling by communicating through hints between the performance and correctness domains. This will allow true focus on the common case and reduction of design complexity, which in turn permits the implementation of ideas previously deemed impractical (including the ones discussed in this chapter). Next, we will discuss how concepts of explicitly decoupled architecture can be used to implement an efficient mechanism to track and enforce memory dependences.

Chapter 3

Decoupled Memory Dependence Enforcement

Due to their limited ability of buffering in-flight instructions, current out-of-order processors fail to hide very long latencies such as those of off-chip memory accesses. This limitation stems from the fact that scaling up buffering structures usually incur latency and energy penalties that undermine the overall design goal. With the popularization of multi-core products, the precious off-chip bandwidth will be subject to more contention and in turn further exacerbate memory latencies. Thus, latency tolerance will continue to be an important design goal of microarchitecture.

In this chapter, we focus on the memory disambiguation and forwarding logic. To orchestrate the out-of-order execution of a large number of in-flight instructions, an efficient mechanism to track and enforce memory dependency is imperative. The conventional approach, as discussed in Section 2.3, uses age-based queues – often collectively referred to as the load-store queue (LSQ) – to buffer and cross-compare memory operations to ensure that the out-of-order execution of memory instructions does not violate program semantics under a certain coherence and consistency model. The LSQ buffers memory updates, commits their effects in-order, forwards values between communicating pairs of load and store, and detects incorrect speculation or potential violation of coherence and consistency. Due to the complex functionality it implements and the fact that the address for a load or a store needs to be matched against those

of the opposite kind in an associative manner and matching entries need to go through time-consuming priority logic, the LSQ is perhaps the most challenging microarchitectural structure to scale up.

Using the principles of explicitly-decoupled architecture, we propose a novel design that moves away from the conventional load-store handling mechanism that strives to perform most accurate forwarding in the first place and proactively detect and recover from any dependence violation. Instead, we adopt a very passive, "slackened" or decoupled approach to forwarding and violation detection. First, we use an L0 cache to perform approximate, opportunistic memory forwarding at the execution stage, hoping that most loads would actually get correct data. In this stage, there is minimum interference to instruction execution. Second, we also avoid relying on proactive monitoring to detect violation. Our correctness guarantee comes from an in-order re-execution of memory accesses. Discrepancy between the two different executions triggers a replay. Such a decoupled approach not only makes it easier to understand, design, and optimize each component individually, it is also effective. Without any associative search logic, in a processor that can buffer 512 in-flight instructions, even a naive implementation of our design performs similarly with a traditional LSQ-based design with optimistically-sized queues. The performance is further improved by using optional optimizations and approaches that of a system with ideally-scaled LSQ: on average, about 2% slower for integer applications and 4.3% slower for floating-point applications. Furthermore, both naive and optimized design scale very well.

The rest of the chapter is organized as follows: Section 3.1 describes the basic design and optional optimization techniques; Section 3.2 describes the experimental methodology; Section 3.3 provides some quantitative analysis; and Section 3.4 concludes.

3.1 Slackened Memory Dependence Enforcement (SMDE)

In an attempt to simplify the design of memory dependence enforcement logic, we adopt a different approach from conventional designs and recent proposals (Section 2.3). Instead of achieving the two goals of high performance and correctness by encouraging a highly out-of-order execution of memory instructions and *simultaneously* imposing a proactive monitoring

and interference of the execution order, we use two decoupled executions for memory instructions, each targeting a different goal: a highly out-of-order *front-end* execution with little enforcement of memory dependency to keep the common-case performance high, and a completely in-order *back-end* execution (with no speculation) to detect violations and ensure program correctness. Separating the two goals allows a much more "relaxed" implementation (hence the name) which not only mitigates the escalating microprocessor design complexity, but can lead to opportunities a more sophisticated and proactive approach can not provide. For example, our design can effortlessly allow an arbitrarily large number of in-flight memory instructions. Such benefits can offset the performance loss due to the simplicity of the design.

In the following, we first describe the basic architectural support to ensure functional correctness of the design. This results in a naive, simplistic design that is hardly efficient. Yet, as we will show later, due to the ability to allow any number of memory instructions to be in-flight at the same time, even this naive version of SMDE can perform similarly as a system with optimistically-sized load-store queue (but is otherwise scaled up to buffer more in-flight instructions). We then discuss simple, optional optimization techniques that address some of the performance bottlenecks. With these optimizations, the design can perform close to idealized load-store queue.

3.1.1 A Naive Implementation of SMDE

Figure 3.1 is the block diagram of the naive design, which shows the memory hierarchy and high-level schematic of the processor pipeline. For memory instructions, a front-end execution is performed out of program order like in a normal processor at the execution stage. However, in a naive SMDE, memory instructions are issued *entirely* based on their register dependences. No other interference is imposed. As a result, it is possible to violate memory dependences and this makes the front-end execution fundamentally speculative (and it is treated as such). Thus the L0 cache accessed in the front-end execution does not propagate any result to L1. To detect violations, memory accesses are performed a second time, totally in-order at the commit stage of the pipeline. Any load that obtains different data from the two executions will take the result of the back-end execution and trigger a squash and replay of subsequent instructions.

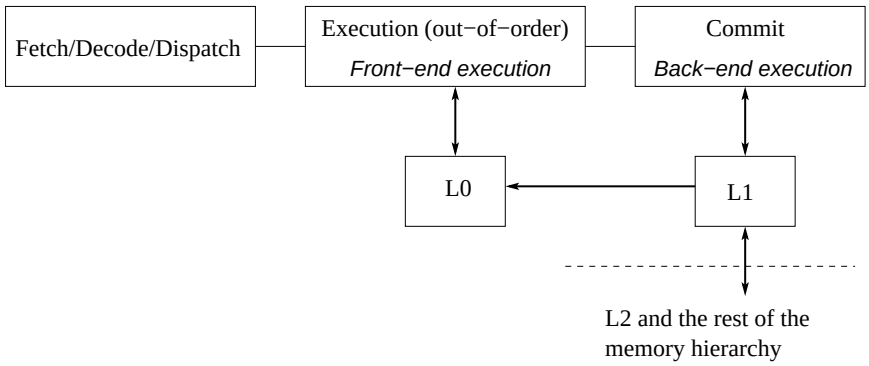


Figure 3.1: Diagram of the pipeline and memory hierarchy of the proposed design.

From one perspective, the only execution in our design that is absolutely required is the back-end execution. In theory, therefore, *any* front-end execution scheme would work (even one that only returns garbage values). This relieves the design burden to maintain correctness for the front-end execution, effectively relegating it to a value predictor. When we simply issue memory instructions based on the register dependences and completely disregard their memory dependences, the front-end execution effectively becomes a slow but very accurate value predictor: on average, about 98% of loads obtain a correct value.

From another perspective, the front-end execution can afford to simplify the memory dependence enforcement logic because the back-end execution provides a safety net for incorrect speculation. We push that simplification to the extreme by completely eliminating any enforcement logic. Such a laissez-faire approach works for normal programs as our empirical observation shows that about 99% of loads happen in the "right" time, that is, after the producer has executed, but before the next writer executes.

The primary appeal of SMDE is its simplicity. The entire framework is conceptually very straightforward and it is rid of the conventional load-store queue using priority CAM logic or any form of memory dependence prediction, thus avoiding the scalability problem or circuit complexity issues. Furthermore, the two execution passes are entirely decoupled, making the design modular. The dedicated back-end execution offers a large degree of freedom to the design of the front-end execution. Any rare incidents such as race conditions, corner cases, or even concerns for soft errors can be completely ignored for design simplicity. We will discuss examples later.

3.1.1.1 Front-end execution

Central to our architectural support is an unconventional, speculative L0 cache. As explained earlier, at the issue time of a load, we simply access this L0 cache. Meanwhile, we also allow stores to write to this L0 cache as soon as they execute. Since the L0 cache is used to handle the common cases, we want to keep its control extremely simple. No attempt is made to clean up the incorrect data left by wrong-path instructions. When a line is replaced, it is simply discarded, even if it is dirty. And in fact, no dirty bit is kept, nor is the logic to generate it. Keeping track of and undoing wrong-path stores and preventing cast-outs undoubtedly complicate the circuitry and undermine the principle of keeping the common case simple and effective.

Initially, it would appear that such an unregulated L0 cache may be quite useless as load may access the cache before the producer store has written to the cache or after a younger store has, and the data could be corrupted in numerous ways: by wrong-path instructions, due to out-of-order writes to the same location, and due to lost updates by eviction. However, our value-driven simulation shows that an overwhelming majority (98% on average) of loads obtain correct values. Upon closer reflection, this is not as surprising as it appears. First, compilers do a good job in register allocation and memory disambiguation. This means that close-by store-to-load communications are infrequent in normal programs. Far apart store-load pairs are very likely to execute in the correct order. Second, write-after-read (WAR) violations are also infrequent: a load followed closely by a same-location store are very likely to be part of a load-operate-store chain, whose execution order will be enforced by the issue logic. Third, data corruptions in the L0 cache are not permanent – they are naturally cleansed by new updates and evictions.

Recall that in a conventional load-store queue, forwarding data from store queue needs associative search followed by priority selection. Additionally, the virtual address needs to be translated before used to search the store queue, otherwise, the forwarding can be incorrect due to aliasing. However rare it is, a partial overlap (where a producer store only writes part of the loaded data) has to be detected. Therefore the logic has to be there and it is invoked every time the queue is searched. In stark contrast, the L0 is simply accessed like a cache. As mentioned before, thanks to the back-end execution, the concern about these corner case situations is that

of performance, not of correctness, and we can safely ignore them in the front-end.

3.1.1.2 In-order back-end execution

In-order re-execution to validate a speculative execution is not a new technique [GGH91; CL04]. However, the extra bandwidth requirement for re-execution makes it undesirable or even impractical. Prior proposals address this issue by monitoring the earlier, speculative execution order, reasoning about whether a re-execution is necessary, and suppressing unnecessary re-executions [GGH91; CL04; Rot05].

A small but crucial difference between our design and this prior approach is that we do not rely on avoiding re-execution. Instead, our dual-cache structure naturally and easily provides the bandwidth needed for the re-execution. Although misses from L0 still access L1 and thus increase the bandwidth demand, this increase is (at least partly) offset by the fact that most wrong-path loads do not access L1 for back-end execution.

The benefit of faithful reload without any filtering is concrete, especially in enforcing coherence and consistency. For modern processors, cache coherence is a must (even in uniprocessor systems, as requiring the OS to maintain that cache is coherent with respect to DMA operation is undesirable). Regardless of how relaxed a memory consistency model the processor supports, cache coherence alone requires careful handling to ensure store serialization: if two loads accessing the same location are re-ordered and separated by an invalidation to that location, the younger load must be replayed. Hence the memory dependence enforcement logic needs to heed every invalidation message, not just those that reach the L1 cache (as cache inclusion does not extend to in-flight loads captured in the load queue). Faithfully re-executing every memory access in-order greatly simplifies coherence and consistency considerations.

Figure 3.2 shows the relationship of the logical structures involved in back-end execution. An age-ordered *memory operation sequencing queue* (MOSQ) is used to keep the address and data of all memory operations generated during the front-end execution. During back-end execution, the recorded address is used to access the L1. Note that the back-end execution is not a full-blown execution, but only a repeated memory access. There is no re-computation of the address, hence no need for functional units or register access ports. For loads, the data returned

from the L1 is compared to that kept in the MOSQ. A difference will trigger a replay. A successful comparison makes the load instruction ready for commit. For stores, the data kept in the MOSQ is written to the L1 cache when the store is committed. The back-end execution is totally in-order, governed by the execution pointer shown in Figure 3.2. Note that for store instructions, re-execution can only happen when they are committed. Essentially, when pointing to a store, the back-end execution pointer (BEEP) is "clamped" to the commit pointer. For load instructions, however, BEEP can travel ahead of the commit pointer allowing reload to happen before commit.

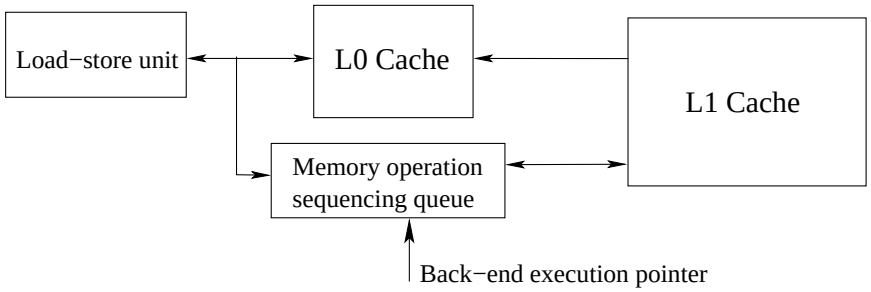


Figure 3.2: In-order back-end execution via the memory operation sequencing queue.

3.1.1.3 Scope of replay

The scope of instructions to be replayed is similar to that in a conventional design [Com00] with an important difference. We do not re-execute the load instruction that triggers the replay. Instead, we only re-execute instructions younger than the load. The reason is two-fold (we *can* and we *must*). First, unlike in a conventional system where the detection of a memory order violation merely suggests that the load obtained potentially incorrect data without providing the right result, in our system, the value from the reload is always correct and can be used to fix the destination register of the load instruction. Second, it is necessary *not* to re-execute the load to avoid the rare but possible infinite loop. When a load is replayed in a conventional system, it will eventually become safe from further replay (*e.g.*, when it becomes the oldest in-flight memory instruction). Therefore, forward progress is always guaranteed. In our system, we do not place *any* constraint on the front-end execution and thus can not expect any load to be correctly processed regardless of how many times it is re-executed. Furthermore, a replay is triggered

by the difference between two loads to two different caches at different time. Fundamentally, there is no guarantee that the two will be the same. In the pathological scenario, disagreement between the two can continue indefinitely. For example, when another thread in the parallel program is constantly updating the variable being loaded.

The replay trap is handled largely the same way as in a conventional design [Com00]. The only additional thing to perform is to fix the destination register of the triggering load.

3.1.1.4 L0 cache cleanup

In addition to the basic operation of replay, we can perform some *optional* cleanup in the L0 cache. We emphasize again that we do not require any correctness guarantee from the front-end execution, so the difference between cleanup policies is that of performance, not correctness.

Intuitively, by the time a replay is triggered, the L0 cache already contains a lot of future data written by in-flight stores that will be squashed. Thus we can invalidate the entire cache and start afresh. This eliminates any incorrect data from the L0 cache and reduces the chance of a future replay. The disadvantage, of course, is that it also invalidates useful cache lines and increases the cache miss rate. This has its own negative effects, especially in modern processors, which routinely use speculative wakeup of dependents of load instructions. When a load misses, this speculation fails and even unrelated instructions may be affected. For example, in Alpha 21264, instructions issued in a certain window following a mis-speculated load will be squashed and restart the request for issue [Com00]. Thus, whole-cache invalidation can be an overkill.

A second alternative, single-flush, is to only flush the line accessed by the replay-triggering load. The intuition is that this line contained incorrect data at the time the load executed and probably still contains incorrect data. This approach stays on the conservative side of L0 cleanup and thus will incur more replays than whole-cache invalidation. However, it does not affect the cache miss rate as much and it is simpler to implement.

A third alternative, which we call selective flush, comes in between the two extremes. In addition to flushing the line that triggered the replay, we can invalidate all the cache lines written to by the squashed stores. This can be done by traversing the MOSQ and using the address of the stores that have finished execution to perform the invalidation.

For implementation simplicity we use single-flush. Note that not flushing any line is also an option, but the replay rate is much higher than single-flush, which makes it unattractive. We will show relevant statistics of different replay strategies in Section 3.3. As we will see, single-flush actually outperforms other alternatives.

3.1.2 Performance Optimizations

In the naive implementation of SMDE, we select the most straightforward implementation in each component. Such a simplistic design leaves several opportunities for further optimization. We discuss a few techniques here.

3.1.2.1 Reducing replay frequency with the fuzzy disambiguation queue

Although an incorrect load using the L0 cache will not affect program correctness, it does trigger a replay which can be very costly performance-wise. Hence, simple mechanisms that help reduce the replay frequency may be desirable. Our analysis shows that a large majority of replays are triggered directly or indirectly (a replay rolls back the processor and thus leaves some future data in the L0 cache which can cause further replays) by violation of read-after-write ordering. In these cases, oftentimes, while the data of the producer store is unavailable, the address is ready. In other words, if we keep track of address information, we can reject some premature loads as in a conventional design [TDF+02] and reduce the replay rate. For this purpose, we introduce a *Fuzzy Disambiguation Queue* (FDQ).

Each entry of the FDQ contains the address of a store and the age of the instruction (Figure 3.3). A straightforward representation of the age is the ROB entry ID plus one or two extra bits to handle the wrap-around[1]. Since we only keep track of stores between address generation and execution, we allocate an entry at address generation and deallocate it when the data is written to the L0 cache.

When a load executes, it sends the address and the age through one *load probe port* to probe the FDQ. Out of all the valid entries, if any entry matches the address of the load and has an

[1] Although one bit is sufficient, a two-bit scheme is easier to explain. The most-significant two bits of the age get increased by 1 every time the write pointer of the ROB wraps around. When comparing ages, these two bits follow: $0 < 1$, $1 < 2$, $2 < 3$, and $3 < 0$.

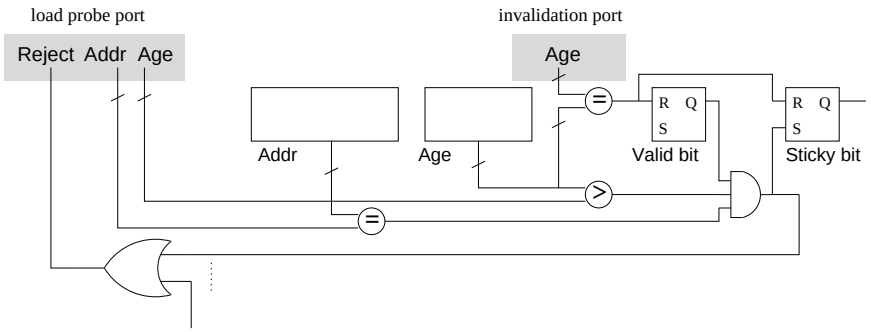


Figure 3.3: The fuzzy disambiguation queue. For clarity, the figure shows one entry of the queue with one load probe port and one invalidation port.

older age, we reject the load (Figure 3.3), which will be retried after a certain delay. At the same time a *sticky bit* is set to prevent the displacement of the matching entry. We do this because given limited space, we want to maximize the number of "useful" entries in the queue. When we allocate an entry in the FDQ, if there is no invalid entry, we randomly evict a valid entry to make room for the new store. If all entries have the sticky bit set, no entry is allocated. When a store executes, it attempts to deallocate its FDQ entry, if any. This is done by sending its age through the *invalidation* port to clear the valid bit of any matching entry (Figure 3.3). When a branch misprediction is detected, we can flash-clear all valid bits to eliminate orphan entries (entries whose owners have been squashed). These orphan entries must be eliminated because once they match a load, they will perpetually reject the load causing a deadlock. We note that for a slightly more complex circuit, we can use the age of the mispredicted branch to selectively invalidate only entries younger than the branch. This invalidates all the orphan entries but keeps other entries intact. Finally, if during issue, stores also probe the FDQ, we can similarly reject "premature" stores to prevent write-after-write violations.

Note that an FDQ differs significantly from an store queue in numerous ways. An FDQ does not have a priority logic: We are only interested in *whether* there is an older store to the same location pending execution, not *which* store. There is no data forwarding logic in an FDQ either. Data is provided by the L0 cache alone (no multiplexer is involved). Since FDQ is an optional filtering mechanism, it is not subject to the same requirements of an store queue for correctness guarantee and hence there is quite a bit of latitude in its design. There is no need to try to accommodate all stores and therefore there is little scalability pressure. Thanks in part to

the shorter life span of an entry (from address generation of a store to actual writing to the L0 cache), a small queue is sufficient for reducing the replay rate. The address used does not need to be translated nor is it necessary to use all address bits.

In contrast to prediction-based memory dependence enforcement [SMR05; SWF05], our FDQ-based mechanism, though perhaps less powerful, is much more straightforward. First, the operation remains entirely address-based. With the addresses, dependence relationship is clear and unambiguous, thus we only need one structure. Memory dependence prediction, on the other hand, requires a multitude of tables, some of which very large and fully associative and many tables have a large number of ports. Second, while it is relatively easy to predict the presence of dependence, it is more challenging to pin-point the producer instruction [MS97]. Thus, to ensure accurate communication, additional prediction mechanisms [SMR05] or address-based mechanisms [SWF05] are used. Third, once a dependence is predicted, it still needs to be enforced. Memory dependence prediction designs [MS97; SMR05; SWF05] depend on the issue logic to enforce the predicted memory-based dependence in addition to the register dependence. We use the simpler rejection and retry method to keep the design less intrusive and more modular.

In summary, the intention of using FDQ is not to rival sophisticated prediction or disambiguation mechanisms in detecting and preventing dependence violations. It is meant as a cost-effective mechanism to mitigate performance loss in an SMDE system.

3.1.2.2 Streamlining back-end execution

When a load reaches the commit stage, even if it has finished the front-end execution, it may still block the stage if it has not finished the back-end execution (reload). This will reduce the commit bandwidth and slow down the system. Ideally, the back-end execution pointer (BEEP) should be quite a bit ahead of the commit pointer, so that reload and verification can start early and finish before or when the instruction reaches the commit stage. Unfortunately, a store can only execute when the instruction is committed. This means, every time BEEP points to a store instruction, the back-end execution will be blocked until the commit pointer “catches up”.

To streamline the back-end execution, we can employ the oft-used write buffer to allow

stores to start early too, which indirectly allows loads to start early. If the processor already has a write buffer (to temporarily hold committed stores to give loads priority), then the buffer can be slightly modified to include a “not yet committed” bit that is set during back-end execution and cleared when the store is committed. Naturally, the write buffer has to provide forwarding to later reloads. Also, when handling memory synchronizers such as a write barrier or a release, the content of the write buffer has to be drained before subsequent reload can proceed. In our design, L0 cache fills do not check the write buffer (for simplicity).

Write buffering always has memory consistency implications. The sequential semantics require a reload to reflect the memory state after previous stores have been performed. A load must also reflect the effects of stores from other processors that according to the consistency model, are ordered before itself. In a sequentially consistent system, stores from all processors are globally ordered. Therefore, when an invalidation is received, it implies that the triggering store (from another processor) precedes all the stores in the write buffer and transitively precede any load after the oldest store in the write buffer. Thus, if BEEP has traveled beyond the oldest store, we need to reset it to the oldest store entry in the MOSQ and restart from there. In a processor with a weaker consistency model, stores tend to be only partially ordered, relaxing the need to roll back BEEP. For example, in a system relying on memory barriers, when we receive an external invalidation, we only need to roll back BEEP to restart from after the oldest memory barrier.

3.1.2.3 Other possibilities of replay mitigation

While the FDQ successfully addresses the largest source of replays (Section 3.3.2), there are other techniques to further reduce replay frequency or to mitigate their performance impact. Some of these can be quite effective in reducing replays. It is also possible to use a write buffer without forwarding capabilities and stall a reload where the write buffer *potentially* contains an overlapping store. We discuss these possibilities here and show some quantitative analysis analysis later in Section 3.3.5.

Replay suppression

When a replay is triggered, the common practice is to discard all instructions after the triggering instruction. This may throw out a lot of useful work unnecessarily. Within the instructions after the triggering load, only the dependents (direct or indirect) need to be re-executed. Therefore a selective replay is conceivable. We modeled selective replay and found that it is useful on top of the naive design. When replay frequency is reduced by optimization techniques, however, the benefit naturally reduces as well. In current microarchitectures, it is not easy to support selective replay. However, there is one degenerate case of it that could be supported: when the triggering load has no in-flight dependents, we only need to fix the destination register and the replay can be suppressed. Our simulations show that such load instructions are not rare: about 14% on average, and can be as high as 100%. Intuitively, it makes sense to detect these cases and avoid replays.

It is quite easy to track which load instruction has no in-flight dependents: At register allocation, we can maintain a bit vector, each bit corresponding to one *physical* register to indicate whether there is an in-flight instruction sourcing from it. Every instruction upon renaming will set the bits corresponding to the (renamed) source registers. When a load instruction is decoded, we reset the bit corresponding to the load's destination physical register. When a load triggers a replay and its destination register's bit is not set, we know there is no dependents in-flight and we can suppress the replay.

Age-based filtering of L0 cache

When the processor recovers from a branch misprediction or a replay, the L0 cache is not proactively cleansed to remove pollution left by squashed instructions. This results in a period when the front-end execution is very likely to load a piece of incorrect data, triggering a replay. One option to mitigate the impact is to filter out some of the pollution by keeping track of the age of the data in the L0 cache: A store updates the age of the cache line and a load checks the age. If the age of the cache line is younger than that of the load, it is probable that the data is left by stores squashed due to misprediction or replay. In that case, going directly to the L1 cache may be a safer bet than consuming the presumably polluted data. With a FDQ, we are already

generating age IDs for memory operations. So tracking the age in the L0 requires only minimal extra complexity.

The age-tracking of L0 cache works as follows. The L0 cache's tag field is augmented with an age entry. A store instruction updates the cache line's age, whereas a load instruction compares its age with the age stored in the cache line at the same time tag comparison is being done. When the age of the load is older, we simply "fake" an L0 cache miss and access the L1. Note that here the age tracking is only a heuristic, therefore the tracking is not exact. The granularity is at the cache line level and we do not enforce the age to monotonically increase. Indeed, when out of order stores happen, age can decrease. Recall that we do not require any correctness guarantee from the front end, so we conveniently choose any tracking to reduce complexity. However, since we are using a finite number of bits to represent age, we need make sure that an old age ID is not misinterpreted as a new one in the future – otherwise, we may incur too many unnecessary misses in the L0. This can be easily achieved: When we extend the ROB ID by 2 bits to represent age, we reserve a coding of the 2-bit prefix as "old". For example, we can cyclically assign 1, 2, and 3 as the prefix for age and reserve prefix 0. An age ID with prefix 0 is older than another age with a non-zero prefix. Thus, when the commit pointer of ROB wraps around, we know that all instructions with a certain prefix (say 2) have committed and that prefix 2 will be recycled for a later time period, we then reset those entries whose age have a prefix of 2 to 0. Thus, these entries will not be incorrectly treated as pollution. When a cache line is brought in by a load, its age prefix is also set to 0.

3.1.3 Differences Between SMDE and Other Approaches

Recently, many techniques were proposed to address the load-store queue scalability issue in one way or another. While most, if not all, techniques focus on providing a scalable alternative design that rivals an ideally scaled load-store queue, our main focus is to reduce the conceptual and implementation complexity. Besides the difference in focus, the mechanisms we use in the speculation and the recovery from mis-speculation also differ from this prior work.

In a two-level disambiguation approach [TIVL05; BZ04; Rot04; ARS03; GAR+05], the fundamental action is still that of an exact disambiguation: comparing addresses and figuring

out age relationship to determine the right producer store to forward from. The speculation is on the *scope* of the disambiguation: only a subset of the stores are inspected. In contrast to these, in our front-end execution, we allow the loads to blindly access the cache structures. Our speculation is on the *order* of accesses: if left unregulated, the relative order of loads and stores to the same address is largely correct (same as program order).

Two recent designs eliminate fully-associative load queue and store queue [SMR05; SWF05]. They rely on dependence prediction to limit the communication of load to only a few stores. This is still a form of scope speculation – the scope of the stores communicating to a particular load is reduced to the extreme of one or a handful. Although both this approach and ours achieve an load-store queue free implementation, the two styles use very different design tradeoffs. While memory dependence is found to be predictable, *pin-pointing* the exact producer of a load instruction, on the other hand, is very ambitious and requires numerous predictor tables. Such a design also requires support from the issue logic to enforce the predicted dependence. Our design performs a different speculation and thus does not require the array of predictors or the extra dependence enforcement support from the issue logic.

The use of a small cache-like structure in architecture design is very common. However, the way our L0 cache is used is very different from prior proposals. In [SWF05], great care is taken to maintain the access order of cache, so that no corruption will take place, even under branch mispredictions. In [MS97], the *transient value cache* is intended to reduce bandwidth consumption of stores that are likely to be killed. It contains only architecturally committed stores and is fully associative. In contrast to both these small caches, our L0 cache is fundamentally speculative. No attempt is made to guarantee the absence of data corruption. Indeed, corruptions of all sorts are tolerated in our L0 cache: out of order updates, wrong-path updates, lost updates due to eviction, virtual address aliasing, or even soft errors due to particle strikes.

For mis-speculation detection and recovery, we use an existing technique: in-order re-execution [GGH91; CL04]. However, the extra bandwidth consumption is a problem associated with this technique [Rot05]. While prior approaches rely on reducing re-access frequency [CL04; Rot05; SMR05], our dual cache structure lends itself well to provide that extra bandwidth demand effortlessly, thereby avoiding the problem.

Finally, we note that two-pass execution is also used in entirely different contexts such as for whole-program speculation [SPR00] and for fault-tolerance [Aus99]. Our design shares the same philosophy of relying on another independent execution to detect problems. However, because of the different context, we do not re-execute the entire program nor attempt to repair all the state.

In all, our SMDE design employs novel speculation strategies and structures and uses existing techniques with new twists to solve problems in a simpler way. As we show in Section 3.3, although our target is design simplicity, the performance of an optimized version comes very close to that of an idealized load-store queue system.

3.2 Experimental Setup

To conduct experimental analyses of our design, we use a heavily modified SimpleScalar [BA97] 3.0d. On the processor model side, we use separate ROB, physical register files, and load queue/store queue. We model load speculation (a load issues despite the presence of prior unresolved stores), store-load replay [Com00], and load rejection [TDF+02]. A rejected load suppresses issue request for 3 cycles. When a store-load replay happens, we model the re-fetch, re-dispatch, and re-execution. When simulating a conventional architecture, unlike what is done in the original simulator, we do not allocate an entry in the load queue for prefetch loads, *i.e.*, loads to the zero register (R31 or F31).

We also extended the simulator to model data values in the caches. Without modeling the actual values in the caches, we will not be able to detect incorrectly executed loads. We model the value flow very faithfully. For example, when a load consumes an incorrect value due to mis-speculation and load from wrong address, its pollution in the L0 cache is faithfully modeled. When a branch consumes a wrong value and arrive at a wrong conclusion, we force the simulator to model the (spurious) recovery.

We also increased the fidelity in modeling the scheduling replay [Kes99; Com00]. A scheduler replay is related to speculative wakeup of dependent instructions of a load [KL04]. In the Alpha 21264 scheduler logic, when the load turns out to be a miss, all instructions issued during a so-called shadow window are pulled back to the issue queue. We model after Alpha

21264: there is a two-cycle shadow window in the integer scheduler, but in the floating-point issue logic, this is handled slightly differently. Only dependents of loads need to be pulled back [Com00].

Our consistency model is after that of the Alpha21264 processor as well. Since we are simulating sequential applications with no write barriers and there is no bus invalidation traffic, the only thing to note is that at the time a store is committed, if it is a cache miss, we keep the data in the store queue (in the conventional configurations) or the write buffer (if any, in the SMDE configurations) and allow the store to be committed from the ROB [Com00]. Note that in a naive SMDE implementation, there is no write buffer, so the store is not committed until the cache miss is serviced.

Processor core	
Issue/Decode/Commit width	8 / 8 / 8
Issue queue size	64 INT, 64 FP
Functional units	INT 8+2 mul/div, FP 8+2 mul/div
Branch predictor	Bimodal and Gshare combined
- Gshare	1M entries, 20 bit history
- Bimodal/Meta table/BTB entries	1M/1M/64K (4 way)
Branch misprediction penalty	7+ cycles
ROB/Register(INT,FP)	512/(400,400)
LSQ(LQ,SQ)	112(64,48) - 1024(512,512), 2 search ports
	1 cycle port occupancy, 2-cycle latency
Memory hierarchy	
L0 speculative cache	16KB, 2-way, 32B line, 1 cycle, 2r/2w ports
L1 instruction cache	32KB, 2-way, 64B line, 2 cycles
L1 data cache	64KB, 2-way, 64B line, 2 cycles, 2r/2w ports
L2 unified cache	1MB, 8-way, 128B line 10 cycles
Memory access latency	250 cycles

Table 3.1: System configuration.

Our quantitative analyses use highly-optimized Alpha binaries of all 26 applications from the SPEC CPU2000 benchmark suite. We simulate half a billion instructions after fast-forwarding one billion instructions. The simulated baseline conventional processor configuration is summarized in Table 3.1. To focus on dynamic memory disambiguation, we size the ROB and register files aggressively, assuming optimization techniques such as [TA03] are used.

3.3 Experimental Analysis

3.3.1 Naive Implementation

We start our experimental analysis with the comparison of IPCs (instruction per cycle) achieved in a baseline conventional system (with LSQ) and in a naive implementation of SMDE. We use a baseline conventional design with optimistically-sized queues: 48 entries in SQ and 64 entries in the LQ. Note that, in a high-frequency design, supporting a large number of entries in LQ and SQ is indeed challenging. Even the increase of the SQ size from 24 to 32 in Intel's Pentium 4 processor requires speculation to ease the timing pressure [BBH+04].

From Figure 3.4, we see that *on average*, the naive SMDE system performs slightly worse than the baseline (-5.33% for INT and -2.76% for FP). While naive SMDE suffers from performance degradation due to replays, it does not have any limit on the number of in-flight memory instructions and this can offset the performance degradation due to replays. In general, floating-point applications tend to benefit from a large number of in-flight instructions and thus tend to perform better in an SMDE system than in a conventional one. Bear in mind that Figure 3.4 shows the naive design, which, though functional, is hardly an efficient implementation of an SMDE paradigm. Yet, we can see that even this simplistic design achieves an acceptable performance level. Given its simplicity, this result is very encouraging.

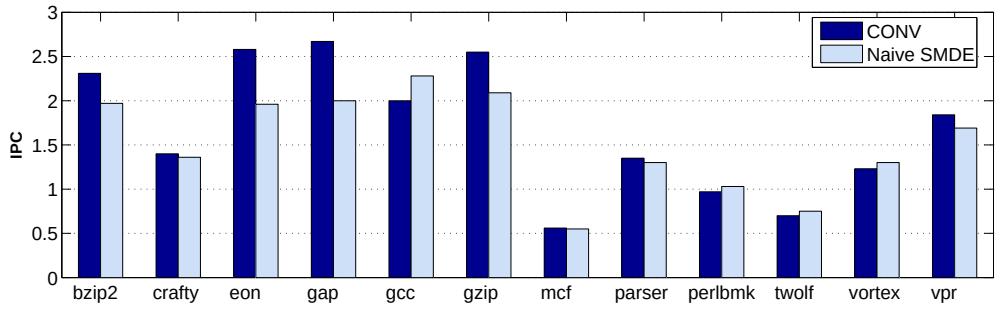
In the above and the following discussion, we use a single-flush policy because of its simplicity (Section 3.1.1.4). In Section 3.3.5, we show some details about the different flush policies.

3.3.2 Effectiveness of Optimization Techniques

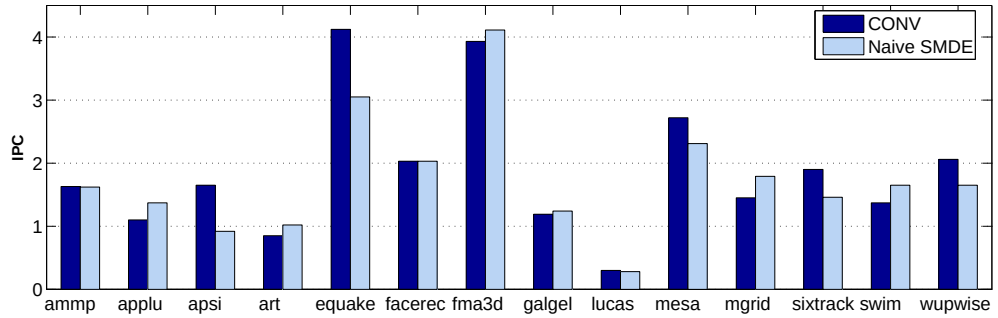
Although the naive design eliminates any stall time due to LSQ fill-up, it introduces other performance degradation factors. We now look at the effect of the mitigation techniques.

Back-end execution bottleneck

Recall that in the back-end execution of the naive implementation, a store blocks the advance of the execution pointer until commit. This makes the latency of subsequent reloads more likely to



(a) Integer applications.

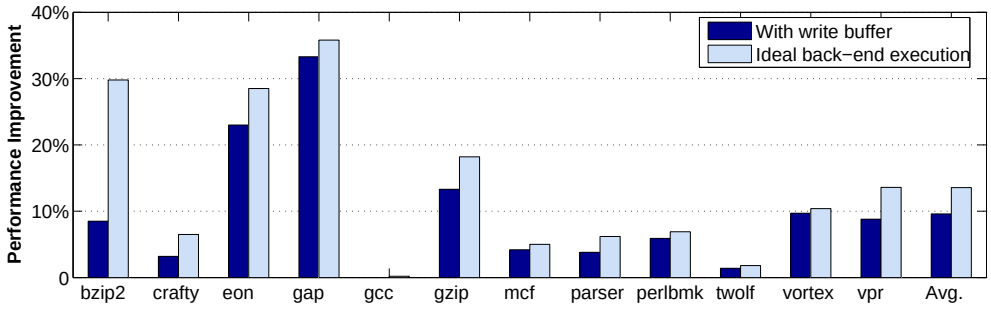


(b) Floating-point applications.

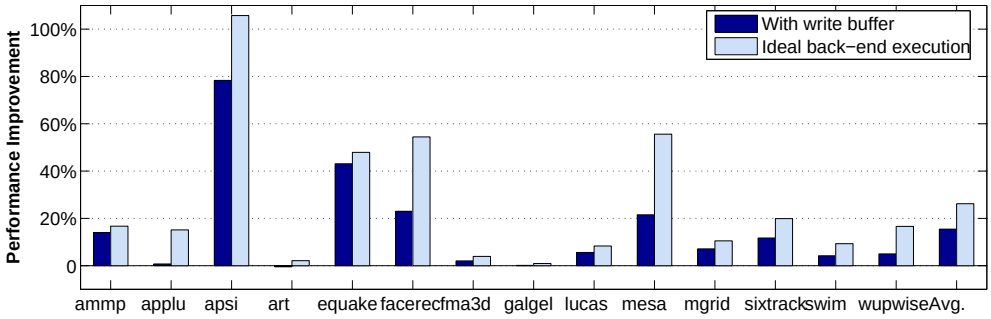
Figure 3.4: Comparison of IPCs in the baseline conventional system (CONV) and a naive SMDE system.

be exposed, reducing commit bandwidth. In Figure 3.5, we show the performance improvement of using the write buffer described in Section 3.1.2.2. We also included a configuration with an idealized back-end execution where the latency and L1 cache port consumption of re-executing loads and stores are ignored. (Of course, replays still happen.) In this configuration, only the baseline processor's commit bandwidth and whether an instruction finishes execution limit the commit. All performance results are normalized to the naive SMDE configuration (without a write buffer).

The first thing to notice is that there is a very significant difference between the naive and the ideal implementation of the back-end execution. For example, in the case of *apsi*, when removing the restriction placed by the back-end execution, the performance more than doubles. Second, an 8-entry write buffer is able to smooth out the re-execution of loads and stores and provide a performance improvement of 10-15%, very close to that of an ideal implementation. Although for a few applications, such as *bzip2* and *mesa*, there is still room for significant improvement.



(a) Integer applications.



(b) Floating-point applications.

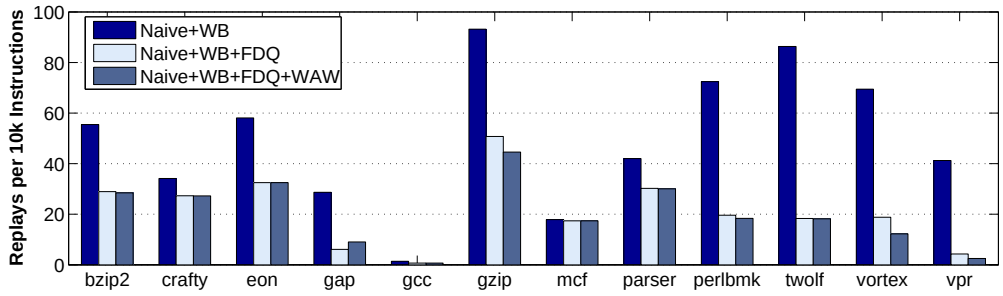
Figure 3.5: Performance impact of using an 8-entry write buffer in the back-end execution and of having an ideal back-end execution.

Replay frequency reduction

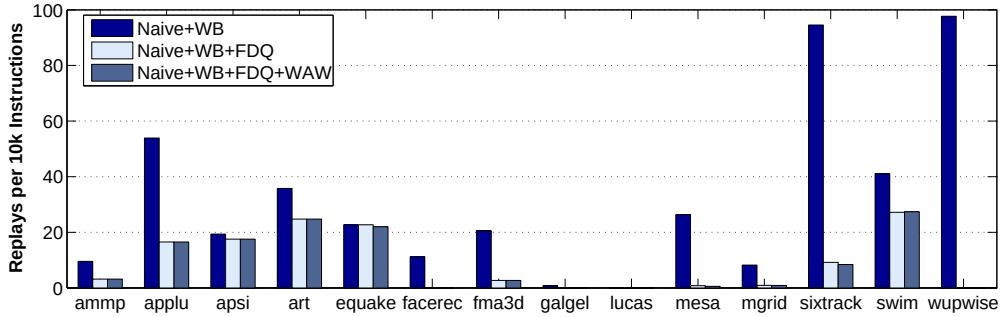
Figure 3.6 shows the frequency of replays before and after enabling a 16-entry FDQ (Section 3.1.2.1). We also use FDQ to detect and reject out-of-order stores to prevent write-after-write (WAW) violations. In this study, all configurations have an 8-entry write buffer in the back-end.

The replay frequency varies significantly from application to application and can be quite high in some applications. After applying the FDQ, the replay frequency drastically reduces for many applications. In *wupwise* for example, the frequency changes from about 97.7 replays per 10,000 instructions to no replays at all in the 500 million instructions simulated. We see that using the FDQ to detect WAW violations have a much smaller impact and can, in some cases, lead to a slight increase in the replay rate. However, the overall effect is positive. In the following, when we use FDQ, by default we detect WAW violations too.

In Figure 3.7, we present another view of the effect of using various optimization tech-



(a) Integer applications.



(b) Floating-point applications.

Figure 3.6: Replay frequency under different configurations.

niques: the breakdown of replays into different categories. A replay is caused by loading a piece of incorrect data from the L0 cache. This can be caused by memory access order violations (RAW, WAW, and WAR) in the front-end execution. A replay can also be caused by L0 cache pollution when the execution is recovered from a branch misprediction or a replay, or due to a cache line eviction. A replay can have multiple causes. For the breakdown, we count a replay towards the last cause in time. Therefore, when a technique eliminates a cause for one load, it may still trigger a replay due to a different cause and increase the number of replays in another category. However, this effect is small. In this experiment, we start from the naive design and gradually incorporate all the optimization techniques. Since there is overlap between the effects of different techniques, the result of applying multiple techniques is not additive, and the techniques introduced later on tend to demonstrate diminishing returns. For clarity, we only show the average of the number of replays per 10,000 committed instructions for integer applications and for floating-point applications.

The first interesting thing to note is that when the write buffer is introduced to the system, replay frequency is significantly reduced. For example, in integer code, the average replays per

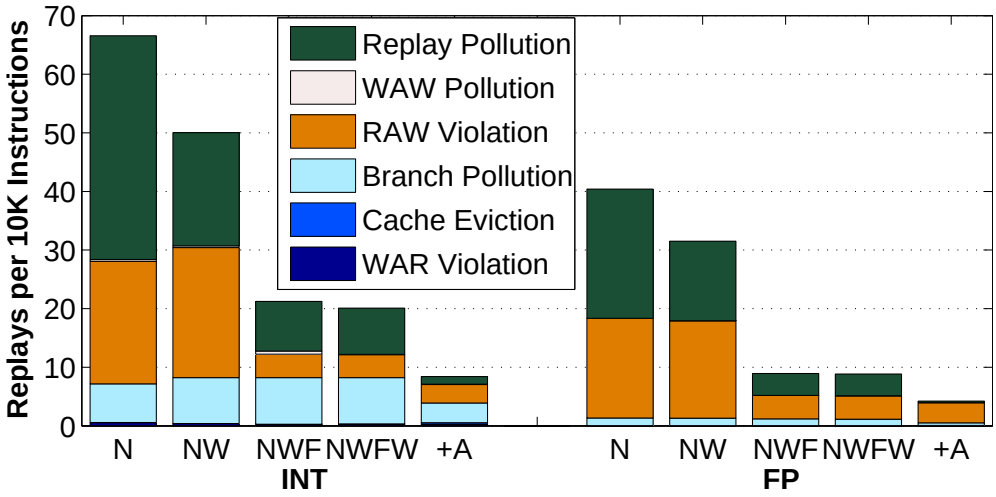


Figure 3.7: The breakdown of the number of replays into different categories. From left to right, the three bars in each group represent different systems: Naive (N), with write-buffer but without the FDQ (Naive + Write-buffer: N+W); with the FDQ to detect RAW violations (N+W+F) and with FDQ to also detect WAW violations (N+W+F+W), and finally, adding age-based filtering (+A) (on top of N+W+F+W). The stacking order of segments is the same as that shown in the legend. Note that the two bottom segments are too small to discern.

10,000 committed instructions reduce from about 67 to 50 in integer programs and from 40 to 31 in floating-point programs. This may appear puzzling initially since the write buffer is not designed to reduce replay frequency. However, as we can see, the reduction is primarily due to that of replay pollution. Indeed, as we speed up the back-end of the processor, we can detect a replay more quickly and prevent too many stores from leaving future data in the L0 cache. In application *bzip2* for example, without write buffer, every replay squashes 46 stores. With the write buffer, this number reduces drastically to 13.

Next, we see that in a system without FDQ (N+W), the two main *direct* sources of replays are branch (misprediction) pollution and RAW violation. Of the two, RAW is the larger source by a small margin in integer applications and by a large margin in floating-point applications. As replays leave the L0 cache in a state with many "future" data, they are likely to trigger replays again. Such secondary replays account for an average of 40% of all replays. In both groups of applications, FDQ is capable of preventing a large majority (75% to 80%) of RAW violation-triggered replays. As a result, the number of secondary replays also reduces significantly (by an average of 55% and 73% in integer or floating-point applications respectively). We can also see

that when FDQ is also used to prevent WAW violations, the number of WAW pollution-triggered replays indeed goes down, though the overall impact is quite small.

Finally, we can see that although in floating-point applications, FDQ can cut down the replays by about 75% to a small 7 per 10,000 instructions, in integer applications, there are still about 20 replays left. These replays are mainly due to branch pollution, directly or indirectly. With the age-based filtering, we are able to filter out a large portion of pollution due to replay and branch misprediction recovery in both groups of applications: 95% (FP) and 84% (INT). The reduction in branch pollution is smaller, but still significant: 60% in both groups of applications.

3.3.3 Putting it Together

We now compare the performance of several complete systems which differ only in the memory dependence enforcement logic. We take the naive design, add an 8-entry write buffer and a 16-entry FDQ. We call this design *Improved*. We compare *Naive* and *Improved* to the baseline conventional system, which uses the conventional disambiguation logic with a 64-entry LQ and a 48-entry SQ, and to an idealized system where we set the LQ and SQ size equal to that of the ROB. In Figure 3.8, these results are shown as minimum, average, and maximum for the integer and floating-point application groups as before. (The per-application detailed results are listed in Table 3.2.) We see that there are applications that perform dramatically worse in the naive design than in the baseline but there are others that achieve significant improvements as well. On average, the naive design performs only slightly worse than the baseline (5.3% and 2.7% slower for integer and floating-point applications, respectively). We point out that with a combined LSQ capacity of 112 entries, even the baseline is optimistically configured. (A smaller but more realistic 32-entry SQ with 48-entry LQ would slow down the system by an average of 11% when executing floating-point applications). Undoubtedly, the naive design is much more complexity-effective.

The *Improved* design is significantly better than *Naive* and comes very close to the ideal configuration. Individual applications' slowdown relative to the ideal configuration can be as high as 15.6%. However, except for 3 applications, all others are within 10% of *Ideal*. In fact,

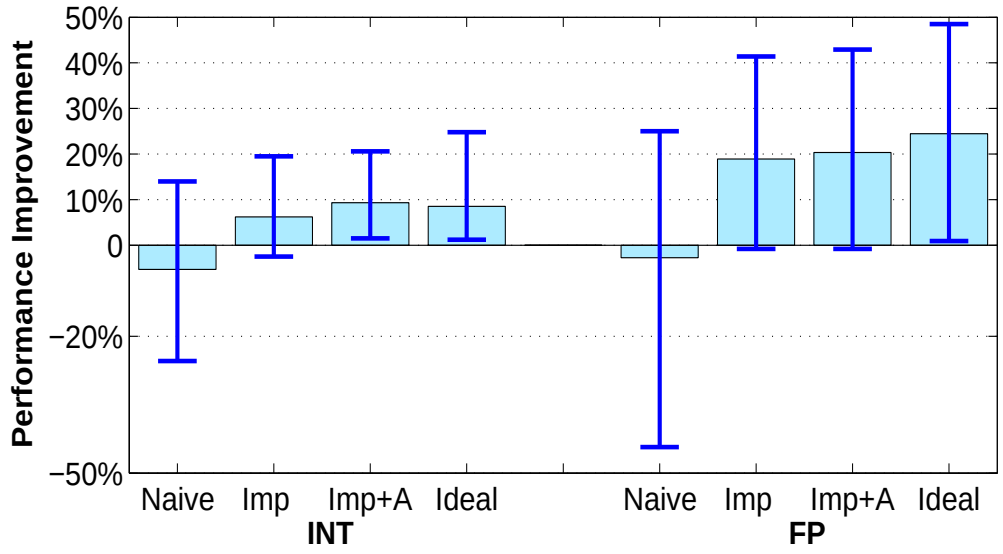


Figure 3.8: Performance improvement of *Naive* and *Improved* (Imp) design over the baseline conventional system. Adding age-based filtering on top of *Improved* (Imp+A) and an ideal conventional system are also shown.

Improved actually outperforms the ideal configuration in seven applications, most notably *gcc* and *equake*. This is not unreasonable: Our design turns the transistors used in building LSQ or predictor tables into a small L0 cache can cache more data and provide the data to the execution core faster. If we apply additional techniques, such as age-based filtering to reduce the replay frequency, a few more applications would run faster in an SMDE design than in *Ideal*. Indeed, adding age-based filtering on *Improved*, we have a system that outperform an ideal LSQ-based system on average (Figure 3.8).

Overall, comparing *Improved* with *Ideal*, the average slowdown is only 2.0% for integer applications and 4.3% for the floating-point applications. Although practicality prevents a pair-wise comparison study with the numerous LSQ-optimization designs, we note that these results (both average and worst-case) come very close to those reported previously, for example, in [SMR05].

Finally, to understand the effect of the age-based filtering (Section 3.1.2.3), we add it to *Improved* and show the effect in Figure 3.8. We can see the notable effect of age-based L0 cache filtering, especially on integer application. Indeed, with this filtering, the average performance improvement over baseline is actually higher than that with ideal LSQ.

	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
1	-14.6	-2.8	-24.0	-25.4	14.0	-18.1	-1.7	-3.4	6.1	8.0	5.7	-7.8
2	-2.5	2.0	0.9	9.4	10.7	0.5	2.9	1.8	15.4	4.3	19.5	9.6
3	1.5	7.0	10.4	11.1	10.4	10.4	3.8	5.1	16.6	4.8	20.6	9.9
4	11.3	10.4	6.2	11.8	1.9	1.2	3.6	1.7	16.2	4.1	24.8	9.0

(a) Integer applications.

amm	app	apsi	art	eqk	fac	fma	gal	luc	mesa	mgr	six	swim	wup
-0.7	25.0	-44.3	19.9	-25.8	-0.1	4.4	4.7	-7.1	-15.0	23.4	-23.3	20.2	-19.8
15.1	39.2	1.0	17.6	6.6	41.4	6.8	5.6	-0.8	20.2	41.1	4.5	30.3	35.9
15.6	42.9	1.9	18.2	19.1	41.4	8.0	5.6	-0.8	20.2	41.3	5.6	38.2	27.3
19.3	48.5	19.6	31.7	4.8	41.1	0.9	7.0	1.5	32.2	46.2	9.7	42.3	37.8

(b) Floating-point applications.

Table 3.2: Performance improvement (in %) of Naive (1), Improved (2), Improved with age-based filtering of L0 cache (3), and Ideal (4) design over the baseline conventional system.

3.3.4 Scalability

Perhaps more interesting than the actual performance is the scalability trend. We performed a limited experiment scaling the system to a 1024-entry ROB. We scale the size of the register files and the issue queues proportionally but keep the disambiguation logic exactly the same as in Figure 3.8 for the baseline conventional system and the SMDE designs (Naive and Improved). In other words, the size of the L0 cache, the write buffer, and the FDQ remains the same. The LQ and SQ in the ideal conventional system, however, are scaled together with the ROB. We again normalize to the baseline conventional configuration. We also "scale up" the branch predictor by randomly correcting 75% of mispredictions in the simulator. We show the result in Figure 3.9.

First, we see that the naive design gains ground against the baseline for floating-point applications. This is expected. Without any limit on the number of in-flight instructions, the naive approach can easily use the enlarged ROB to exploit long-range ILP. This benefit adds to that of better branch prediction in the scaled-up system. For the baseline conventional system however, the large ROB has no effect due to the limit of the LSQ. The benefit almost entirely comes from better branch prediction in the scaled-up system. Thus, naive SMDE's performance improves

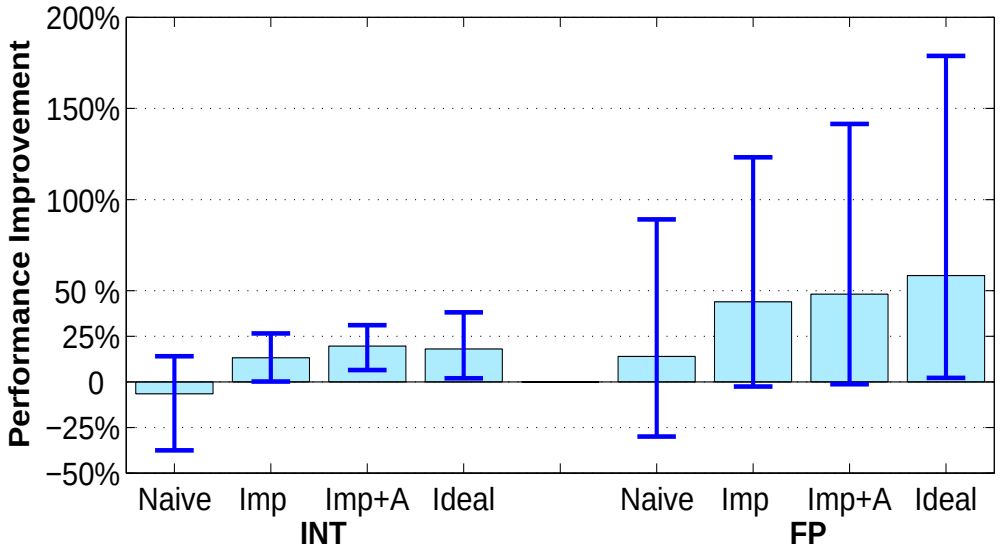


Figure 3.9: Scalability test for SMDE configurations (*Naive* and *Improved*) and the ideal conventional configuration.

relatively to the baseline.

Second, the improved SMDE design continues to track the ideal conventional configuration well. This is a key benefit in addition to the simplicity of the SMDE design: Once the design is in place, hardly any resource scaling is needed.

Finally, age-based filtering shows more benefit in the scaled up configuration. However, the difference is not dramatic. If the underlying microarchitecture changes and the cost of replay is higher, then it may become a more useful mitigation technique.

3.3.5 Other Findings

Finally, during the design and evaluation of SMDE, we performed many quantitative experiments. We summarize some of the findings here.

Replay suppression

When a replay is triggered, there are times when no in-flight instruction is a dependent of the triggering load. Depending on the exact configuration, the percentage of such cases vary but remains non-negligible. For example, in the *Improved* configuration, on average, there are

about 20% of replay-triggering loads in floating-point applications that do not have any in-flight dependents. In certain applications this rate can be as high as 100%. Therefore, in an SMDE design, we do not need to trigger an actual replay of subsequent instructions. However, the end performance implication is rather limited in this configuration: about 1%.

Flush policy

When handling a replay, we can perform some L0 cache cleaning. We emphasize again that this is purely an optimization to reduce the chance of more replay and does not affect correctness in any way. Among the possible policies, we compared no flushing (F0), flush the line that triggered the replay (F1), selective flushing (FS) by walking through the MOSQ to flush all the lines belonging to the to-be-squashed stores, and finally, flush the entire L0 cache (FA). It is not hard to see that F0 and F1 are fairly straightforward to implement, while FA and especially FS can be complex in circuit and expensive in energy at runtime, though they definitely reduce the replay rate.

In Table 3.3, we show the percentage of loads triggering replay in the *Improved* design if we use different flushing policies. We can see that as we flush more data from the L0, naturally, the replay frequency reduces. However, L0 cache miss rate increases as we flush more data. An L0 cache miss not only delays the execution of dependents of the load, it also disrupts the execution of instructions scheduled in the vicinity of the load due to scheduling replay. As we can see, all other policies give lower average performance when compared to F1. (The effect of FS is very close to that of F1.) Fortunately, unlike FS, F1 is very easy to implement.

Understanding the write buffer

The write buffer we use serve three purposes. First, it buffers write misses. When a store instruction is being committed but misses in the cache, depending on the design of the memory subsystem, the processor may not be able to remove the instruction from the ROB and continue to commit other instructions. In Alpha [Com00], for example, the instruction is removed from the ROB, whereas the data is kept in the SQ in order to properly forward to load instructions. In SMDE, without a SQ, we can either stall commit when there is a write miss, or if there is

	INT				FP			
	Max	Avg	Min	Perf.	Max	Avg	Min	Perf.
F0	218	54	1	-7.43%	80	18	0	-5.51%
F1	45	20	1	0%	27	9	0	0%
FS	27	12	1	0.99%	24	5	0	-1.49%
FA	28	9	1	-2.35%	23	4	0	-3.25%

Table 3.3: Number of loads triggering replay per 10,000 instructions in *Improved* under different L0 cache flushing policies (shown in maximum, average, and minimum of the entire group of applications), and their average performance impact (Perf.) in percentage compared to the single-line flush policy (F1).

the write buffer, we can keep the data in the write buffer and retire the instruction. Note that at the execution time for the store instruction, processors typically prefetch the cache line. Thus, write miss at commit time is quite rare. A second functionality of the write buffer is to improve the utilization of cache port. When a store being committed compete with a load for the cache ports, the write buffer allows both to proceed by time-shifting the store's usage of cache port to a later cycle. Finally, the third functionality is to allow the BEEP pointer to travel ahead with respect to the retirement pointer of the ROB so as to hide the latency of back-end execution of load instructions.

To understand the effect of all three functions, we incrementally add them to the *Naive* system and measure their effect in terms on performance improvement on top of *Naive* (without a write buffer). Table 3.4 shows this experiment. As we can see, hiding load latency in the back-end execution is the most important contribution of the write buffer, but the other functionalities also contribute non-trivial amount of improvement.

	INT	FP
WB holding write misses only	2.15%	5.75%
Also time-shift cache ports	4.47%	7.69%
All three functions	15.95%	16.08%

Table 3.4: Effect of different functionalities of the write buffer.

Membership test

As we saw earlier, the effect of even a small write buffer is quite substantial. However, the forwarding capability from a small write buffer is not necessarily critical. In fact, we found that only 3-4% of loads forward from this write buffer on average. Therefore it is conceivable to have a non-forwarding write buffer and provide a simple and quick membership test to detect any address overlapping and stall the load until the conflicting store is drained out of the buffer. We performed a limited study using an ideal membership test to study the performance impact of having to delay conflicting loads. We found that the average performance degradation is about 0.5% for both groups of applications and the maximum slowdown is only about 2% in any single application.

3.4 Conclusions

In this chapter, we have presented a slackened memory dependence enforcement (SMDE) approach aimed at simplifying one of the most complex and non-scalable functional blocks in modern high-performance out-of-order processors. In an SMDE design, memory instructions execute twice, first in a front-end execution where they execute only according to register dependences and access an L0 cache to perform an opportunistic communication. They then access memory a second time, in program order, accessing the non-speculative L1 cache to detect mis-speculations and recover from it.

A primary advantage of SMDE is its simplicity. It is also very effective. We have shown that even a rudimentary implementation rivals a conventional disambiguation logic with optimistically sized load queue and store queue. When two optional optimization techniques are employed, the improved design offers performance close to that of a conventional system with ideal load-store queue. Another advantage of the design is that when scaling up the in-flight instruction capacity, almost no change is needed and yet the performance improves significantly.

The SMDE approach is distinct in several ways from conventional design and recent proposals. First, it is significantly more decoupled between the forwarding and monitoring/verification component. This allows for modular design, verification, and optimization. Second, the for-

warding component, working at core execution speed has minimal external monitoring or interference. There is no need to communicate with front-end predictors either. There is even no need for address translation. Third, the verification component is straightforward and handles all cases: mis-speculation or coherence/consistency constraints. By providing a separate cache, we *avoid* the bandwidth consumption problem of re-execution.

Next, we will widen the scope of explicit decoupling by applying these principles to improve traditional instruction level parallelism (ILP) lookahead.

Chapter 4

Explicitly Decoupled ILP Lookahead

While lookahead techniques have the potential to uncover significant amount of ILP, conventional microarchitectures impose practical limitations on its effectiveness due to their monolithic implementation. Correctness requirement limits the design freedom to explore probabilistic mechanisms and makes conventional lookahead resource-intensive: registers and various queue entries need to be reserved for every in-flight instruction, making deep lookahead very expensive to support. Moreover, the design complexity is also high as introduction of any speculation necessitates fastidious planning of contingencies.

In contrast to this "integrated" lookahead design, in an explicitly decoupled architecture, a decoupled agent is to provide the lookahead effects. Furthermore, we also want to minimize the mutual dependence between the lookahead agent on the normal processing agent (the *optimistic* and the *correctness* core, respectively in our design shown in Figure 4.1). This decision has implications on how we maintain autonomy of the cores and manage the deviance between them.

Autonomy

A key point of our design is that the optimistic core can be *specialized* to perform lookahead more effectively by leveraging the lack of correctness constraints. To maintain autonomy of the lookahead with respect to normal processing, we use an independent thread of control – the

optimistic thread. Having its own thread of control in the optimistic core also allows us to freely exploit speculative, optimistic software analyses or transformations.

We could simply use another copy of original program binary as the optimistic thread. This is straightforward but suboptimal. A skeletal version of the program that contains only instructions relevant to future control flow and data accesses is enough. There is no need to include computation that is only necessary for producing the right program output and non-essential for lookahead. We can rely on software analysis to generate such a "skeleton" in a probabilistic fashion. In this thesis, our software analysis is done on the program's binary. Performing the tasks on binaries has the significant benefit of hiding all the implementation details beneath the contractual interface between the hardware and the programs, and maintaining semantic binary compatibility: In each *incarnation* of an explicitly decoupled architecture, we can customize the instruction set of the optimistic core without worrying about future compatibility obligations.

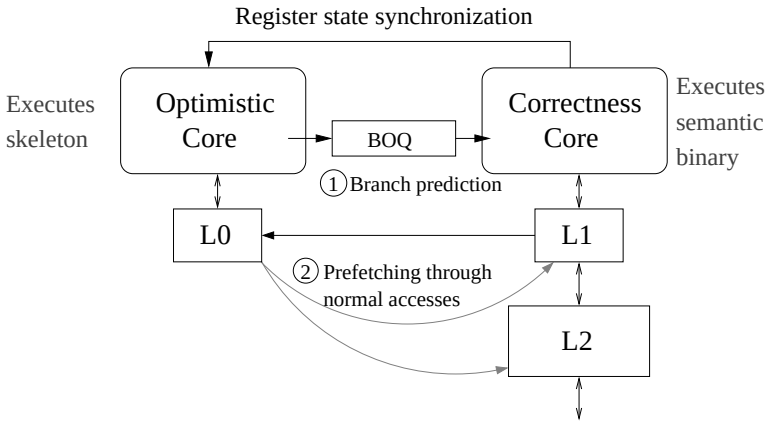


Figure 4.1: The optimistic core, the correctness core, and the organization of the memory hierarchy. The optimistic core ① explicitly sends branch predictions to the correctness core via the branch outcome queue (BOQ) and ② naturally performs prefetching with its own memory accesses.

Managing deviance

The removal of correctness constraints in the performance domain provides the freedom to explore cost-effective performance-boosting mechanisms and avoid excessive conservativeness. However, it would inevitably lead to deviation from the desired result. For example, approximations in the skeleton generation, a logic simplification in the architecture design, or device

glitches due to insufficient margin can all cause the architectural state in the performance domain to deviate from the desired state. If the design heavily depends on vast amounts of predictions and on the preciseness of the predicted information from the performance domain, such deviations are likely to result in costly remedies in the correctness domain and ultimately limit our freedom in exploring unconventional and optimistic techniques.

To build in an inherent tolerance for such deviations, we do not rely on the optimistic core to provide value predictions and only draw branch direction predictions from it. This is done using a FIFO structure *Branch Outcome Queue* (BOQ) as shown in Figure 4.1. This also allows us to detect the control flow divergence between the two threads. When this happens, the correlation between the execution of the two threads is reduced and at some point, the state of the optimistic core needs to be reinitialized to maintain its relevance in lookahead. We call this a *recovery*. In this thesis, for simplicity, a recovery is triggered whenever a branch misprediction is detected in the correctness core and a recovery involves copying architectural register state from the correctness core to the optimistic core. We note that while we are actively exploring alternatives (of recovering on every misprediction), we have not found a design with superior performance.

Even with the recovery mechanism, memory writes in the performance domain are still fundamentally speculative and need to be contained within its local cache hierarchy. We use one private cache (L0, for notional convenience). By simply sharing the rest of the memory hierarchy between the two cores, we can tap into the rest of the architectural state in a complexity-effective manner. L0 never writes back anything to the rest of the hierarchy.

Finally, there are times when the type of boosting a particular implementation performs is not yielding sufficient benefit to offset the overheads such as that from the recoveries. In that case, the better alternative is to halt the optimistic thread and perform a recovery later when the execution moves to a code region where boosting will be effective again. We leave this as future work.

4.1 Background

Uniprocessor microarchitectures have long used look-ahead to exploit parallelism implicit in sequential code. However, even for high-end microprocessors, the range of look-ahead is rather limited. Every in-flight instruction consumes some microarchitectural resources and practical designs can only buffer on the orders of 100 instructions. This short "leash" often stalls look-ahead unnecessarily, due to reasons unrelated to look-ahead itself. Perhaps the most apparent case is when a long-latency instruction (*e.g.*, a load that misses all on-chip caches) can not retire, blocking all subsequent instructions from retiring and releasing their resources. Eventually, the back pressure stalls the look-ahead. This case alone has elicited many different mitigation techniques. For instance, instructions dependent on a long-latency instruction can be relegated to some secondary buffer that are less resource-intensive to allow continued progress in the main pipeline [CCE+09; SRA+04]. In another approach, the processor state can be check-pointed and the otherwise stalling cycles can be used in a speculative mode that warms up the cache [DM97; MSWP03; CCYT05; CCE+09]. One can also predict the value of the load and continue execution speculatively [KKCM05; CST+04]. Even if the prediction is wrong, the speculative execution achieves some degree of cache warming.

If there are additional cores or hardware thread contexts, the look-ahead effort can be carried out in parallel as the program executes, rather than being triggered when the processor is stalled. Look-ahead can be targeted to specific instructions, such as so-called delinquent loads [CWT+01], or can become a continuous process that intends to mitigate all misses and mispredictions. In the former case, the actions are guided by backward slices leading to the target instructions and the backward slices are spawned as individual short threads, often called helper threads [FTEJ98; CSK+99; ZS01; APD01; Luk01; CWT+01; RS01; MPB01; LWW+02]. In the latter case, a dedicated thread runs ahead of the main program thread. This run-ahead thread can be based on the original program [PSR00; BNS+03b; Zho05; MMR07; GT07; LWW+02] or based on a reduced version that only serves to prefetch and to help branch predictions [GH08].

While the two approaches are similar in principle, there are a number of practical advantages of using a single, continuous thread of instructions for look-ahead. First, as shown in Figure 4.1,

the optimistic core or the look-ahead thread is an independent thread. Its execution and control is largely decoupled from the main thread. (For notational convenience, we refer to this type of design as decoupled look-ahead.) In contrast, embodying the look-ahead activities into a large number of short helper threads inevitably requires micro-management from the main thread and entails extra implementation complexities. For instance, using extra instructions to spawn helper thread requires modification to program binary and adds unnecessary overhead when the run-time system decides not to perform look-ahead.

Second, passing branch hints (more in Section 4.3), is straightforward in decoupled look-ahead. Because the look-ahead code mirrors that of the main thread, there is a one-to-one mapping between the branch streams. Using individual helper threads to selectively precompute certain branch outcomes requires extra support to match the producer and the consumer of individual branch hints.

Third, prefetching too early can be counter-productive and needs to be avoided. This becomes an issue when helper threads can also spawn other helper threads to lower the overhead on the main thread [CWT$^+$01]. In decoupled look-ahead, since the look-ahead thread pipes its branch outcome through a FIFO to serve as hints to the main thread, it naturally serves as a throttling mechanism, stalling the look-ahead thread before it runs too far ahead or we can easily delay the triggering of prefetches.

Finally, as program gets more complex and uses more ready-made code modules, "problematic" instructions will be more spread out, calling for more helper threads. The individual helper threads quickly add up, making the execution overhead comparable to a whole-program based decoupled look-ahead thread. As an illustration, Table 4.1 summarizes the statistics about those instructions which are most accountable for last-level cache misses and branch mispredictions. For instance, the top static instructions that generated 90% of the misses and mispredictions in a generic baseline processor accounted for 8.7% of total dynamic instruction count. Assuming on average each such instruction instance is being targeted by an extremely brief 10-instruction long helper thread, the total dynamic instruction count for all the helper threads becomes comparable to the program size. If we target more problematic instance (*e.g.*, 95%), the cost gets even higher.

	Memory references				Branches			
	90%		95%		90%		95%	
	DI	SI	DI	SI	DI	SI	DI	SI
bzip2	1.86	17	3.15	27	3.9	52	4.49	64
crafty	0.73	23	1.04	38	5.33	235	6.14	309
eon	2.28	50	3.34	159	2.02	19	2.31	23
gap	1.35	15	1.44	23	2.02	77	2.64	130
gcc	8.49	153	8.84	320	8.08	1103	8.41	1700
gzip	0.1	6	0.1	6	8.41	40	8.66	52
mcf	13.1	13	14.7	16	9.99	14	10.2	18
parser	1.31	41	1.59	57	6.81	130	7.3	183
pbmk	1.87	35	2.11	52	2.88	92	3.21	127
twolf	2.69	23	3.28	28	5.75	41	6.48	56
vortex	1.96	42	2	67	1.24	114	1.97	167
vpr	7.47	16	11.6	22	4.8	6	4.88	7
Avg	3.60%	36	4.44%	68	5.10%	160	5.56%	236

Table 4.1: Summary of top instructions accountable for 90% and 95% of all last-level cache misses and branch mispredictions. Stats collected on entire run of ref input. DI is the total dynamic instances (measured as a percentage of total program dynamic instruction count). SI is total number of static instructions.

Key differences from the related work

It is worth highlighting here the key differences between our proposal and previous proposals. Note that the differences are often the result of difference in goal.

Decoupling correctness and performance issues is not a new concept. We want to make a case for a more explicit, up-front approach to decoupling, which makes performance optimization and correctness guarantee more independent than prior art. This is reflected in

1. The division of labor in the two cores: The optimistic core is only attempting to facilitate high performance by passing hints and other meta data. In the common case, it only provides good hints, whereas the leading cores in [Aus99; SPR00; Zho05] will produce complete and correct results.

2. The minimal mutual dependence between them: Neither does the trailing core require a large amount of accurate information from the leading core (such as architectural state

to jump start future execution [ZS02]), nor does the leading core heavily depend on the trailing core to perform its task [Smi84].

4.2 Basic Support

The potential of explicitly-decoupled architecture lies in the opportunities it opens up for *efficient* and *effective* optimizations. The required support to allow the optimistic core to perform self-sustained lookahead is rather basic and limited.

4.2.1 Software support

A key requirement for the envisioned system to work effectively is that the optimistic core has to *sustain* a performance advantage over the correctness core so as to allow *deep* lookahead. A key opportunity is that the skeleton only needs to perform proper data accessing, which is only part of the program, and may be able to skip the remainder. This is not a new concept. Indeed, the classic access/execute decoupled architecture [Smi84] exploits the same principle to allow the access stream to stay ahead. However, the challenge is that our optimistic core is significantly more independent and has to do enough work to ensure a highly accurate control flow. As it turns out, using conventional analysis on the binary, we can not successfully remove a sufficient number of instructions: about 10-12% dynamic instructions (most of which prefetches) can be removed from the program binary without affecting the program control flow. This is not sufficient to sustain a speed advantage for the optimistic thread. While extremely biased branches (identified through profiling) can be removed or turned into unconditional branches reducing the need for some branch condition computation, solely relying on this is also insufficient.

A simple but important observation is that the optimistic thread has access to the architectural memory hierarchy in the correctness domain and therefore can obtain the data from memory when the producer store is sufficiently upstream in the instruction sequence that at the time of load – it would have been executed by the correctness core. We do not need to include the store and its backward slice in the skeleton (illustrated in Figure 4.2). Note that this is also exploited earlier in [ZS02].

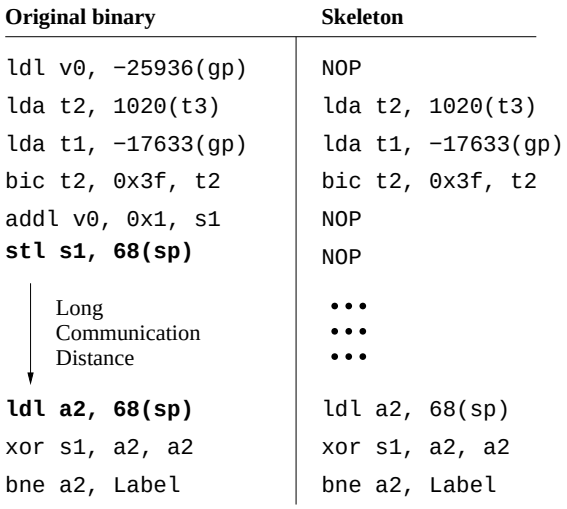


Figure 4.2: Illustration of avoiding unnecessary computation in the skeleton. When a store has a long communication distance with its consumer load the computation chain leading to the store is omitted in the skeleton.

4.2.2 Architectural support

The architectural support required to enable our explicitly decoupled architecture is also limited. For the most part, both cores operate as self-sufficient, stand-alone entities. The only relatively significant coupling between the two cores is that the correctness core's memory hierarchy also serves as the lower levels of the memory hierarchy for the optimistic core. Note that, the accesses from the optimistic core to L1 is infrequent as it only happens when the L0 misses. Hence, extra traffic due to servicing L0 misses is insignificant. Indeed, as we will show quantitatively later, the increase in L1 traffic is more than offset by the decrease of L1 accesses from the correctness core because of better branch prediction.

Recovery mechanism

Another support needed is the recovery mechanism. A branch outcome queue (BOQ) is used to pass on the branch direction information from the optimistic core to the correctness core. When such a prediction is detected in the correctness core as incorrect, a recovery is triggered. The correctness core drains the pipeline and passes the architectural register state back to the optimistic core. Since the L0 cache is corrupted because of wrong-path execution, some cleansing

may be helpful. For simplicity, we reset the entire L0 cache upon a recovery.

Correctness core fetch stage

For design simplicity, the fetch stage of the correctness core is frozen when the BOQ is empty. This ensures the "alignment" of the branches: the next branch outcome to be deposited by the optimistic core in the BOQ is always intended for the next branch encountered by the correctness core. If, to avoid stalling, the correctness core consults a different branch predictor when the BOQ is empty, then extra circuitry is required to keep track of how many predictions are made using the alternative source, to determine whether the two cores are still on the same control flow, and to decide if the optimistic core has already taken over so as to switch back to the BOQ as the prediction source. In short, the circuit support would be non-trivial. And as we will show later, when the entire system is properly optimized, stalls in the correctness core can be kept low. Nevertheless, alternatives of this policy are interesting to study.

4.3 Optimizations

By separating out correctness concerns, explicitly decoupled architecture allows designers to make different trade-offs and devise more effective performance optimization strategies. A primary implication of decoupling is that not all mis-speculations need to be corrected or even detected in the performance domain. In a conventional design that tightly couples correctness and performance, the complexity of such detection and recovery logic may significantly affect cost-effectiveness of the implementation, reduce the appeal of an otherwise sound idea, and can even defeat the purpose of speculation. Designs can use new, probabilistic mechanisms to explore optimization opportunities in a more cost-effective way and avoid the complex algorithms and circuitry that place stringent requirements on implementation. We discuss a few opportunities that we have explored.

4.3.1 Skeleton Construction

Recall that the skeleton does not need to contain long-distance stores (Section 4.2) and their computation chain. However, the communication relationship between loads and stores is not always clear, especially when dealing only with program binaries. Fortunately, our binary parsing only needs to approach the analysis in a probabilistic fashion, and we can use profiling to easily obtain a statistical picture of load-store communication patterns. The process is as follows and we use a binary parser based on `alto` [MDWB01] to perform the analysis and transformations.

Profiling

We first perform a profiling step to obtain certain information parsing the binary alone can not obtain. First of all, we can obtain the destinations of indirect jump instructions. Again, we do not need to capture all possible destinations, thanks to the lack of correctness requirement for the optimistic core. With this information, we can make the control flow graph more complete.

Secondly, we collect statistics about short-distance load-store communications. Using a training input, we obtain the list of stores with *short instances*. A short instance is a dynamic store instance whose consumer load is less than d_{th} instructions downstream. We set d_{th} to 5000 in this work. We found that the profile results are not sensitive to d_{th}. For every store with short instances, we tally the total number of dynamic instances as well as short instances. For the latter, the statistics are further subdivided based on the identity of their consumer loads (illustrated in Figure 4.3 and elaborated later). This is needed in later analysis because whether a short instance matters depends on if the consumer load is part of the skeleton.

Finally, the profiling run also identifies load instructions that are likely to miss in the (L2) cache and branches with strong biases. The statistical miss frequencies are recorded for later analysis. Branch bias factored with cost (additional instructions added) is used to label some branches as biased. In general, these branches have a bias greater than 99.9%.

After converting conditional branches with strong biases to unconditional branches, among the conditional branches which still stay in the skeleton are the highly-predictable branches. Executing these branches is useless for the forward progress of the skeleton after branch predictors

Stores	Total Dyn. Instances.	L_1	L_2	L_3	L_4
S_1	10,000		100		
S_2	200,000	1,000	50		
S_3	5,000			40	60
⋮					

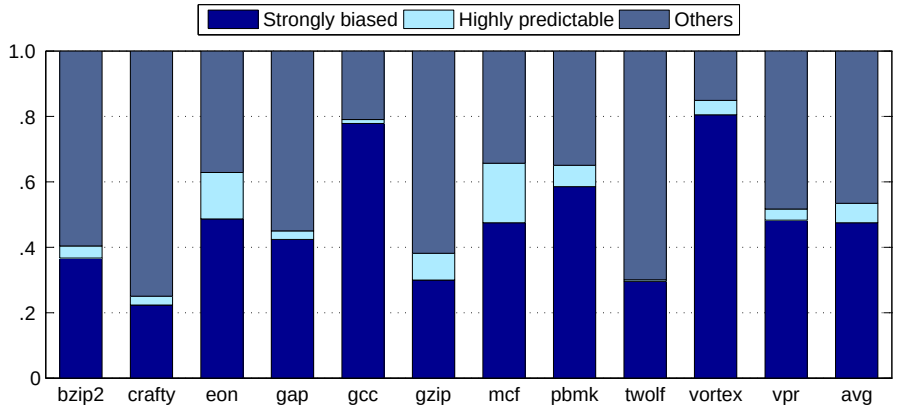
Figure 4.3: Illustration of statistics on short store instances. For example, the statistics show that store S_2 executed 200,000 times with 1050 short instances, 1000 times communicating with load L_1 and 50 times with L_2.

are properly trained. If not executing these branches significantly shrinks the skeleton size, we can play simple software and hardware tricks to use alternate ways to train branch predictors. Figure 4.4 shows the breakdown of conditional branches. As it turns out, highly-predictable branches constitute only a small portion of all the branches and removing them achieves further skeleton reduction on average of only 0.46% for integer applications and 1.33% for floating-point applications. Since the potential is not significant enough, wo do not attempt to exploit the highly-predictable branches.

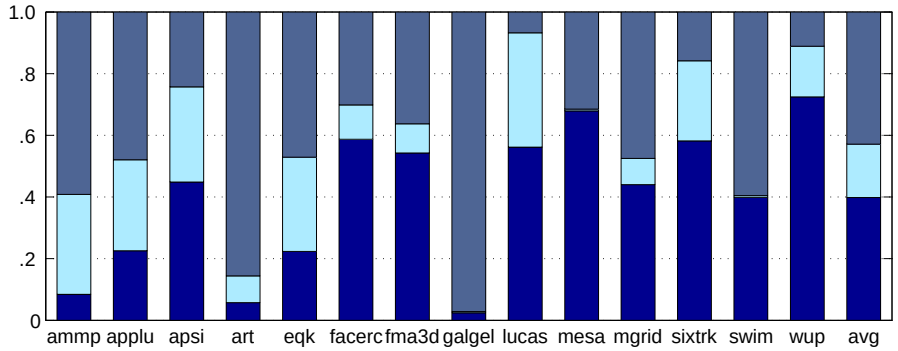
Binary analysis

With this profile information, we then proceed to build a program skeleton. The goal of the skeleton is to closely track the original program's control flow and be able to pass on branch prediction information and issue timely prefetches. Thus, the first thing we do is to mark branch instructions as selected in the skeleton (illustrated in Figure 4.6-(b)). Next, traversing the data-flow graph backward, we select all the instructions on the backward slice of the branch instructions (illustrated in Figure 4.6-(c)). Branches considered extremely biased are turned into NOPs or unconditional branches and therefore they do not have any backward slice. Following this, we need to deal with memory dependences and include producer stores that feed into the loads included in the current skeleton.

Recall that a memory-based dependence can be "long-distance" in that the load comes long after the finish of the store and there is a high probability that the trailing correctness core has also finished the store and can provide the data. In that case, including the store in the



(a) Integer applications.



(b) Floating-point applications.

Figure 4.4: Normalized breakdown of conditional branches for SPEC2000 applications.

skeleton would unnecessarily slow down the optimistic thread and increase waste. On the other hand, if one instance of memory dependence is short-distance, not including the store in the skeleton can lead to the load consuming a wrong value and may eventually lead to a costly recovery. Therefore, our goal is to minimize the number of stores included in the skeleton while limiting the number of potential recoveries. The aforementioned profile information helps us to determine which store is more profitable to keep. We use the following algorithm (illustrated in Figure 4.5) to determine the list of candidate stores to include in the skeleton.

- Step 1. We scan the current skeleton and compile a list of loads already selected ($List_{LD}$).
- Step 2. For every load in the list, we check the profiling statistics (Figure 4.3) and add its producer stores to the a list of candidate stores to be included in the skeleton $List_{ST}$. Note that based on the short-distance matrix, only producer store that ever had short-distance

```
skeleton = {conditional branches};
for inst in {conditional branches}:
   skeleton += backwardSlicing(inst);

do {
   list_ld = listOfLoadsIn(skeleton);
   list_st = storesDrivingLoads(list_ld);
   list_st = prune(list_st);
   for inst in list_st:
      skeleton += backwardSlicing(inst);
} while newLoadsAddedTo(skeleton)
```

Figure 4.5: Pseudocode of the algorithm used for skeleton construction.

communication with a load of interest are added to $List_{ST}$. For each store added, we also keep track of the total number of dynamic instance (I_t) and the number of short instances (I_s) communicating with a load in $List_{LD}$. For example, if L_1 is in the skeleton, S_2 will be added to the candidate list together with the contribution of 1000 short instances. Later on, if we encounter L_2 in the skeleton, in addition to adding S_1 to the candidate list $List_{ST}$, we also adjust S_2's total short instance to 1050. This way, I_s only reflects those short instances that matter to the skeleton.

- Step 3. After a complete pass, we trim $List_{ST}$ by removing unprofitable stores. The benefit of keeping a store in this list is to guarantee correct dependence for the short instances. This can prevent the optimistic thread from consuming an incorrect result and eventually triggering a costly recovery. The cost of keeping the store, on the other hand, is adding extra instructions to the optimistic thread which slow it down and consume more energy. The cost/benefit ratio can be crudely approximated by the ratio I_t/I_s. Therefore, we sort the candidate list by descending order of the ratio I_t/I_s and start to trim stores from the top of the list. To limit the potential number of recoveries, we stop trimming before the accumulated I_s is over a certain threshold. In this thesis, this threshold is set to one 10,000th of the total number of dynamic instructions in the profiling run.

- Step 4. We then include stores in the trimmed $List_{ST}$ in the skeleton. This also involves

data-flow analysis to add their backward slice as well. As a result, more loads will be added and $List_{LD}$ will be expanded. We thus go back to Step 2 and iterate through the steps. We can iterate until the list converges or until we hit the threshold for the number of iterations. Our experience shows that typically 2-3 iterations are enough to converge. When we finish, we have a *basic* skeleton.

- Step 5. Finally, we insert prefetch instructions for those loads likely to miss in the cache and are not already included in the skeleton. Whether to include a particular load is also determined by its cost-benefit ratio. The benefit (of adding a prefetch) is approximated as the miss penalty multiplied by the miss probability. The cost is approximated by the number of instructions added to compute the address. If the ratio is lower than a threshold (empirically set to 3), the prefetch is inserted.

Eliminating dynamically dead instructions

An instruction is dynamically dead if the value produced by it is never used in the program (referred to as dead value). Past research shows that the majority of the dynamically dead instructions arises from a small set of static instructions that produces dead values most of the time [BS02]. Profiling is used to identity static instructions which frequently produces dead values ($> 99\%$ in our setup). These instructions are than removed from the skeleton. Except a few applications, our analysis shows that only few dead instructions are a part of the skeleton binary and on average it does not show a significant further reduction in size after removing dead instructions. Table 4.2 show detailed results.

bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
0.32	2.70	1.33	0.38	0.07	0.02	0.00	2.44	1.10	4.43	0.83	1.33

(a) Integer applications.

amm	app	apsi	art	eqk	fac	fma	gal	luc	mesa	mgr	six	swim	wup
0.00	2.63	0.00	0.00	3.83	1.46	0.08	0.28	0.15	0.00	0.60	0.00	1.00	0.72

(b) Floating-point applications.

Table 4.2: Percentage reduction in skeleton size after removing dead instructions for SPEC2000 applications.

Address	Binary		Address	Binary		Address	Binary	
0x12000ea10	ldt	$f30, 0(t12)	0x12000ea10	ldt	$f30, 0(t12)	0x12000ea10	ldt	$f30, 0(t12)
	cmplt	t11, a2, s6		cmplt	t11, a2, s6		**cmplt**	**t11, a2, s6**
	cpys	$f31,$f31,$f18		cpys	$f31,$f31,$f18		cpys	$f31,$f31,$f18
	bis	zero, t11, a3		bis	zero, t11, a3		**bis**	**zero, t11, a3**
	beq	s6, 0x12000ea68		**beq**	**s6, 0x12000ea68**		**beq**	**s6, 0x12000ea68**
	bis	zero, a0, s6		bis	zero, a0, s6		**bis**	**zero, a0, s6**
	ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)
	ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)
0x12000ea30	xor	a3, a1, s0	0x12000ea30	xor	a3, a1, s0	0x12000ea30	**xor**	**a3, a1, s0**
	bne	s0, 0x12000ea58		**bne**	**s0, 0x12000ea58**		**bne**	**s0, 0x12000ea58**
	ldt	$f24, 0(s6)		ldt	$f24, 0(s6)		**ldt**	**$f24, 0(s6)**
	ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)		ldq_u	zero, 0(sp)
	fble	$f24, 0x12000ea58		**fble**	**$f24, 0x12000ea58**		**fble**	**$f24, 0x12000ea58**
	ldq	s0, 0(t10)		ldq	s0, 0(t10)		ldq	s0, 0(t10)
	s8addq	a3, s0, s0		s8addq	a3, s0, s0		s8addq	a3, s0, s0
	ldt	$f25, 0(s0)		ldt	$f25, 0(s0)		ldt	$f25, 0(s0)
	mult	$f25,$f20,$f25		mult	$f25,$f20,$f25		mult	$f25,$f20,$f25
	addt	$f18,$f25,$f18		addt	$f18,$f25,$f18		addt	$f18,$f25,$f18
0x12000ea58	addl	a3, 0x1, a3	0x12000ea58	addl	a3, 0x1, a3	0x12000ea58	**addl**	**a3, 0x1, a3**
	lda	s6, 16(s6)		lda	s6, 16(s6)		**lda**	**s6, 16(s6)**
	cmplt	a3, a2, s0		cmplt	a3, a2, s0		**cmplt**	**a3, a2, s0**
	bne	s0, 0x12000ea30		**bne**	**s0, 0x12000ea30**		**bne**	**s0, 0x12000ea30**
0x12000ea68	ldt	$f16, −8(t12)	0x12000ea68	ldt	$f16, −8(t12)	0x12000ea68	ldt	$f16, −8(t12)
	addl	s5, 0x1, s5		addl	s5, 0x1, s5		**addl**	**s5, 0x1, s5**
	lda	t10, 8(t10)		lda	t10, 8(t10)		lda	t10, 8(t10)
	cmplt	s5, t1, a3		cmplt	s5, t1, a3		**cmplt**	**s5, t1, a3**
	lda	t12, 64(t12)		lda	t12, 64(t12)		lda	t12, 64(t12)
	addt	$f16,$f18,$f16		addt	$f16,$f18,$f16		addt	$f16,$f18,$f16
	cmpteq	$f30,$f16,$f30		cmpteq	$f30,$f16,$f30		cmpteq	$f30,$f16,$f30
	mult	$f16,$f16,$f14		mult	$f16,$f16,$f14		mult	$f16,$f16,$f14
	stt	$f16, −64(t12)		stt	$f16, −64(t12)		stt	$f16, −64(t12)
	addt	$f21,$f14,$f21		addt	$f21,$f14,$f21		addt	$f21,$f14,$f21
	ftoit	$f30, s6		ftoit	$f30, s6		ftoit	$f30, s6
	cmoveq	s6, 0, t4		cmoveq	s6, 0, t4		cmoveq	s6, 0, t4
	bne	a3, 0x12000ea10		**bne**	**a3, 0x12000ea10**		**bne**	**a3, 0x12000ea10**

(a) (b) (c)

Figure 4.6: A demonstration of initial steps in skeleton construction. Instructions selected in the skeleton are shown in bold. First, only the branch instructions are selected (b). Next, the instructions on the backward slice of the branch instructions are selected (c).

Eliminating useless branches

Note that in terms of what information to pass between the two domains in an explicitly decoupled architecture and how to obtain that information in the performance domain, the design space is vast. The basic skeleton we formed is a code that not only strives to stay on the right path to maintain relevance, but also attempts to execute *every* branch in the original semantic binary. This is a design choice, not a necessity to support deep lookahead. We explore this

option because handling frequent branch misprediction is a necessity that affects all microarchitectural components. If the correctness domain can expect a highly-accurate stream of branch predictions, its microarchitecture can be *fundamentally* simplified. Because of this choice, we found that the skeleton includes branches completely useless for its own execution. These include empty if-then-else structures and sometimes empty loops as shown in Figure 4.7. In these cases, including the branch can be very inefficient, especially in the case of empty loops: when the loop branch is biased and turned into an unconditional branch, the optimistic thread will be "trapped" in the loop until the trailing correctness thread catches up, finishes the same loop, and generates a recovery. Not only will the optimistic thread forfeit any lead upon reaching the empty loop, it also wastes energy from then on until recovery, doing absolutely nothing useful.

In these cases, by not executing the branch, we avoid unnecessary waste in the optimistic core and may even manage to avoid a costly recovery. It is straightforward to identify these branches using the parser. The only issue when skipping them is that of branch "alignment": Because there is a one-to-one correspondence of branches between the optimistic and correctness thread (so as to use a simple FIFO for the BOQ), if the optimistic thread skips a branch, the correctness thread will (mis)interpret the next piece of prediction as that of the skipped branch, thus losing alignment.

We maintain alignment by replacing the branch to be skipped with a special branch instruction in the skeleton. Specifically, we add three types of branches[1]: BDC, BUT, and BUF as discussed in Table 4.3. Even though these dummy branches do not have an outcome of direction, we can still encode the dominant direction found in profiling (*e.g.*, BUF.T to indicate likelihood of taken) as a prediction passed to the correctness core. This is especially sensible for the loop-branch. Of course, a misprediction for branches like BUF.T does not result in recovery, but in our simplified correctness core, it does result in draining of the pipeline (Section 4.3.3). When a prediction is unavailable, we simply stall the correctness core's fetch and wait until the branch is resolved.

[1]Unlike extending the (semantic) ISA, adding these instructions intended only for the performance domain does not present a compatibility issue.

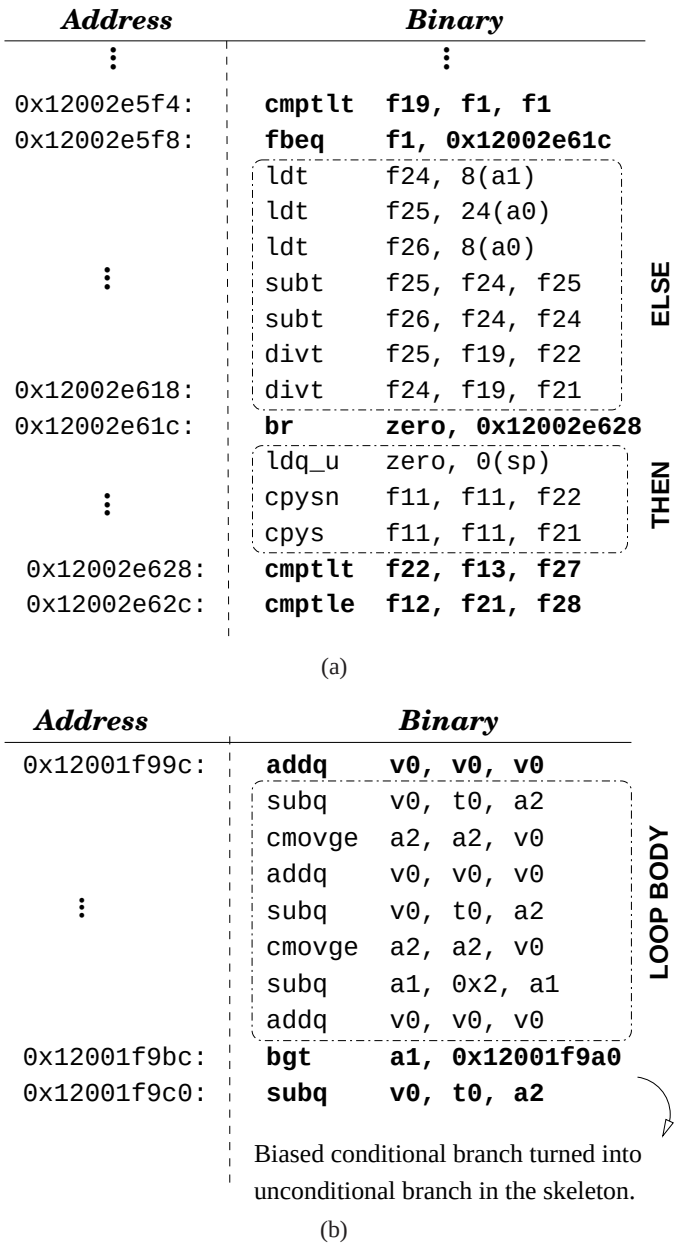


Figure 4.7: Examples of empty if-then-else block (a) and loop (b) in the skeleton of real applications. Instructions selected in the skeleton are shown in bold.

4.3.2 Cost-Effective Architectural Support

Effectively tolerating long latencies is a primary goal in lookahead. By separating correctness concerns, explicitly decoupled architecture opens up new opportunities for novel techniques for better concurrency and thus latency tolerance. Below we discuss a few straightforward

Symbol	Correctness thread interpretation and action
BDC (don't-care)	Branch prediction unavailable for this branch.
BUF (until fall through)	Branch prediction unavailable for the loop. Stop drawing from BOQ until this branch falls through.
BUT (until taken)	Branch prediction unavailable for the loop. Stop drawing from BOQ until this branch is taken.

Table 4.3: Replacing useless branches in the skeleton.

examples of probabilistic mechanisms.

4.3.2.1 Off-chip memory accesses

Stalling induced by off-chip accesses can seriously impact the optimistic thread. Given the freedom we enjoy in the optimistic core, there are quite a few options to avoid waiting for an off-chip memory access.

Simplistic value substitution

Perhaps the simplest (and indeed a simplistic) way is to give up waiting and feed some arbitrary value to the load instruction in order to *naturally* flush it out of the pipeline. This may seem senseless as a wrong value may cause the optimistic thread to veer off the correct control flow and render it irrelevant and maybe even harmful. However, there are several natural tolerance mechanisms that come to the rescue: the data loaded may not be control-flow related but is part of the prefetching effort; the load may even be dynamically dead; and the error in the value may be masked by further computation or comparisons. We show two examples of masking from real benchmarks.

Figure 4.8-(a) shows a very typical code sequence where the loaded value is compared to a constant to determine branch direction. Figure 4.8-(b) shows another kind of masking. In this example, under certain conditions, the loaded value is canceled out in the computation and no longer matters. In summary, in many cases, the exact value of a load does not matter and it is more important to flush the long-latency instruction out of the system so as to continue useful work downstream rather than to wait for the memory to respond – unless the optimistic thread is

sufficiently ahead. We only feed a substitute value when the distance between the two threads is below a threshold. This distance is measured by the number of branch predictions in the BOQ.

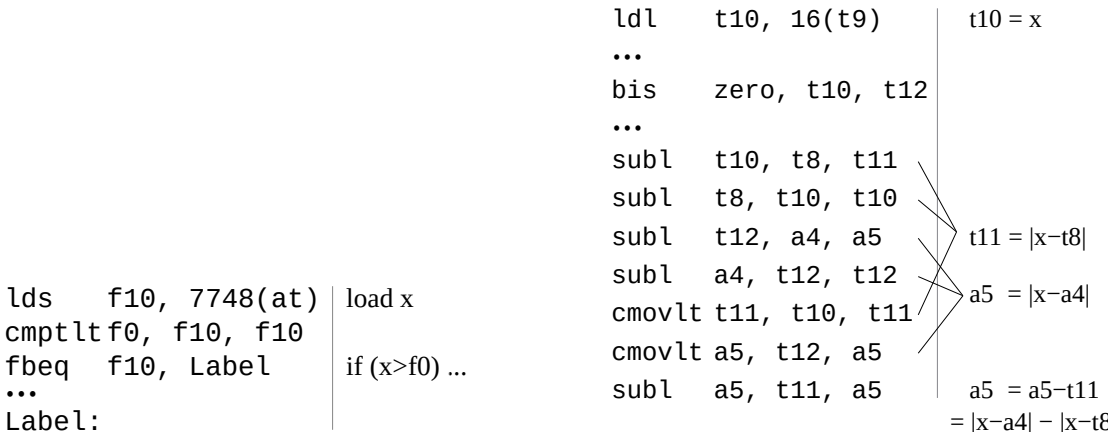


Figure 4.8: The inherent masking effect in real programs.

In terms of determining the substitute value, we can obviously use a conventional value predictor or even a special-purpose predictor [MHP05]. However, we opt for the much simpler approach of providing a 0, partly because it is the most frequently occurring value in general. The feeding of a substitute value can be triggered in a number of ways: when a load has blocked instruction commit for a while; when an outstanding memory access has timed out; or when the L2 cache explicitly notifies the core of a miss.

Explicit flushing

Another option of making L2 misses non-blocking is to explicitly flush the instruction out of the system without executing it [MSWP03]. Since there is no result from the load, its dependent instructions can not execute normally either and have to be explicitly flushed out too. This can be done by adding a poison-propagation system in the issue logic: Any dependent instruction of the load is poisoned and thus removed from the system without execution. All these instructions in turn poison their dependents. Branches are “sink” instructions beyond which the poison will not propagate. For poisoned branch instructions, because we are flushing them out instead of executing them, their original prediction naturally stands as the final result. For store instructions, there are two options: propagating the poison bit through the L0 cache or stopping the

propagation just as branches do.

Hybrid solution

These two options mentioned above have complementary pros and cons. For value substitution, even when the fed value itself is incorrect, thanks to the masking effect, we may be able to do some productive work such as prefetching using the substitute value. In contrast, explicit flushing does not even execute dependent instructions – even when the exact value of the load does not matter – and therefore will provide little boosting. On the other hand, in value-based flushing, if the fed value is used to compute the branch outcome, it may send the optimistic thread off the correct control flow and incur a costly recovery down the road. In comparison, the branch predictor is perhaps more trustworthy than the computed outcome. Therefore, we can adopt a hybrid solution that feeds the load with the value as in value-based flushing, but also propagate a poison bit as in explicit flushing, to be a warning sign that the value is not reliable. If a branch instruction is poisoned, we do not execute it but keep the prediction as the outcome.

Our evaluation shows that hybrid solution is only marginally better than simplistic value substitution. In the final evaluation we choose simplistic value substitution, since the primary benefit of our approach is its simplicity: An independent logic determines when to use value substitution and when it is used, the rest of the core is unchanged – there is no extra logic to explicitly tag results as invalid and propagate the “poison”. Secondarily, as our examples show, in some cases, the exact value may not matter much. Explicitly flushing the apparent dependence chain without executing them prevents the prefetching benefit.

Clearly, zero value substitution does not always work well. In particular, when the value is some form of an address, substituting a zero is often a sure way to get into trouble. A light-weight solution we adopted is to identify “address” loads using the parser and encode them differently to prevent the hardware from doing zero value substitution. In other words, these loads will stall if they miss in the L2. We choose this because the hardware support needed is minimum.

Pipeline stalls due to store miss

An off-chip memory access caused by a store miss can block the retirement stage of optimistic core leading to pipeline stalls. Due to freedom from correctness guarantees in the optimistic core, a few options to avoid waiting for an off-chip memory access are allowing the store to complete ① without writing its value to the L0 cache, or ② by allocating an empty line in the L0 cache to keep its value. However, for the optimistic core, the pipeline stalls due to store miss are on average only a small fraction of the total execution time ($< 1\%$). This is because a significant number of stores responsible for stalls are either removed from the skeleton or are converted to prefetches.

4.3.2.2 Delayed release of prefetches

If the optimistic thread achieves very deep lookahead, we do not want the prefetches to be issued too early. Thus we record the addresses into a *prefetch address buffer* (PAB) together with a timestamp indicating the appropriate future moment to release it. This time is set to be about one memory access latency prior to estimated execution time of the load in the correctness thread. One subtle issue we encountered is the timing of address translation. Since loading with a virtual address can cause a TLB miss, which can potentially take another off-chip access, we try to put translated address into the PAB. However, if the translation causes a TLB miss at the time of depositing the address, instead of blocking and waiting for the TLB to fill, we keep the virtual address and perform the translation when the prefetch is released from the PAB.

4.3.2.3 Managing stale data

As discussed above, the optimistic core is relying on the correctness core to provide the data when the distance between a load and its producer store is long. To allow the data to be obtained from the correctness core, *i.e.*, from the L1 cache or beyond, stale data should be removed from the L0 cache to force an access to the L1. Fortunately, the cache's replacement algorithm, which typically uses LRU-like policies, already purges some stale data out of the cache naturally. We also explored other proactive options discussed below.

First, we tried using explicit invalidation instructions in the skeleton as follows: instead

of removing a store with a long communication distance, we change it into an invalidation instruction and keep the address computation chain. This does not work as well as we expected because other variables sharing the same cache line are also evicted. This not only increases miss rate, but can cause a read of stale value in L1 if the evicted line contains newly updated variables, which is itself a potential source of recoveries. Timing is another factor. Ideally, the invalidation is issued just prior to the consumer load. In practice, we replace the stores with invalidation, resulting in early invalidations, which are particularly counter productive for stores with truly long communication distances: Not only can they evict other useful data (and potentially cause extra recoveries), but they leave a large time window for the invalidated line to be brought back before it is updated by the correctness thread. One mitigating mechanism is to queue such invalidation requests to activate at a later time. This is only a marginally effective remedy.

Second, we tried a timer-based purging mechanism of least recently used lines. Specifically, a "recently-accessed" (R) bit is added to every cache line. This bit is set upon every L0 access. A coarse-grained periodical timer triggers the invalidation of those lines with R bit not set, and clears the bit of other lines. In addition to removing stale data out of L0 cache, it also accelerates the removal of incorrect data and does lead to a lower rate of recovery. Unfortunately, increase in cache miss rate for some applications (especially integer codes) outweighs the benefit of lower recovery rate.

In all, none of these approaches brings enough consistent benefits (1-2% performance gain in our simulations) to justify the extra complexity. In our final design, we leave it to the cache replacement to probabilistically evict undesired cache lines. We couple that with periodically forced recoveries: if no recovery has occurred for a long time (150,000 cycles in our experiments), we force the optimistic core to synchronize with the correctness core, cleaning the registers and the L0.

4.3.3 Complexity Reduction

One important advantage of using explicitly decoupled architecture is the possibility to significantly reduce the circuit and design complexity of microarchitectural structures. In the opti-

mistic core, we can afford to focus only on the common case and have designs that do not always work. In the correctness core, we can use simpler algorithms and less ambitious implementations as there is less need to *aggressively* exploit ILP. It can use a simpler, throughput-optimized microarchitecture and smaller, less power-hungry structures. Because the core bears the burden of guaranteeing the correctness, a much simpler implementation can have a series of benefits. For instance, having fewer timing critical paths means that the whole core is less vulnerable to PVT variation concerns. Using a simpler and smaller core also makes fault tolerance easier and more efficient. Here we discuss a few straightforward opportunities to reduce complexity from a generic microarchitecture to serve as cores inside an explicitly decoupled architecture. We leave the exploration of special-purpose throughput-oriented design as future work.

Correctness core

Perhaps the most important simplification opportunity comes from branch handling in the correctness core, thanks to the much more accurate branch directions provided by the optimistic core. Conventional architecture requires immediate reaction upon a detected misprediction and needs the capability to (a) quickly restore register alias table (RAT) mapping; and (b) purge *only* wrong-path instructions and their state from various microarchitectural structures such as the issue queue, the load-store queue, and the re-order buffer. In contrast, because mispredictions are truly rare (see Section 4.5), the correctness core does not need instant reaction. Instead, upon the detection of a misprediction, the core can drain the right-path instructions and reset the pipeline – no partial repair is needed. Hence, checkpointing of RAT upon branch instruction is no longer necessary. This has a secondary effect of reducing the possibility of stalling when running out of RAT checkpoints.

Additionally, the characteristics of the program execution in the correctness core is different from that in a conventional core. First, cache misses are significantly mitigated. Thus, the core will be less sensitive to the reduction of in-flight instruction capacity. Second, with the optimistic core taking care of lookahead, latency of various operations becomes less critical so long as the throughput is sufficient. For example, the system's overall performance will be less sensitive to modest frequency changes in the correctness core. This makes it easy to use

conservatism to deal with variations. Third, advanced features in the microarchitecture can be avoided. For example, load-hit prediction is widely used in order to schedule dependents of loads as early as possible [Com00]. The result of such speculation is extra complexity dealing with mis-speculation and supporting *scheduling replays* [KL04]. We can do away with such speculation in the correctness core. Another example is to simplify the issue logic with some form of in-order constraints, such as in-order issue *within* any reservation station/queue. This would eliminate the circuitry to compact empty entries and simplify the wakeup-select loop.

In short, a whole array of complexity-performance tradeoffs can be performed. As we do not have quantitative models of the complexity benefit in terms of design effort reduction or critical path length reduction, we show the performance sensitivity of these complexity reduction measures in Section 4.5. In particular, we show that the performance impact of a simplification is in general much lower than in a conventional monolithic microarchitecture.

Optimistic core

Logic blocks in the optimistic core can also be simplified. Such a simplification can even trade off correctness for lower complexity and better circuit timing. Take the complex memory dependence logic for example. We first eliminate the load queue altogether, ignoring any potential replays due to memory dependency or coherence/consistency violation. We also simplify forwarding in the store queue by removing the priority encoding logic, which is considered a scalability bottleneck of store queue. Rather than relying on the priority encoder to select the right store among multiple candidates for forwarding, we ensure that at most one store (the youngest) will respond to a search. This is achieved by disabling the entry of an older store with the same address upon the issue of a new store. The circuit can be implemented with a “disabled” (D) bit per entry, which prohibits the entry from any subsequent address comparison. When a store address is resolved and broadcasted to the store queue. Any older store (not yet disabled) that matches the address will set the D bit. Any younger store that matches the address will pull down a global line to cause the incoming store’s D bit to be set. We still maintain the age-based mask generation in the circuit [Mei03] to tell the younger from the older.

Finally, we bypass TLB access and use virtual address to search the store queue, ignoring

virtual address aliasing. Conventionally, the virtual address needs to be translated before used in cache access or store queue searches. In the case of cache, we can use virtually-indexed and physically-tagged cache to overlap some of the address translation latency. Instead of doing something similar to the store queue, which further adds to the complexity, we can again ignore the rare case (of virtual page aliasing) and use (part of) the virtual address to directly search the store queue. Note that in SPEC CPU applications we studied, there is no virtual page aliasing to begin with. Furthermore, there is no need to faithfully compare all the address bits. In the simulation windows we studied, using 24 bits of address practically never resulted in any mis-forwarding. Even if only 16 bits are used, mis-forwarding still remains relatively infrequent: about once for every 20,000 instructions in integer code and once every 300,000 instructions for floating-point code.

4.4 Experimental Setup

4.4.1 Architectural support

We perform our experiments using an extensively modified version of SimpleScalar [BA97]. Support was added on top of the original simulator to simulate optimistic and correctness cores. We extended the simulator to model values in the cache in order to faithfully model the optimistic thread's execution. We propagate values along with instructions in the pipeline and instructions are emulated on the fly to correctly model the behavior of optimistic core context. For example, when a load in optimistic core consumes a value which is different from value originally used by the emulation, we force re-emulation of all its dependent instructions by propagating value down the dependency chain and register file. This also involves extra support for the simulation of branches. The baseline simulator is not purely execution-driven in that the outcome of the instruction execution is performed at dispatch time and the subsequent timing simulation does not affect instruction outcome. Thus whether a branch is mispredicted is known ahead of simulation and only those known to be mispredicted can be backtracked. However, in the explicitly decoupled architecture simulator, the outcome is indeed timing-dependent. While we make a best-effort guess at the dispatch time about whether the later branch execution will

contradict the prediction, the actual execution result can be different. In some applications, the number of branches that show this difference can be significant. Thus, we create a simulator checkpoint for every branch just as a processor does.

We also emulate a separate oracle context in lock step with optimistic core. This thread helps to indicate when the optimistic thread veers off the right path and is useful for analysis purposes. We stop emulating the oracle context as soon as optimistic core diverts from the original path, and is re-synchronized when correctness core forces a recovery. This allows the simulator to collect certain statistics otherwise hard to obtain.

4.4.2 Microarchitectural fidelity

The simulator is much enhanced to improve microarchitectural fidelity. The issue queues, register files, ROB, and LSQ are separate entities. A discrete event queue is added to allow faithful modeling of latency and contention of memory accesses. Features like load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the store queue while retiring it from ROB are all faithfully modeled [Com00]. We also changed the handling of prefetch instructions (load to ZERO register – R31). By default, the original simulator not only unnecessarily allocates an entry in the load queue, but fails to retire the instruction immediately upon execution as indicated in the alpha processor manual [Com00]. This makes them essentially blocking prefetches and can significantly affect the baseline processor's performance as we will see later. In our simulator, a prefetch neither stalls nor takes resource in the load queue. An advanced hardware-based global stream prefetcher based on [PK94; GB05b] is also implemented between the L2 cache and the main memory: On an L2 miss, the stream prefetcher detects an arbitrarily sized stride by looking at the history of past 16 L2 misses. If the stride is detected twice in the history buffer, an entry is allocated on the stream table and prefetch is generated for the next stride address. Stream table can simultaneously track 8 different streams. For a particular stream, it issues a next prefetch only when it detects the use of previously prefetched cache line by the processor. It can issue a maximum of 16 prefetches before a stream is discarded.

4.4.3 Power modeling

Both dynamic and leakage energy models are implemented. We extended Wattch [BTM00] for our design to estimate the dynamic energy component. Area estimation is needed to calculate the global clock distribution, buffer, and leakage power in our model. We attempted to rely on actual circuit synthesis to obtain the area estimation. Specifically, we used Illinois Verilog Alpha Model (IVM) and modified the hardware description to mimic the correctness core and L0 cache. Unfortunately, the process can not give a highly reliable area estimation: due to lack of access to RAM intellectual property cell libraries, in IVM, the branch predictor tables and caches are scaled down considerably to permit hardware synthesis. Instead of having 4K or 8K entries in various predictor tables (Table 4.4), the implementation caps the size of all tables to no more than 512 entries. Under these conditions, the synthesis result shows that the area overhead due to adding an extra core and the L0 to be about 64% that of the baseline core. For the architectural simulations, however, we conservatively assumed the additional area is equivalent to a full-blown core plus a full-sized L1 cache – a 100% overhead in core area. Even under this assumption, the overall chip area penalty is still limited as the L2 cache and other system logic are shared. In our floorplan based on the scaled-down Alpha 21364, the total chip overhead is just about 16%.

Leakage power is temperature-dependent and computed based on predictive SPICE circuit simulations for 45nm technology using BSIM3 [BSI99]. We used HotSpot [SSH+03] to model dynamic temperature variation across the proposed floorplan. Floorplans for both baseline and explicitly decoupled architecture are derived from the scaled-down version of Alpha 21364 floorplan for 45nm. We base device parameters on the 2004 ITRS projection of 45nm CMOS technology file. We make aggressive assumptions on leakage to reduce residual energy consumption during idling. Specifically, we reduced the ITRS projected 65nm source-drain leakage current density and gate-oxide leakage current density by a factor of 5x and 10x respectively, factoring in the recent advances in leakage current reduction [BCGM07; GJT+07]. Without this adjustment, the energy result would be even more favorable for our design as for most applications, explicitly decoupled architecture consumes less leakage energy. This is because while the execution time reduction is quite significant for many applications, the area

increase is rather limited in comparison. Finally, we use aggressive conditional clocking in the model (*_cc3* in Wattch).

Baseline core	
Fetch/Decode/Commit width/ROB	8 / 4 / 6 /128
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div
Issue queue / Registers (INT, FP)	(32, 32) / (80, 80)
LSQ(LQ,SQ) 2 search ports	64 (32,32)
Branch predictor	Bimodal and Gshare combined
- Gshare	8K entries, 13 bit history
- Bimodal/Meta table/BTB	4K/8K/4K (4 way) entries
Branch misprediction penalty	at least 7 cycles
L1 instruction cache (Not shared)	64KB, 1-way, 128B line, 2 cycles
L1 data cache	32KB, 4-way, 64B line, 2 cycles, 2 ports
L2 unified cache (shared)	1MB, 8-way, 128B line 15 cycles
Memory access latency	400 cycles
Aggressive core	
Fetch/Decode/Commit width/ROB	16 / 6 / 12 / 512
Functional units	INT 3+1 mul +1 div, FP 3+1 mul +1 div
Issue queue / Registers (INT, FP)	(64, 64) / (400, 400)
LSQ(LQ,SQ) 2 search ports	128 (64,64)
Branch predictor	Bimodal and Gshare combined
- Gshare	1M entries, 20 bit history
- Bimodal/Meta table/BTB	1M/1M/64K (4 way) entries
Branch misprediction penalty	at least 7 cycles
L1 instruction cache (Not shared)	128KB, 2-way, 128B line, 2 cycles
L1 data cache	48KB, 6-way, 64B line, 2 cycles, 3 ports
L2 unified cache (shared)	1MB, 8-way, 128B line 15 cycles
Memory access latency	400 cycles
Correctness core: Baseline core without branch predictor and with circuit simplifications to RAT logic as discussed in Section 4.3.3.	
Optimistic core: Baseline core with microarchitectural designs discussed in Section 4.2.2. L0 cache: (16KB, 4-way, 32B line, 2 cycle, 2 ports). Round trip latency to L1 is 6 cycles	
Communication: BOQ: 512 entries; PAB: 256 entries; register copy latency (during recovery): 32 cycles	
Process specifications: Feature Size: 45nm; Frequency: 3 GHz; V_{dd}: 1 V	

Table 4.4: System configuration.

4.4.4 Applications, inputs, and architectural configurations

We use highly-optimized Alpha binaries of SPEC CPU2000 benchmarks and SPLASH-2 suite. For profiling, we use the *train* input and run the applications to completion. For evaluation, we simulate 100 million instructions after skipping over the initialization portion as indicated in [SC00] using *ref* input. Our baseline core configuration, is a generic high-end microarchitecture loosely modeled after POWER4's core [TDF+02]. To provide a reference for the lookahead effect of explicitly decoupled architecture, we also use a very aggressively configured core. Details of the configurations are shown in Table 4.4.

4.5 Experimental Analysis

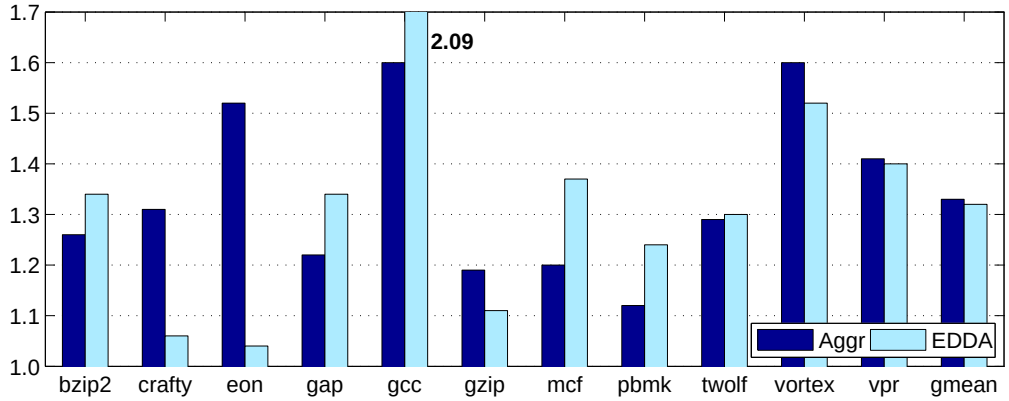
4.5.1 Benefit analysis

A primary benefit of explicitly decoupled architecture is that it allows *complexity-effective* designs – by using simple mechanisms but targeting high-impact performance bottlenecks. explicitly decoupled architecture allows design effort to be focused more on exploring new opportunities rather than on ensuring an optimization technique actually works in silicon all the time. Unfortunately, we are not yet capable of quantifying design effort nor can our current design – far from mature – be used to convincingly justify the upfront cost of explicit separation and the resulting partial redundancy.

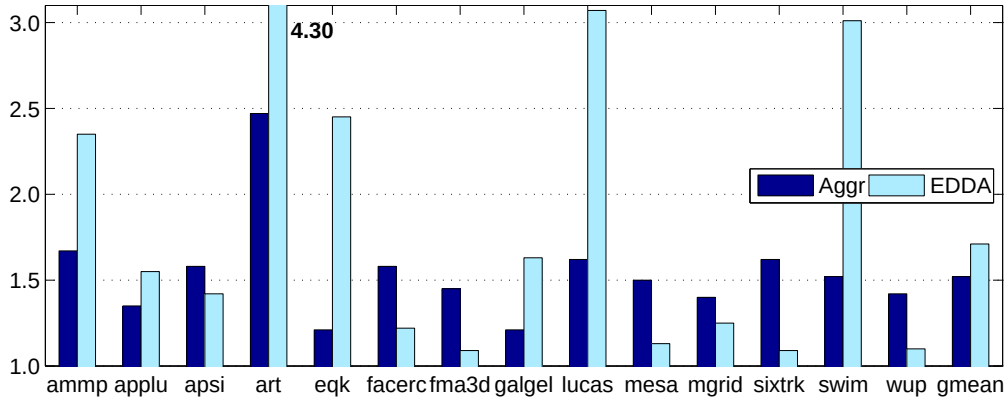
In the following we hope to offer some evidence that this paradigm is worth further exploration. A particular point we want to emphasize is the no single aspect (*e.g.*, absolute performance) taken in isolation can be construed as a figure of merit. Put together, these results show that even with only intuitive and straightforward techniques, the discussed design still (a) achieves good performance boosting, (b) does not consume excessive energy, and (c) provides robust performance and better tolerance than conventional design to circuit-level issues and to the resulting conservatism.

Performance gain of optimism

Figure 4.9 shows the speedup of the proposed explicitly decoupled architecture over a baseline core and the speedup of expanding the baseline core into an impractically aggressive monolithic core. Since our explicitly decoupled architecture is performing traditional ILP lookahead, it is not surprising to see the two options follow the same trend: in general, floating-point codes tend to benefit more noticeably. On average, speedup achieved by explicitly decoupled architecture is more pronounced. The geometric means are 1.32 (INT) and 1.71 (FP) compared to the aggressive core's 1.33 (INT) and 1.52 (FP). Table 4.5 shows the detailed IPC results.



(a) Integer applications.



(b) Floating-point applications.

Figure 4.9: Speedup of proposed explicitly decoupled architecture (EDDA) and an aggressively configured monolithic core (Aggr.) over baseline (BL) for SPEC INT (a) and FP (b) and the geometric means.

While we are focusing on ILP exploitation, the effect can be equally important in explicitly

	bzip	crafty	eon	gap	gcc	gzip	mcf	perlbmk	twolf	vortex	vpr
BL	1.30	2.24	2.61	1.82	2.19	2.07	0.59	0.91	0.57	1.31	1.27
Aggr.	1.63	2.93	3.96	2.23	3.52	2.45	0.71	1.01	0.73	2.10	1.79
EDDA	1.74	2.38	2.73	2.44	4.59	2.29	0.81	1.12	0.73	2.00	1.78

(a) Integer applications.

amm	app	apsi	art	eqk	fac	fma	gal	luc	mesa	mgr	six	swim	wup
0.78	1.78	1.72	0.28	1.08	2.94	2.70	2.55	0.56	2.98	3.05	2.85	1.25	3.75
1.30	2.40	2.73	0.68	1.30	4.66	3.91	3.08	0.91	4.46	4.28	4.60	1.90	5.32
1.83	2.75	2.45	1.19	2.63	3.59	2.94	4.15	1.72	3.36	3.82	3.12	3.77	4.11

(b) Floating-point applications.

Table 4.5: Detailed IPC results of baseline (BL), proposed explicitly decoupled architecture (EDDA), and an aggressively configured monolithic core (Aggr.) for SPEC INT (c) and FP (d).

parallel programs. We have applied the same methodology to parallel programs – ignoring any opportunity to target shared-memory issues in the design of explicitly decoupled architecture – and have also observed significant performance gains. Figure 4.10 summarizes the performance gain of SPLASH applications running on a CMP with explicitly decoupled cores compared to that with conventional cores. The configuration of each CMP core is same as defined in Table 4.4. To mimic realistic cache miss rates for SPLASH applications, we use reduced cache sizes (as recommended in [WOT+95]): 8 KB for L1 caches and and 256 KB for the L2 cache. With this configuration their global L2 miss rate closely matches with the miss rate of SPEC CPU2000 integer applications in the baseline cache configurations. If the cache sizes increase further, the data tend to fit within the caches and the miss rates become unrealistically small. As a reference, a configuration with twice as many cores (16-way) is also shown. For 16-way CMP we also double the size of L2 cache to avoid any increase in global L2 miss rate due to change in working set size with additional threads. An important point to note is that exploiting ILP is not guaranteed to be less effective than exploiting TLP (thread-level parallelism) for parallel codes. As can be seen, even for these highly-tuned scientific applications, scaling to more cores does not give perfectly linear speedup and in some cases producing (far) lower return than improving ILP. Exploring multiprocessor-aware optimistic techniques in explicitly decoupled architecture will likely unleash even more performance potential.

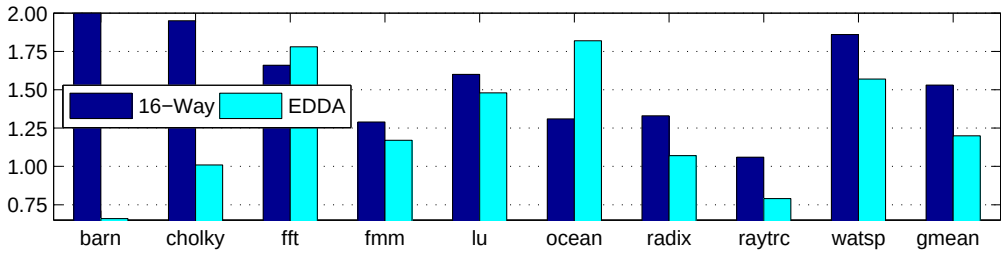


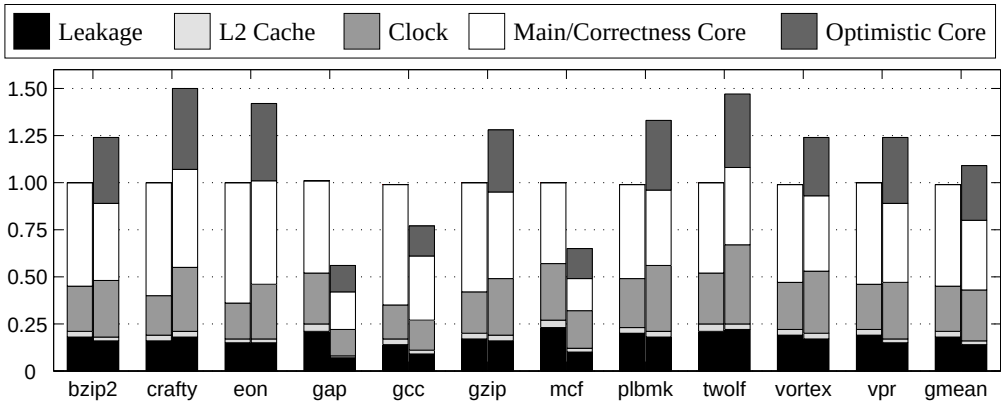
Figure 4.10: Speedup of SPLASH applications running on an 8-way CMP. Each conventional core in CMP system is replaced by an explicitly decoupled core. For contrast, the speedup of a 16-way CMP (also doubling the size of L2) is also shown. In some cases, enhancing ILP even outperforms doubling the number of cores.

The results suggest that decoupled lookahead can uncover some parallelism not already exploited in a state-of-the-art microarchitecture, at least for some applications. In some cases, the return is significant. Note that the specific numbers are secondary as the design point is chosen to illustrate the potential rather than to showcase an optimized design. Indeed, as we show later, the correctness core can be much less aggressive with virtually no performance impact. The key point is that this is achieved with simple techniques designed to minimize circuit-implementation challenges. Given the correctness decoupling, the entry barrier for implementing other optimizations in the performance domain should be considerably lower than in a conventional system. More investigation would discover other "low-hanging fruits" to achieve high performance with low complexity. In turn, the performance "currency" can be used to pay for other important design goals.

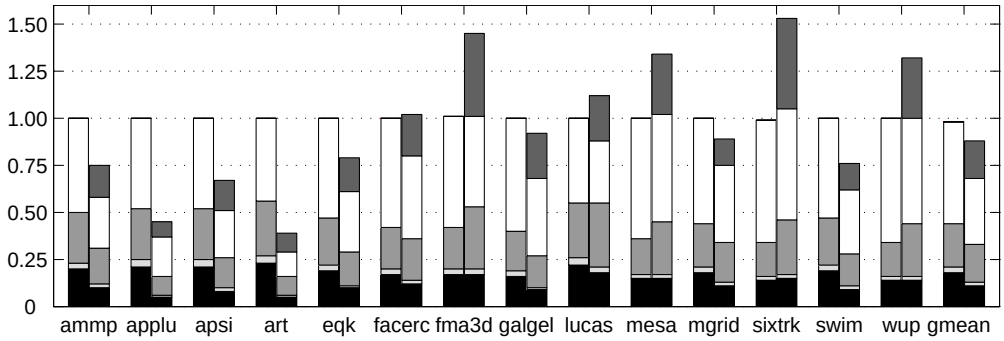
Energy implications

An apparent disadvantage of an explicitly decoupled architecture is that it is power-inefficient: roughly two cores are used and the program needs to execute twice and this may lead one to believe that energy consumption would double. In reality, the energy overhead can be far less due to a number of factors. First, the skeleton is after all not the entire program and in certain applications can be quite small (Section 4.5.2). Second, many energy components do not double. For instance, only the optimistic core wastes energy executing wrong-path instructions following mispredicted branches. Third, when a prefetch is successful, it is a good energy tradeoff to execute a few more instructions to avoid a long-latency stall which burns power while do-

ing nothing. As can be seen in Figure 4.11, which shows normalized energy consumption with breakdown in both explicitly decoupled architecture (EDDA) and baseline systems, a significant performance improvement results in lower energy consumption in EDDA. Indeed, explicitly decoupled architecture results in an average of 11% energy *reduction* for floating-point applications. Even for integer codes, the energy overhead of explicitly decoupled architecture is only 10%.



(a) Integer applications.



(b) Floating-point applications.

Figure 4.11: Normalized energy consumption of explicitly decoupled architecture (EDDA) (right bar) and baseline systems (left bar). Each bar is further broken down into 5 different sub-categories.

While there is clearly room for improvement, we note that the study is conservative in that the explicitly decoupled architecture configuration is hardly optimized for energy-efficiency. For instance, no attempt is made to shut down the optimistic core when it is not effective and no energy benefits of the architectural simplifications (Section 4.3.3) are taken into account. Moreover, aggressive assumptions are made to reduce residual energy consumption during idling

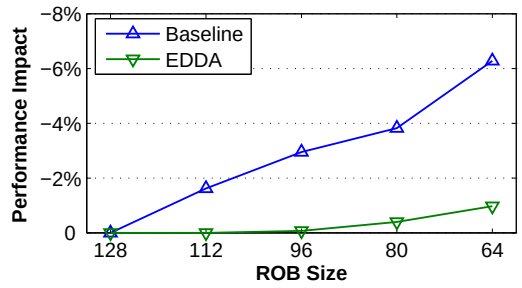
(discussed in Section 4.4).

Performance cost of conservatism in explicitly decoupled architecture

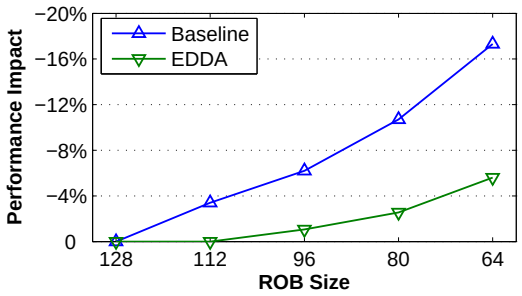
Unlike the optimistic core where timing glitches or logic errors do not pose a threat to system integrity, the correctness core faces the challenge that it has to be logically correct and functioning reliably despite adverse runtime conditions such as unpredictable PVT variations. Passive conservatism is perhaps still a most practical and effective approach to dealing with the challenge: avoid complexity in the logic design, mitigate timing critical paths, and build in operating margins. The opportunity in an explicitly decoupled architecture is that these acts of conservatism do not carry as high a (performance) price tag as in a conventional system. Below, we show that indeed, when we increase the conservativeness in the design and configuration of the optimistic core, the performance impact is insignificant and much less pronounced than doing so in a conventional monolithic design.

First, we hypothesized earlier that the correctness core will be less sensitive to in-flight instruction capacity as it does not need to rely on aggressive ILP exploitation. We study this by gradually scaling down the microarchitectural resources. Figure 4.12 shows the average performance impact. We scale other resources (issue queue, load-store queue, and renaming registers) proportionally with the size of ROB. We contrast this sensitivity with that of a conventional system. The figure clearly shows a much slower rising curve for explicitly decoupled architecture (EDDA) indicating less performance degradation due to reduction of microarchitecture “depth”.

Second, the microarchitectural design of the correctness core can be simplified by avoiding complex performance features. We discussed eliminating load-hit speculation and the necessary scheduling replay support in Section 4.3.3. Figure 4.13-(a) shows the performance impact in explicitly decoupled architecture and in a conventional core. Clearly, load-hit speculation is a useful technique in a conventional core. With decoupled lookahead, its utility in the correctness core is largely negligible – in fact, on average, we even achieve a slight performance improvement for floating-point applications. This is mainly because with effective lookahead, in floating-point applications, there are abundant ready instructions in the correctness core. Therefore, there is more potential cost and little benefit in doing load-hit speculation. Figure 4.13-(b)



(a) Integer applications.



(b) Floating-point applications.

Figure 4.12: Performance impact on Baseline and explicitly decoupled architecture (EDDA) system with reduction in in-flight instruction capacity.

shows the impact of simplifying the integer issue queue to be in-order. Similarly, this is creating a smaller performance penalty than would be in a conventional core. Again, this is because the throughput for integer instructions is generally sufficient and the resulting serialization is less costly than in a conventional core.

Third, building in extra timing margin is a simple way to increase a chip's reliability under runtime variations. However, in a conventional system, such margin directly impacts performance. In an explicitly decoupled architecture, building in timing margin for the correctness core has less direct impact thanks to decoupling. To demonstrate this, we reduce the frequency of the correctness core by 10%. We show the performance degradation and contrast that to the conventional system under the same frequency reduction. From Figure 4.13-(c), we see that the EDDA system has a much smaller sensitivity to such modest frequency reduction. Performance degradation is less than 2% on average, 4-5 times smaller than that of a conventional system.

In summary, as expected, the overall EDDA's performance is insensitive to the simplification or extra conservatism introduced to the correctness core – so long as its throughput is sufficient.

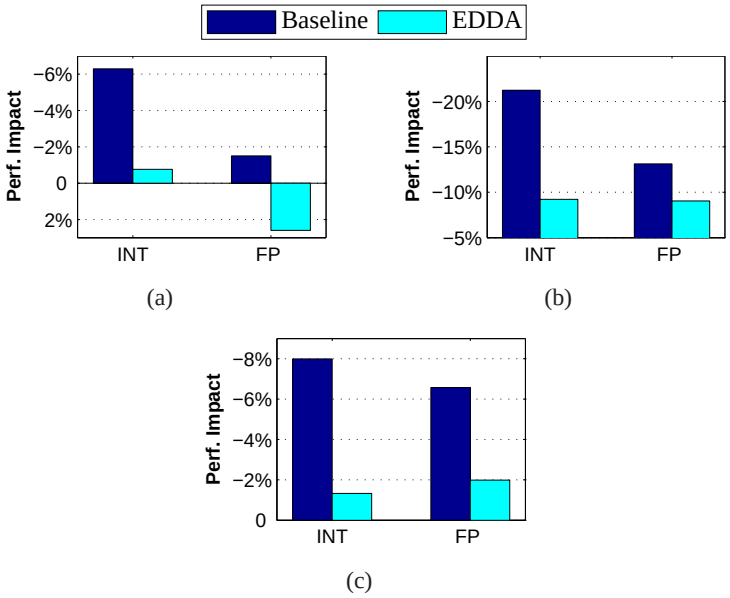


Figure 4.13: Performance impact of architectural simplification – removing load-hit speculation mechanism (a) and in-order int issue queue (b) and modest clock frequency reduction (c) – in Baseline (BL) and explicitly decoupled (EDDA) systems.

This means that performance gained using complexity-effective optimistic techniques can be spent to reduce design effort and improve system integrity.

Sensitivity of performance domain circuit errors

In our explicitly decoupled architecture, the correctness core is not merely providing a correctness safety net and we do not compare the entire output from the two cores and resynchronize whenever there is a difference as some other designs with similar lead/trailing cores such as DIVA [Aus99], Tandem [MMR07], and Paceline [GT07]. In our current implementation, we only resynchronize when the leading thread veers off the right control flow. (In a future incarnation, even this could be relaxed.) This difference allows the optimistic core to be even less affected by circuit errors (such as due to insufficient margins) than the lead cores in these related designs. This can be shown in a very limited experiment by systematically injecting errors into the committed results and observe their impact on our explicitly decoupled architecture.

Our injection is different in two respects from common practice. First, we inject multi-bit errors. Unlike particle-induced errors which are found to cause mostly single-bit errors –

largely due to limited energy a particle transfers to the silicon, a timing margin failure can cause multi-bit errors. Second, we only inject errors into committed results since we are only interested in finding out the masking effect of our explicitly decoupled architecture execution model. We use systematic sampling and flip all bits of the result of a committed instruction. Our simulator, which tracks value in all microarchitectural components, allows us to faithfully track the propagation of errors. Figure 4.14 shows the average number of recoveries incurred per injection at different injection frequencies.

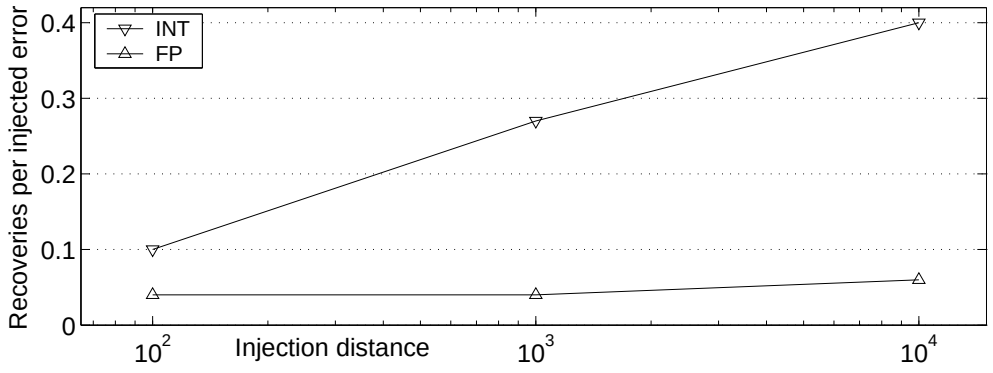


Figure 4.14: Recoveries per injected error as a function of error injection distance (number of committed instructions per injected error).

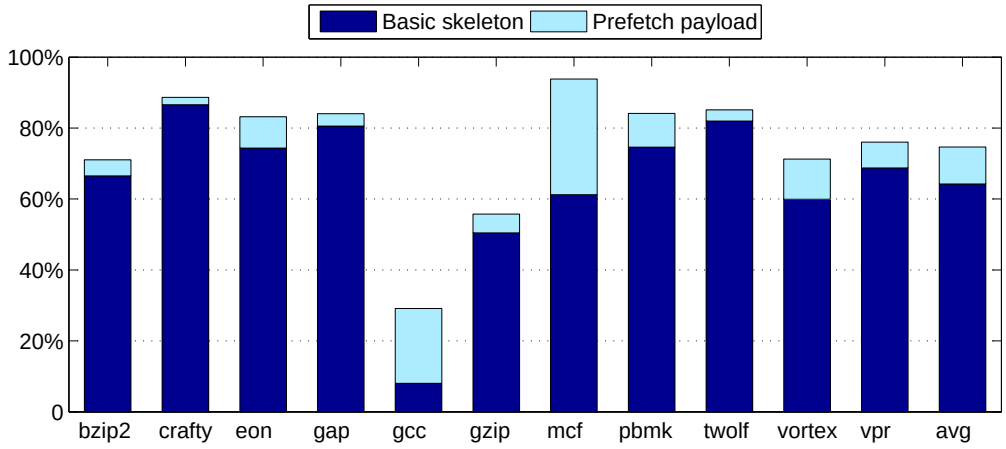
We see that the number varies with the application and in some cases, the lookahead activities can intrinsically tolerate a large degree of circuit operation imperfection. For example, for floating-point applications, out of 20 injected errors, only 1 results in a recovery. Even when circuit errors happen once every 1000 instructions, recovery due to circuit errors will be insignificant. For integer applications, the rate depends on error injection frequency: as error frequency increases, the chance of an error causing a recovery reduces. Intuitively, this is because of increasing likelihood of a single recovery fixing multiple latent errors.

4.5.2 System diagnosis

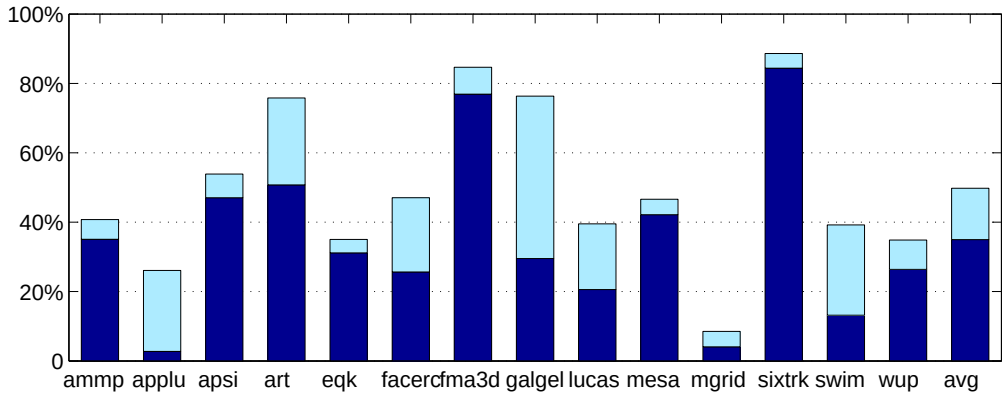
Next, we discuss detailed statistics to help understand the behavior of the system.

Skeleton

We first look at the skeleton generated by the parser. Figure 4.15 shows the size of the skeleton as the percentage of dynamic instructions in the original binary excluding the NOPs. As would be expected, the skeleton of the integer applications are bigger due to the more complex control flow, whereas the skeleton for floating-point applications are much leaner. On average, excluding prefetches and the special branches (shown in Table 4.3), the skeleton contains an average of 68% (INT) and 34% (FP) of the instructions from the original binary.



(a) Integer applications.



(b) Floating-point applications.

Figure 4.15: Percentage of dynamic instructions left in the skeleton.

Controlled by a separate thread, the lookahead effort on the optimistic core is no longer tightly bound to (and slowed down by) the actual computation the program has to perform. As seen in Figure 4.15, the skeleton is quite a bit shorter than the original binary. For some

applications, the skeleton also becomes less memory-bound and thus stalled less often when executing. To put the code size reduction into perspective, if we use the conventional baseline microarchitecture to serve as the optimistic core, the effect of the skeletal execution alone can achieve speedup of 0.97 to 1.89 with a geometric mean of 1.17 in integer codes and 1.07 to 3.01 with a geometric mean of 1.54 in floating-point codes.

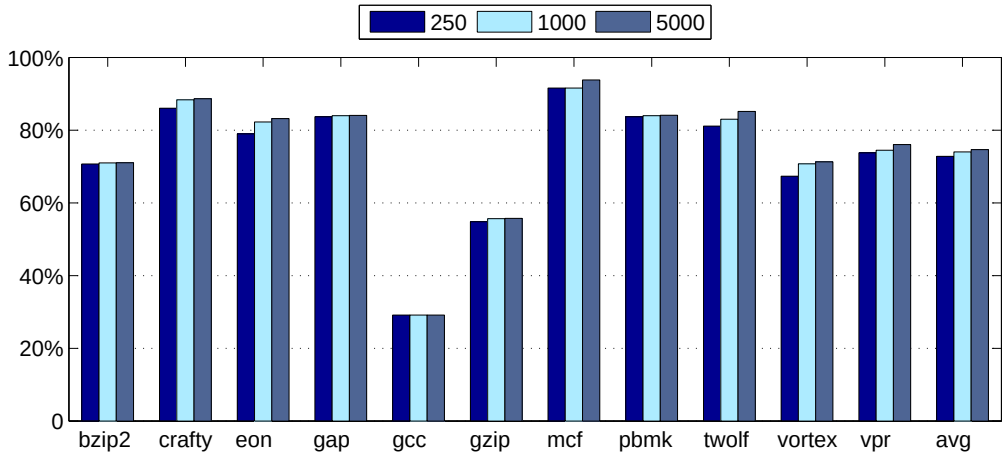
In Section 4.3.1 we choose the value of d_{th} to 5000. Choosing a smaller d_{th} improves the chance of further reducing the skeleton size, however, it also reduces the maximum lead that can be maintained between optimistic and correctness thread. For most of the applications, Figure 4.16 shows a insignificant change in the size of skeleton when d_{th} is set to 1000 or 250. In general distance between dynamic store instance and its consumer load is either very small or very large. Hence, the profile results are not sensitive to d_{th}.

Architectural techniques

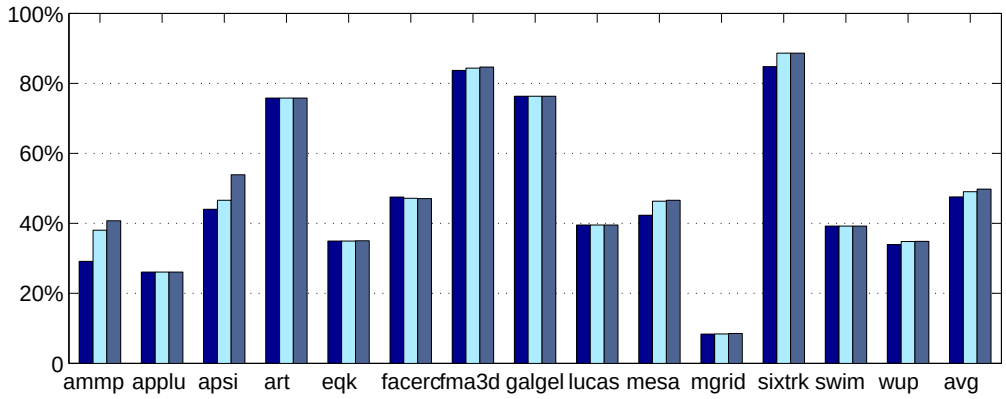
Within the performance domain, one can employ optimization mechanisms without complex backups to handle mis-speculation or other contingencies. Zero value substitution discussed in Section 4.2.2 is one example of low-complexity techniques possible in explicitly decoupled architecture. This simple change resulted in a modest increase in the number of recoveries (about 1.12 per 10,000 committed instructions in integer programs) but allows a net performance benefit of about 12%[2].

In addition to adding mechanisms for performance gain, we can also take away implementation complexity. For example, in the optimistic core, we drastically simplified the load-store queue logic at the expense of correctness (recall that the load queue is completely removed and the priority logic in the store queue is also removed). On average, in 10,000 instructions, less than 0.18 loads receive incorrect forwarding, adding less than 0.05 recoveries. The overall performance cost is virtually negligible (included in results shown in Figure 4.9). Figure 4.17 shows the recovery rate due to software and hardware approximations. Note that even in the extreme cases of *gap* and *twolf*, the benefit of lookahead still outweighs the overhead of recover-

[2] A recovery causes the correctness core to completely drain the pipeline before copying the registers. This is faithfully modeled and statistics show that the actual penalty is about 150 cycles on average. With these recovery frequencies, the latency of register copying is not a significant factor. Doubling the latency to 64 cycles will add about 1% overhead to integer applications and even less for floating-point applications.



(a) Integer applications



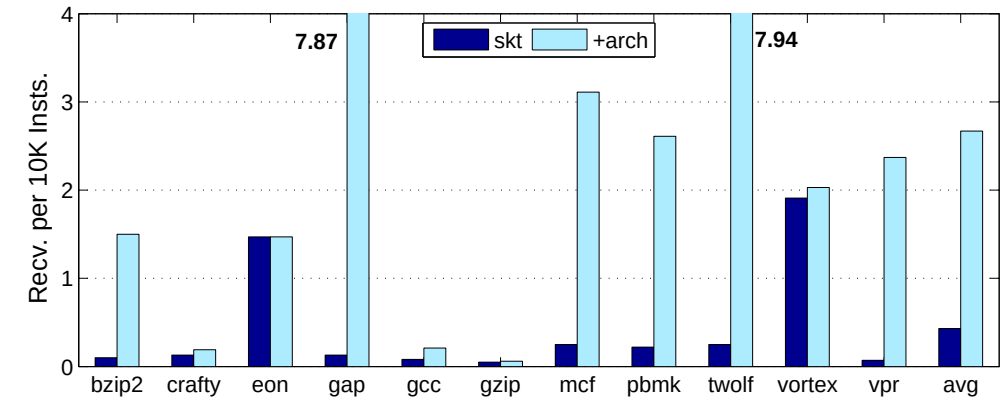
(b) Floating-point applications

Figure 4.16: Comparison of skeleton size for different d_{th}'s (Section 4.3.1).

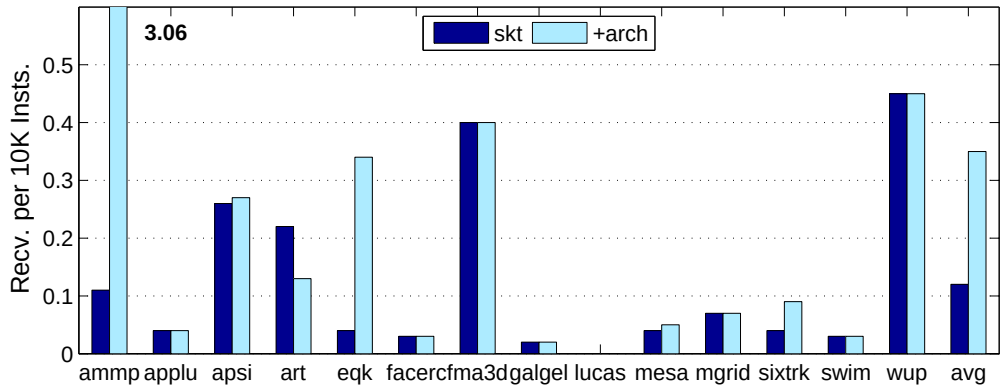
ies. Also note that while the architectural behavior and the overhead of recoveries are faithfully modeled, the potential positive impact on cycle time or load latency is not considered in our analysis.

Lookahead effects

When the optimistic thread can sustain deep lookahead, it is not difficult to understand why the correctness core is sped up. In our system, both branch mispredictions and cache misses are significantly mitigated in the trailing correctness core. On average, the reduction in misprediction is 90% and 89% for integer and floating-point applications respectively. The reduction in L2 misses, as shown in Table 4.6 is 92% (INT) and 94% (FP). The L1 misses reduce by 93%



(a) Integer applications.



(b) Floating-point applications.

Figure 4.17: The number of recoveries per 10,000 committed correctness thread instructions. "skt" shows the number due to skeletal execution and "+arch" shows the result after enabling architectural changes in the optimistic core.

(INT) and 94% (FP) on average.

There is evidence that the correctness core – which is just a simplified conventional core – is sometimes throughput saturated. A key issue in explicitly decoupled architecture design is how to support wider pipeline and higher throughput in a complexity-effective way. This is part of our future work.

Finally, we note that backing L0 cache with L1 does not increase the burden of the L1 cache. In fact, with the exception of only a few applications, the total traffic to L1 is reduced, thanks to the reduction in wrong-path instructions in the correctness core. The traffic reduction averages 17%(INT) and 11% (FP) and can be as much as 53%. Accesses from L0 only account for an average of 6% (INT) and 7% (FP) of the total L1 accesses.

	bzip	crafty	eon	gap	gcc	gzip	mcf	perlbmk	twolf	vortex	vpr	avg.
MR	0.46	0.02	0.0	0.83	0.01	0.01	3.2	0.8	1.03	0.3	0.29	0.63
RD	93.2	93.8	94.5	99.7	92.8	99.5	97.2	87.9	72.3	88.2	98.2	92.5

(a) Integer applications.

amm	app	apsi	art	eqk	fac	fma	gal	luc	mesa	mgr	six	swim	wup	avg.
1.25	0.72	0.51	16.44	0.36	1.24	0.00	0.02	2.87	0.00	0.22	0.01	1.66	0.01	1.81
95.3	99.9	99.2	99.2	96.6	99.2	93.9	99.6	47.8	99.5	99.6	94.7	99.7	96.3	94.3

(b) Floating-point applications.

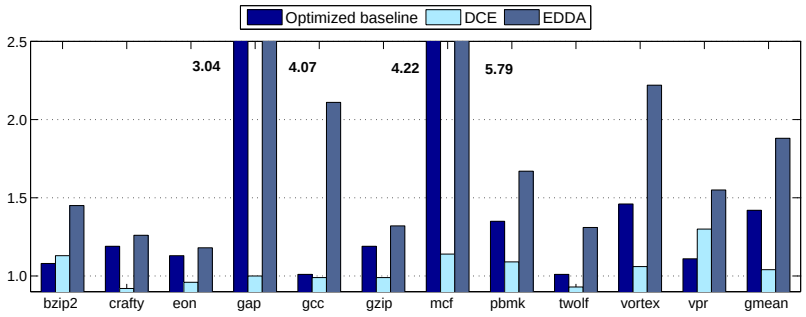
Table 4.6: Global L2 blocking miss rate (MR %) for baseline system (with an aggressive hardware prefetcher) and percentage reduction (RD %) by using explicitly decoupled architecture (without a hardware prefetcher) for correctness core.

Comparison with dual-core execution

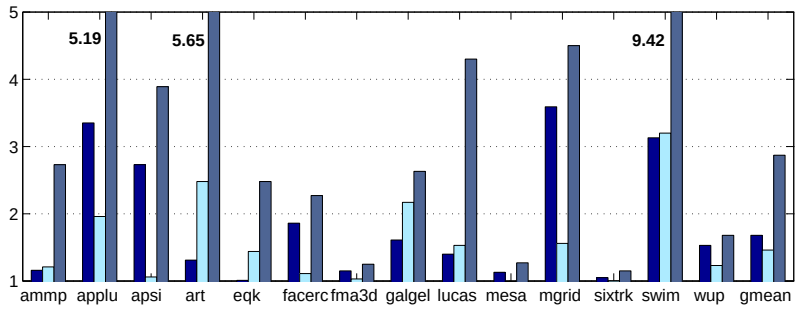
If we reduce explicitly decoupled architecture to a purely performance enhancing mechanism, it resembles a class of techniques represented by decoupled access/execute architecture [Smi84], Slip-stream [PSR00; SPR00], dual-core execution (DCE) [Zho05], Flea-flicker [BNS+03b], Tandem [MMR07], and Paceline [GT07]. Among these, we compare to DCE as architecturally, it is perhaps the most closely related design: both try to avoid long-latency cache miss-induced stalls to improve performance.

Before discussing the statistics, we note that a key design difference stems from the different design goals and the resulting constraints. DCE (and to varying extents, Slip-stream, Flea-flicker, Tandem, and Paceline) focuses on *peripherally augmenting* an existing multi-core microarchitecture. Explicitly decoupled architecture emphasizes on *changing* conventional design practice and making microarchitecture more decoupled (to reduce implementation complexity and increase resilience to variation-related conservatism). We *customize* the leading core/thread specifically for lookahead. Our lookahead is not slowed down by unrelated normal computation, and the optimistic architectural design takes full advantage of correctness non-criticality. In contrast, the leading core/thread in these three different designs all execute almost the entire program and use a hardware that is designed with proper execution rather than lookahead in mind. Note that achieving clock speed improvement in the lead core as done in Tandem and Paceline is orthogonal to our design (which improves IPC) and can therefore be incorporated.

In Figure 4.18, we show the speedups. We note that our reproduction of DCE based on the proposal is only a best-effort approximation. Not all details can be obtained, such as the baseline's prefetcher design. We used our own instead. In order not to inflate the benefit of explicitly decoupled architecture, we chose the best configuration for a sophisticated stream prefetcher [GB05b]. Without this prefetcher, the baseline performance will be lowered by an average of 1.56X and as much as 4X. Finally, as discussed earlier, fixing the modeling of prefetch instructions allows sometimes dramatic performance differences (*e.g.*, in *swim*). Without this change, our version of DCE obtains performance improvements that match [Zho05] relatively well. For better direct comparison, we shift the basis of normalization in Figure 4.18 to the suboptimal baseline and show our default baseline as the "optimized" baseline. Our lookahead-specific design understandably provides more performance boosting.



(a) Integer applications



(b) Floating-point applications

Figure 4.18: Comparison of explicitly decoupled architecture (EDDA) and dual-core execution (DCE). All results are shown as speedup relative to the suboptimal simulator baseline. The optimized baseline, which has been used throughout the thesis, is also shown.

4.6 Recap

In this chapter, we have shown that an explicitly-decoupled implementation can competently perform lookahead and deliver solid performance improvement. This is done without requiring complex circuitry that is challenging to implement. Explicit decoupling is a key enabling factor. Note that explicitly decoupled architecture does have an up-front cost in infrastructure. However, with further exploration, more novel techniques can be integrated to amortize the cost. In the next chapter, we will look at more elaborate optimization for the optimistic core.

Chapter 5

Speculative Parallelization in Decoupled Look-ahead

In this chapter, we explore speculative parallelization in a decoupled look-ahead agent (optimistic core). Intuitively, speculative parallelization is aptly suited to the task of boosting the speed of the decoupled look-ahead agent for two reasons. First, the code slice responsible for look-ahead does not contain all the data dependences embedded in the original program, providing more opportunities for speculative parallelization. Second, the execution of the slice is only for look-ahead purposes and thus the environment is inherently more tolerant of dependence violations. We find these intuitions to be largely born out by experiments and that speculative parallelization can achieve significant performance benefits at a much lower cost than needed in a general purpose environment.

The rest of this chapter is organized as follows: Section 5.1 discusses background; Section 5.2 details the architecture design; Sections 5.3 and 5.4 show the experimental setup and discuss experimental analysis of the design.

5.1 Background

The common purpose of look-ahead is to mitigate misprediction and cache miss penalties by essentially overlapping these penalties (or "bubbles") with normal execution. In contrast, speculative parallelization (or thread-level speculation) intends to overlap normal execution (with embedded bubbles) of different code segments [SBV95; RS01; ZS02; AD98; SM98; CMT00; BS06]. If the early execution of a future code segment violates certain dependences, the correctness of the execution is threatened. When a dependence violation happens, usually the speculative execution is squashed. A consequence of the correctness issue is that the architecture needs to carefully track memory accesses in order to detect (potential) dependence violations. In the case of look-ahead execution, by executing the backward slices early, some dependences will also be violated, but the consequence is less severe: (*e.g.*, less effective latency hiding). This makes tracking of dependence violations potentially unnecessary. Because of the non-critical nature of look-ahead thread, exploiting speculative parallelization in the look-ahead context is less demanding. In some applications, notably integer codes represented by SPEC integer benchmarks, the performance of a decoupled look-ahead system is often limited by the speed of the look-ahead thread, making its speculative parallelization potentially profitable.

5.2 Architectural Design

5.2.1 Baseline Decoupled Look-ahead Architecture

We build support for speculative parallelization in the look-ahead system explained in Chapter 4. To recap, in this system, shown in Figure 4.1, a statically generated look-ahead thread executes on a separate core (optimistic core) and provides branch prediction and prefetching assistance to the main thread (correctness core). Next we will discuss the design of software and hardware support to enable speculative parallelization.

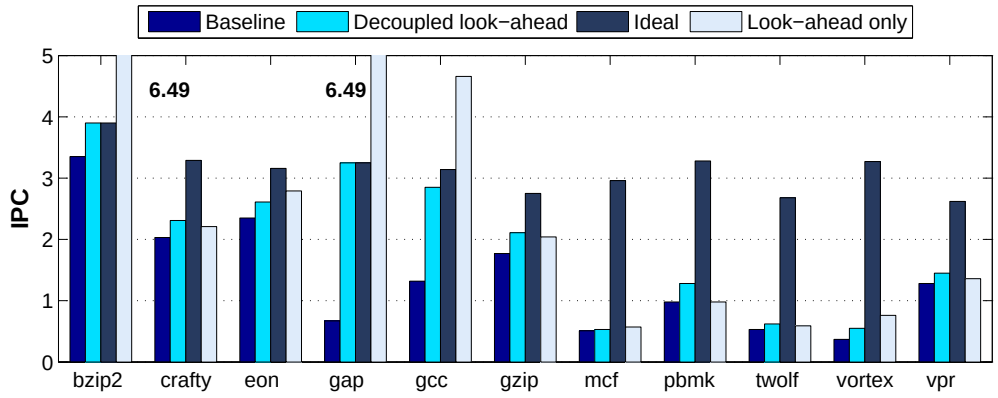
Motivation for look-ahead thread parallelization

With two threads running at the same time, the performance is dictated by whichever runs slower. If the look-ahead thread is doing a good enough job, the speed of the duo will be determined by how much execution parallelism can be extracted by the core. The performance potential can be approximated by idealizing cache hierarchy and branch prediction. Note that idealization tends to produces a loose upper-bound. On the other hand, when the look-ahead thread is the bottleneck, the main thread can only be accelerated up to the speed of the look-ahead thread running alone, which is another performance upper-bound. While the details of the experimental setup will be discussed later, Figure 5.1 shows the performance of a baseline decoupled look-ahead system described above measured against the two upper-bounds just discussed. As a reference point, the performance of running the original binary on a single core in the decoupled look-ahead system is also shown.

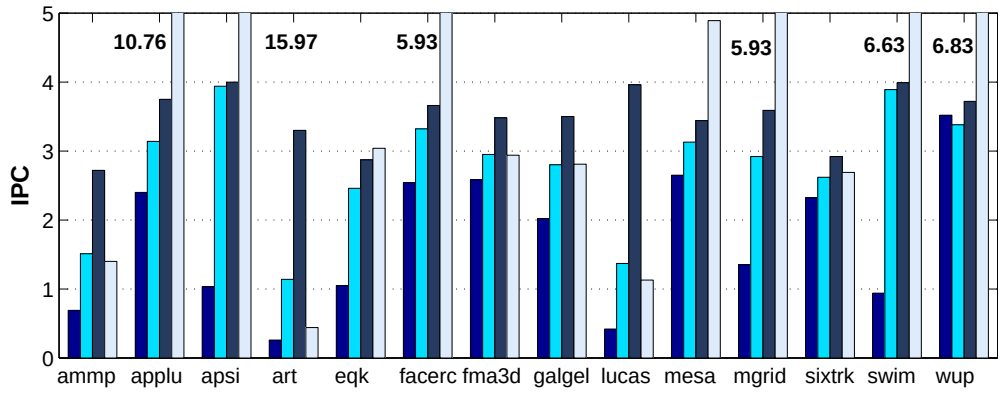
The applications can be roughly grouped into three categories. First, for 9 applications (*bzip*, *gap*, *gcc*, *apsi*, *facerec*, *mesa*, *sixtrack*, *swim*, and *wupwise*), the baseline look-ahead is quite effective, allowing the main thread to sustain IPC close to that in the ideal environment (within 10%). Further improving the speed probably requires at least making the core wider.

For a second group of 2 applications (*applu* and *mgrid*), the absolute performance is quite high (with IPCs around 3) for the baseline decoupled look-ahead system but there is still some performance headroom against the ideal environment. However, the speed of the look-ahead thread does not appear to be the problem as it is running fast enough on its own. There are a range of imperfections in the entire system that are the source of the performance gap. For example, the quality of information may not be high enough to have complete coverage of the misses. The look-ahead thread's L0 cache may hold stale data causing the thread to veer off the actual control flow causing recoveries.

For the remaining 14 applications, the application's speed is mainly limited by the speed of the look-ahead thread, indicating potential performance gain when the look-ahead thread is accelerated.



(a) Integer applications.



(b) Floating-point applications.

Figure 5.1: Comparison of performance of a baseline core, a decoupled look-ahead execution, and the two upper-bounds: a baseline with idealized branch prediction and memory hierarchy, and look-ahead thread running alone. Since the look-ahead thread has many NOPs that are removed at the pre-decode stage, their effective IPC can be higher than the pipeline width. Also note that because the baseline decoupled look-ahead execution can skip through some L2 misses at runtime, the speed of running look-ahead thread alone (but without skipping any L2 misses) is not a strict upper bound, but an approximation.

5.2.2 New Opportunities for Speculative Parallelization

There are two unique opportunities in the look-ahead environment to apply speculative parallelization. First, the construction of the skeleton removes some instructions from the original binary and thereby removes some dependences. Our manual inspection of a few benchmarks finds a repeating pattern of complex, loop-carried dependences naturally removed as the skeleton is being constructed (see example in Figure 5.2). This process gives rises to more loop-level parallelism.

A second opportunity is the lowered requirement for speculative parallelization. When two sequential code sections A and B are executed in parallel speculatively, register and memory dependences that flow from A to B are preserved by a combination of explicit synchronization and squash-and-re-execution of instructions detected to have violated the dependence. Because register dependences are explicit in the instruction stream, explicit synchronization can be used to enforce them. Nevertheless, the extra synchronization adds to the overhead of execution and demands special hardware support. For memory dependences, without complete static knowledge of the exact addresses, the compiler can not identify all possible dependences and runtime tracking becomes necessary.

In contrast, when we try to exploit speculative parallelization in the inherently non-critical look-ahead thread, correctness is no longer a must, but rather a quality of service issue. Fewer violations presumably lead to better accuracy in the code that reduces the chance for resynchronizing with the main thread and generates more useful prefetches. Thus, probabilistic analysis of the dependence is an acceptable approach for finding potential parallelism and generating multiple threads to exploit it.

Finally, as a byproduct, spawning a thread also helps in some cases of recovery. A recovery happens when the branch outcomes from the look-ahead threads deviate from that of the main thread. For simplicity, we reboot the look-ahead thread. Because the look-ahead threads can be running far ahead of the main thread, such a recovery can wipe out the lead the look-ahead thread accumulated over a period of time. Spawning a secondary thread provides a natural mechanism to preserve part of the work that has already been done as we can reboot one thread while keep the other running (shown in Figure 5.3). We will discuss this in more detail later.

```
   0x120021d00:    subl    a2,   0x2, a3
   0x120021d04:    ldq     a5,   208(sp)
   0x120021d10:    srl     a3,   t5,  a4
   0x120021d18:    and     a3,   t4,  a3
   0x120021d20:    and     a4,   0x7, a4
   0x120021d28:    addl    zero, a3,  a3
   0x120021d30:    mulq    a5,   a4,  a4
   0x120021d38:    ldq     ra,   152(sp)
❶  0x120021d3c:    lda     s5,   -1(s5)
   0x120021d48:    s4addq  a2,   ra,  t12
❷  0x120021d50:    addl    a2,   0x2, a2
   0x120021d58:    ldl     t12,  -8(t12)
   0x120021d60:    s8addq  a3,   a4,  a3
   0x120021d68:    addq    s1,   a3,  a3
   0x120021d6c:    srl     t12,  t5,  s2
❸  0x120021d78:    ldt     $f7,  0(a3)
❹  0x120021d7c:    ldt     $f12, 8(a3)
   0x120021d80:    and     s2,   0x7, s2
   0x120021d90:    mulq    a5,   s2,  s2
   0x120021d98:    and     t12,  t4,  t12
   0x120021da0:    addl    zero, t12, t12
   0x120021da8:    s8addq  t12,  s2,  s2
   0x120021db8:    addq    s0,   s2,  s2

        floating-point computation

❺  0x120021e48:    stt     $f15, 0(s2)
❻  0x120021e4c:    stt     $f21, 8(s2)
   0x120021e58:    bgt     s5,   0x120021d00
```

Figure 5.2: Example of removal of loop-carried dependences after constructing the skeleton for *lucas*. For clarity, only a portion of the original loop is shown. In this assembly format, the destination register is the last register in the instruction for ALU instructions, but the first for load instructions. In the skeleton version of this loop, instructions 3, 4, 5, 6 are converted to prefetches and floating-point computation is removed. Inspection of the loop shows that registers sp, t5, t4, s1, s0 are loop-invariant. Registers a5, a3, a4, ra, t12, s2, f7, f12, f15, f21 are local to this loop and can be derived from loop-invariants and register s5, a2. Registers s5 and a2 project a false loop-carried dependences on index variable (instruction 1 and 2). With instructions 3, 4, 5, 6 gone this code no longer has true loop-carried dependence.

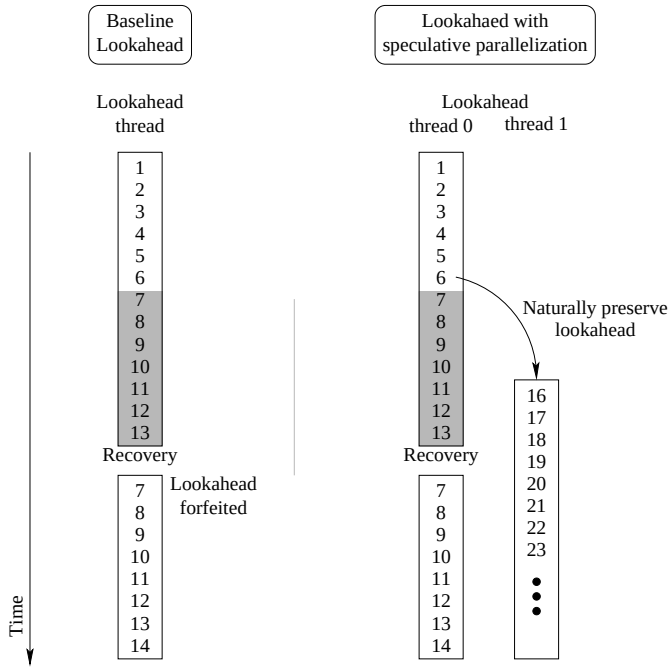


Figure 5.3: Illustration of how spawning a secondary thread naturally preserve lookahaed.

5.2.3 Software Support

Dependence analysis

To detect coarse-grain parallelism suitable for thread-level exploitation, we use a profile guided analysis tool. The look-ahead thread binary is first profiled to identify dependences and their distance. To simplify the subsequent analysis, we collapse the basic block into a single node, and represent the entire execution trace as a linear sequence of these nodes. Dependences are therefore between basic blocks and the distance can be measured by the distance of nodes in this linear sequence as shown in Figure 5.4-(a).

Given these nodes and arcs representing dependences between them, we can make a cut before each node and find the minimum dependence distance among all the arcs that pass through the cut. This minimum dependence distance, or D_{min}, represents an approximation of parallelization opportunity as can be explained by the simple example in Figure 5.4. Suppose, for the time being, that the execution of a basic block takes one unit of time and there is no overlapping of basic block execution. Node d in Figure 5.4, which has a D_{min} of 3, can therefore be scheduled to execute 2 units of time ($D_{min} - 1$) earlier than its current position in the trace

– in parallel with node b. All subsequent nodes can be scheduled 2 units of time earlier as well, without reverting the direction of any arcs.

Of course, the distance in basic blocks is only a very crude estimate of the actual time lapse between the execution of the producer and consumer instructions. [1] In reality, the size and execution speed of different basic blocks is different and their executions overlap. Furthermore, depending on the architectural detail, executing the producer instructions earlier than the consumer does not necessarily guarantee the result will be forwarded properly. Therefore, for nodes with small D_{min} there is little chance to exploit parallelism. We set a threshold on D_{min} (in Figure 5.4) to find all candidate locations for a spawned thread to start its execution.

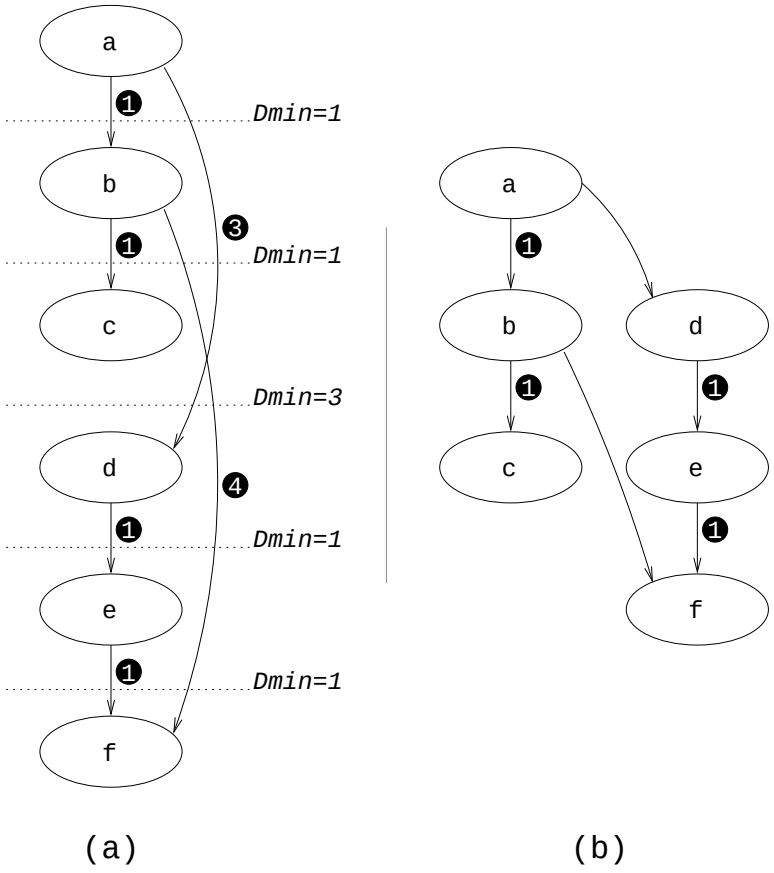


Figure 5.4: Example of basic-block level parallelization.

[1] A somewhat more appropriate notion of dependence distance that we use is the actual number of instructions in between the source and destination basic blocks.

Selecting spawn and target points

With the candidates selected from profile-based analysis, we need to find static code locations to spawn off parallel threads (*e.g.*, node a in Figure 5.4-b), and locations where the new threads can start their execution (*e.g.*, node d in Figure 5.4-b). We call the former *spawn points*, the latter *target points*. We first select target points. We choose those static basic blocks whose dynamic instances consistently show a D_{min} larger than the threshold value.

Next, we search for the corresponding spawn points. The choices are numerous. In the example shown in Figure 5.4, if we ignore the threshold of D_{min}, the spawn point of node d can be either a or b. Given the collection of many different instances of a static target point, the possibilities are even more numerous. The selection needs to balance cost and benefit. In general, the more often a spawn is successful and the longer the distance between the spawn and target points the more potential benefit there is. On the other hand, every spawn point will have some unsuccessful spawns, incurring costs. We use a cost benefit ratio ($\Sigma distances/\#false\ spawns$) to sort and select the spawn points. Note that a target point can have more than one spawn points. Figure 5.5 shows the pseudocode of algorithm used in the thesis.

Loop-level parallelism

Without special processing, a typical loop using index variable can project a false loop-carried dependence on the index variable and masks potential parallelism from our profiling mechanism. After proper adjustments, parallel loops will present a special case for our selection mechanism. The appropriate spawn and target points would be the same static node. The number of iterations to jump ahead is selected to make the number of instructions close to a target number (1000 in our setup).

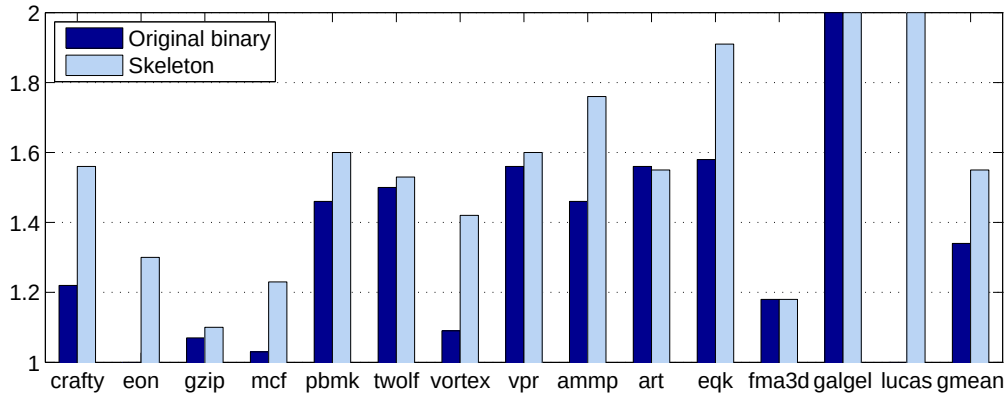
Available parallelism

A successful speculative parallelization system maximizes the opportunities to execute code in parallel (even when there are apparent dependences) and yet does not create too many squashes at runtime. In that regard, our methodology has a long way to go. Our goal in this chapter is to show that there are significant opportunities for speculative parallelization in the special

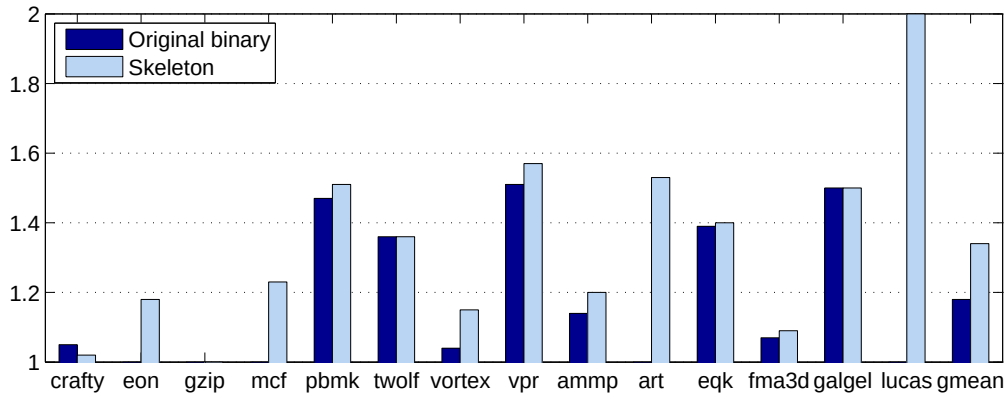
```
bbList = {Sequential list of basic blocks from a dynamic trace};
depArcs = {};
dminList = {};
spawnTargets = {};
spawnPointFreq = {};
spawnPair = {};

for depPair(a, b) in bbList:
   depArcs[a][b] = distance(a, b);

for node in bbList:
   arcList = arcsCrossing(node, depArcs);
   if dmin(arcList) > D_LOW and dmin(arcList) < D_HIGH:
      dminList[node] = dmin(listArcs);

// Create a list of potential spawn−target pairs
for (node, d) in dminList:
   target = staticId(node);
   spawn = staticId(node−d);

   spawnTargets[spawn][target]['totalDis'] += d;
   spawnTargets[spawn][target]['pairFreq'] += 1;
   spawnPointFreq[spawn] = 0;

// Compute dynamic count of each spawn point
for node in bbList:
   if staticId(node) in spawnPointFreq:
      spawnPointFreq[staticId(node)] += 1;

// Compute cost benefit ratio for each pair
for (spawn, target, totalDis, pairFreq) in spawnTargets:
   spawnTargets[spawn][target]['benefit'] = totalDis/(spawnPointFreq[spawn] − pairFreq);

// Sort spawn−target pairs using cost benefit ratio
spawnTargets = sortPairsMaxtoMin(spawnTargets, 'benefit');

//Select stable spawn−target pairs
for (spawn, target, benefit) in spawnTargets:
   if spawn not in spawnPair and benefit > B_MIN:
      spawnPair += (spawn, target);
```

Figure 5.5: Pseudocode of algorithm used to detect coarse-grain parallelism.

environment of look-ahead, even without a sophisticated analyzer. Figure 5.6 shows an approximate measure of available parallelism recognized by our analysis mechanism. The measure is simply that of the "height" of a basic block schedule as shown in Figure 5.4. For instance, the schedule in Figure 5.4-a has a height of 6 and the schedule in Figure 5.4-b has a height of 4, giving a parallelism of 1.5 (6/4). (Note that in reality, node d's D_{min} is too small to be hoisted up for parallel execution.) For simplicity, at most two nodes can be in the same time slot, giving a maximum parallelism of 2. For comparison, we also show the result of using the same mechanism to analyze the full program trace.



(a)



(b)

Figure 5.6: An approximate measure of available parallelism in the trace of the look-ahead thread and the main program thread (a). A more constrained measure of parallelism that is likely to be exploited (b).

The result shows that there is a significant amount of parallelism, even in integer code. Also, in general, there is more parallelism in the look-ahead thread than in the main program thread.

The most extreme example is *lucas* where the main program thread is completely serial (with a potential of 1) and the look-ahead thread is completely parallel (with a potential of 2).

Figure 5.6-(b) shows the amount of parallelism when we impose further constraints, namely finding stable spawn-target point pairs to minimize spurious spawns. The result suggests that some parallelism opportunities are harder to extract than others because there are not always spawn points that consistently lead to the execution of the target point. We leave it as future work to explore cost-effective solutions to this problem.

5.2.4 Hardware Support

Typical speculative parallelization requires a whole host of architectural support such as data versioning and cross-task dependence violation detection [RSC+06]. Since look-ahead does not require correctness guarantee, we are interested in exploring a design that minimizes intrusion.

To enable the speculative parallelization discussed so far, there are three essential elements that we need to (1) spawn a thread, (2) communicate values to the new thread, and (3) properly merge the threads.

Spawning a new thread

The support we need is not very different from existing implementation to spawn a new thread in a multi-threaded processor. The key difference is that the spawned thread is executing a future code segment in the same logical thread (as shown in Figure 5.7). If everything is successful, it is the primary look-ahead thread which did the spawning that will terminate. Therefore, the starting PC of the new thread needs to be recorded. When the primary thread reaches that same PC – more specifically, when the instruction under that PC is about to retire – the primary thread simply terminates, without retiring that instruction, since it has been executed by the spawned thread.

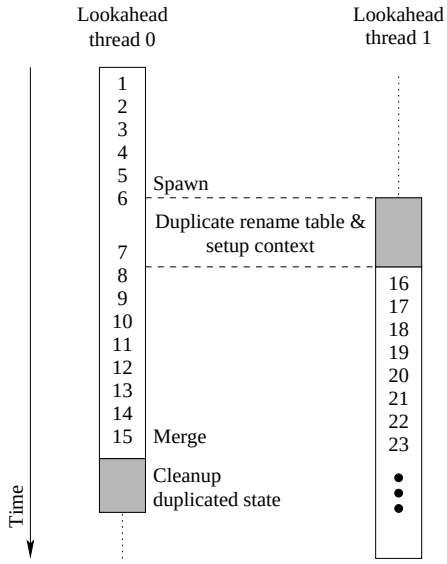


Figure 5.7: Illustration of runtime thread spawn and merge.

Support for register access

When a new thread is spawned, it inherits the architectural state including memory content and register state. While this can be implemented in a variety of environments, it is most straightforward to support in a multithreaded processor. We focus on this environment.

When the spawning instruction is dispatched, the register renaming map is duplicated for the spawned thread. With this action, the spawned thread is able to access register results defined by instructions in the primary thread prior to the spawn point. Note that if an instruction (with a destination register of say, $R6$) prior to the spawn point has yet to execute, the issue logic naturally guarantees that any subsequent instruction depending on $R6$ will receive the proper value regardless of which thread the instruction belongs to (*e.g.*, register r6 in Figure 5.8).

When a rename table entry is duplicated, a physical register is mapped in two entries and both entries can result in the release of the register in the future. A single bit per rename table entry is therefore added to track "ownership" ('O' bit in Figure 5.8). When the spawned thread copies the rename table, the ownership bits are set to 0. A subsequent instruction overwriting the entry will not release the old register, but will set the ownership bit to 1 (*e.g.*, in Figure 5.8, 'O' bit of register r6 in spawned thread's map table is set to 1 after renaming).

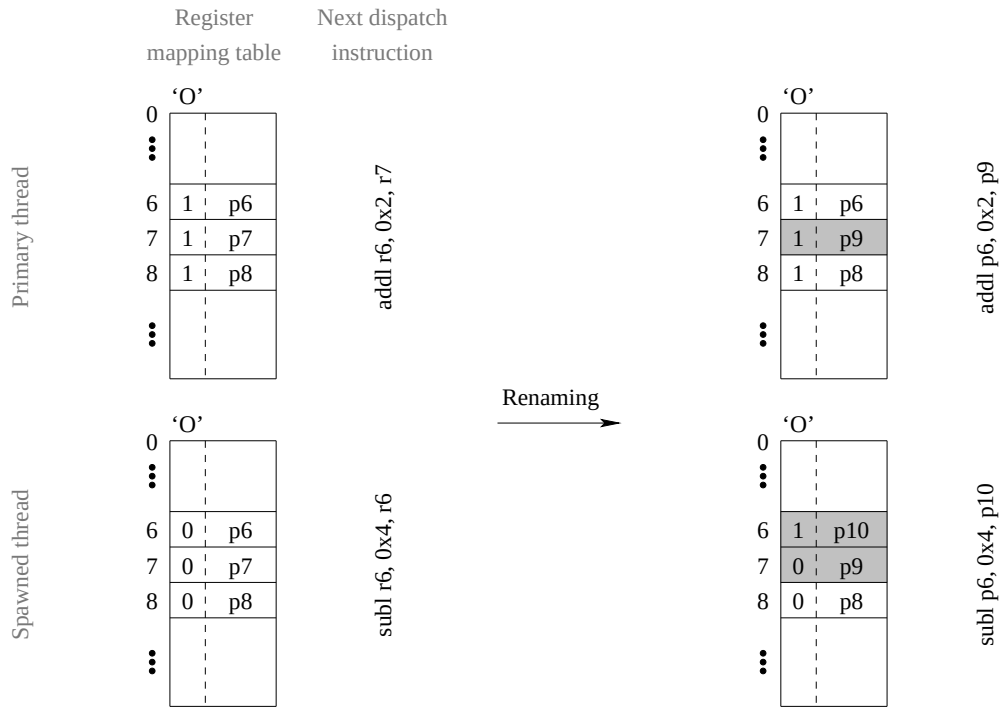


Figure 5.8: Register renaming support for the spawned thread. Entry 'O' in map tables refers to "ownership" bit.

Support for memory access

Getting memory content from the primary thread is also simplified in the multithreaded processor environment since the threads share the same L0 cache. An extreme option is to not differentiate between the primary look-ahead thread and the spawned thread in cache accesses. This option incurs the danger that write operations to the same memory location from different threads will not be differentiated and subsequent reads will get wrong values. However, even this most basic support is a possible option, though the performance benefit of parallel look-ahead is diminished as we will show later.

A more complete versioning support involves tagging each cache line with thread ID and returning the data with the most recent version for any request. For conventional speculative parallelization, this versioning support is usually done at a fine access granularity to reduce false dependence violation detections [GVSS98]. In our case, we use a simplified, coarse-grain versioning support without violation detection, which is a simple extension of the cache design in a basic multithreaded processor.

The main difference from a full-blown versioning support is two-fold. First, version is only attached to the cache line as in the baseline cache in a multi-threaded processor. No per-word tracking is done. Similar to versioning cache, a read from thread i returns the most recent version no later than i. A write from thread i creates a version i from the most recent version if none exists. The old version is tagged (by setting a bit) as being replaced by a new version. This bit is later used to gang-invalidate replaced lines. Second, no violation detection is done. When a write happens, it does not search future versions for premature reads. The cache therefore does not track whether any words in a line have been read.

5.2.5 Runtime Spawning Control

Based on the result of our analysis and to minimize unnecessary complexities, we opt to limit the number of threads spawned at any time to only one. This simplifies hardware control such as when to terminate a running thread and versioning support. It also simplifies the requirement on dependence analysis.

At runtime, two thread contexts are reserved (in a multithreaded core) for look-ahead. There is always a primary thread. A spawn instruction is handled at dispatch time and will freeze the pipeline front end until the rename table is duplicated and a new context is set up (as shown in Figure 5.7). If another thread is already occupying the context, the spawn instruction is discarded. Since the spawn happens at dispatch, it is a speculative action and is subject to a branch misprediction squash. We therefore do not start the execution immediately, but wait for a short period of time. This waiting also makes it more unlikely that a cross-thread read-after-write dependence is violated. When the spawn instruction is indeed squashed due to branch misprediction, we terminate the spawned thread.

When the primary thread reaches the point where the spawned thread started execution, the two successfully merge. The primary thread is terminated and the context is available for another spawn. The spawned thread essentially carries on as the primary look-ahead thread. At this point, we gang invalidate replaced cache lines from the old primary thread and consolidate the remaining lines into the new thread ID.

When the primary thread and the spawned thread deviate from each other, they may not

merge for a long time. When this happens, keeping the spawned thread will prevent new threads from being spawned and limit performance. So run-away spawns are terminated after a fixed number of cycles (computed from the length of biggest parallelizable code section for an application).

5.2.6 Communicating Branch Predictions to the Primary Thread

Branch predictions produced by look-ahead thread(s) are deposited in an ordered queue called branch queue. There are many options to enforce semantic sequential ordering among branch predictions even though that they might be produced in different order. One of the simpler options we explored in this chapter is to segment the branch queue in a few banks of equal size as shown in Figure 5.9. The bank management is straightforward. When a thread is spawned a new bank is assigned in the sequential order. A bank is also de-allocated in the sequential order as soon as the primary thread consumes all branch predictions form that bank. If the youngest bank currently in use is full, the next sequential bank, if available, is assigned to the look-ahead thread. This is, however, not possible when the older bank is full and the thread using it have not yet merged. Our results show that very rarely an older bank fills up. So for these rare cases, to avoid deadlock and yet manage sequential ordering, we can afford to kill the younger thread and reclaim bank for re-assignment.

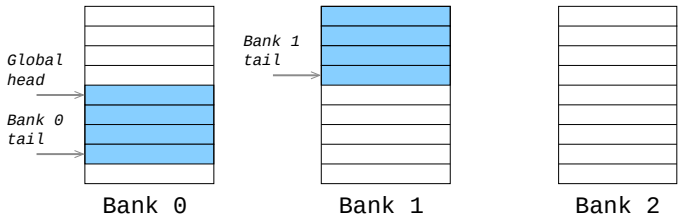


Figure 5.9: Example of a banked branch queue. Bank 0 and bank 1 are written by primary (older) look-ahead and spawned (younger) look-ahead threads respectively. Primary thread uses global head pointer, currently pointing to an entry in bank 0, to read branch predictions.

5.3 Experimental Setup

We build hardware support for speculative parallelization on top of modified version of SimpleScalar [BA97] presented in Section 4.4. Lookahead core was augmented to support Simultaneous Multi-Threading (SMT) and speculative parallelization.

For simulations we use SPEC CPU2000 benchmarks compiled for Alpha. We use the *train* input for profiling, and run the applications to completion. For evaluation, we use *ref* input. We simulate 100 million instructions after skipping over the initialization portion as indicated in [SC00].

5.4 Experimental Analysis

We first look at the end result of using our speculative parallelization mechanism and then show some additional experiments that shed light on how the system works and point to future work to improve the design's efficacy.

	bzip2	crafty	eon	gap	gcc	gzip	mcf	perlbmk	twolf	vortex	vpr
1	1.32	2.30	2.63	1.92	2.20	2.14	0.51	0.89	0.57	1.93	1.31
2	1.75	2.47	2.90	3.35	4.60	2.34	0.83	1.14	0.74	2.24	1.77
3	1.75	2.48	2.91	3.36	4.81	2.36	0.84	1.35	1.01	2.27	2.43

(a) Integer applications.

amm	app	apsi	art	eqk	fac	fma	gal	luc	mesa	mgr	six	swim	wup
0.79	1.81	1.75	0.27	1.09	3.03	2.72	2.35	0.58	2.99	3.03	2.84	1.26	3.77
1.92	2.76	2.34	1.13	2.73	3.65	3.12	3.79	1.75	3.36	3.83	3.12	3.78	4.11
1.93	2.75	2.49	1.57	2.85	3.68	3.12	4.17	2.44	3.37	3.83	3.12	3.78	4.11

(b) Floating-point applications.

Table 5.1: IPC of baseline (1), baseline look-ahead (2), and speculatively parallel look-ahead (3).

5.4.1 Performance analysis

Recall that a baseline look-ahead system can already help many applications to execute at high throughput that is close to saturating the baseline out-of-order engine. For these appli-

cations, the bottleneck is not the look-ahead mechanism. Designing efficient wide-issue execution engine or look-ahead to assist speculative parallelization of the main thread are directions to further improve their performance. For the remaining applications, Figure 5.10 shows the relative performance of a baseline, single-threaded look-ahead system and our speculative, dual-threaded look-ahead system. All results are normalized to that of the single-core baseline system.

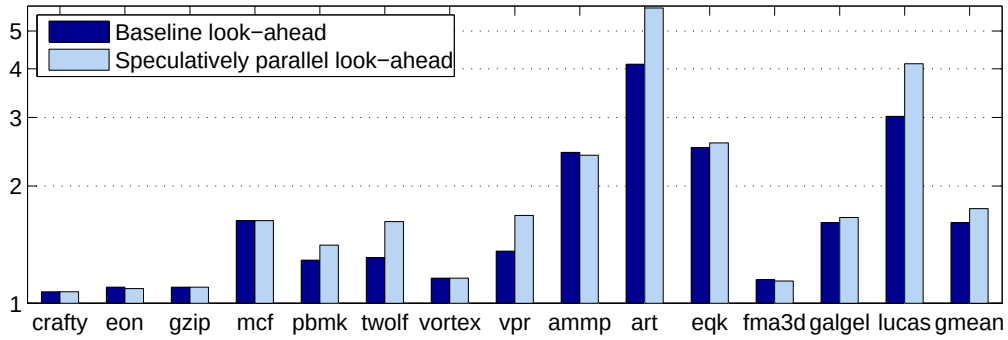


Figure 5.10: Speedup of baseline look-ahead and speculatively parallel look-ahead over single-core baseline architecture. Because the uneven speedups, the vertical axis is log-scale.

Our speculative parallelization strategy provides up to 1.39x speedup over baseline look-ahead. On average, measured against the baseline look-ahead system, the contribution of speculative parallelization is a speedup of 1.13. As a result, the speedup of look-ahead over a single core improves from 1.61 for single look-ahead thread to 1.75 for two look-ahead threads. If we only look at the integer benchmarks in this set of applications, traditionally considered harder to parallelize, the speedup over baseline look-ahead system is 1.11.

It is worth noting that the quantitative results here represent what our current system allows us to achieve. It does not represent what could be achieved. With more refinement and trial-and-error, we believe more opportunities can be explored. Even with these current results, it is clear that speeding up sequential code sections via decoupled look-ahead is a viable approach for many applications.

Finally, for those applications where the main thread has (nearly) saturated the pipeline, this mechanism does not slow down the program execution. The detailed IPC results for all applications are shown in Table 5.1.

5.4.2 Comparison with conventional speculative parallelization

As discussed earlier, the look-ahead environment offers a unique opportunity to apply speculative parallelization technology partly because the code to drive look-ahead activities removes certain instructions and therefore provide more potential for parallelism. However, an improvement in the speed of the look-ahead only indirectly translates into end performance. Here, we perform some experiments to inspect the impact.

We use the same methodology on the original program binary and support the speculative threads to execute on a multi-core system. Again, our methodology needs further improvement to fully exploit available parallelism. Thus the absolute performance results are almost certainly underestimating the real potential. However, the relative comparison can still serve to contrast the difference in the two setups.

Figure 5.11 shows the results. For a more relevant comparison, conventional speculative parallelization is executed on two cores to prevent execution resource from becoming the bottleneck. It is worth nothing, the base of normalization is not the same. For the conventional system, the speedup is over a single core running the program. For our system, the speedup is over a baseline look-ahead thread, which has much higher performance.

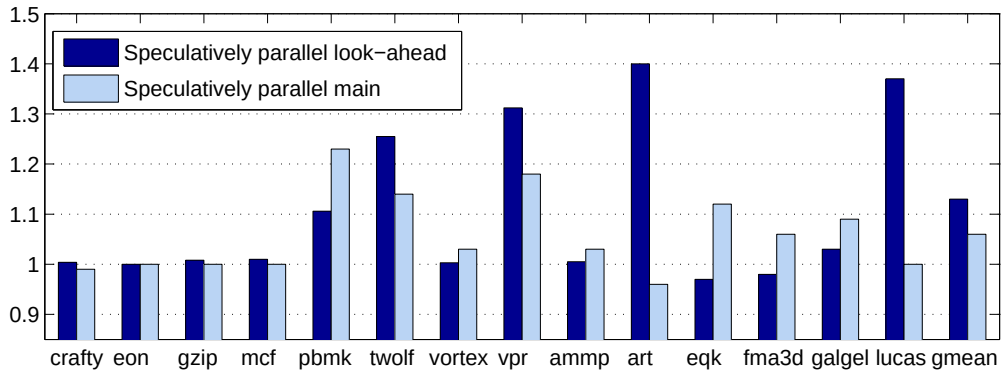


Figure 5.11: Comparison of the effect of our speculative parallelization mechanism on the look-ahead thread and on the main thread.

As we showed earlier using a simple model of potential parallelism (Figure 5.6), there is more parallelism in the look-ahead binary (the skeleton) than in the full program binary. In Figure 5.11, we see that in many cases, this translates into more performance gain in the end for the look-ahead system. However, there are phases of execution where the look-ahead speed is

not the bottleneck. A faster look-ahead thread only leads to filling the queues faster. Once these queues that pass information to the main thread fill up, look-ahead stalls. Therefore, in some cases, the advantage in more potential parallelism does not translate into more performance gain.

5.4.3 System Diagnosis

Recoveries

When the look-ahead thread's control flow deviates from that of the main thread, the difference in branch outcome eventually causes a recovery. The ability of the look-ahead thread to run far ahead of the main thread is crucial to performing useful help. This ability is a direct result of ignoring uncommon cases and other approximations, which come at the price of recoveries. Speculative parallelization in the look-ahead environment also introduces its own sets of approximations. Otherwise, the opportunities will be insufficient and the implementation barrier will be too high. However, too much corner-cutting can be counter-productive. Table 5.2 summarizes the maximum, average, and minimum recovery rate for integer and floating-point applications.

	INT			FP		
	Max	Avg.	Min	Max	Avg.	Min
Baseline look-ahead	3.21	1.24	0.05	2.87	0.34	0.00
Spec. parallel look-ahead	6.55	1.21	0.05	2.83	0.34	0.00

Table 5.2: Recovery rate for baseline and speculatively parallel look-ahead systems. The rates are measured by the number of recoveries per 10,000 committed instructions in the main thread.

For most applications, the recovery rate stays essentially the same. For some applications the rate actually reduces (*e.g.*, *perlbmk*, from 3.21 to 0.43). Recall that skipping an L2 miss (by returning a 0 to the load instruction) is an effective approach to help the look-ahead thread stay ahead of the main thread. Frequent applications of this technique inevitably increase recoveries. In our system, this technique is only applied when the trailing main thread gets too close. With speculative parallelization, the need to use this technique decreases, as does the number of resulting recoveries.

Partial recoveries

As discussed earlier, when a recovery happens, we reboot the main look-ahead thread, but if there is another look-ahead thread spawned, we do not terminate the spawned thread, even though the recovery indicates that some state in the look-ahead threads is corrupted. We have this option because the look-ahead activities are not correctness critical. This essentially allows a partial recovery (without any extra hardware support) and maintains the lead of look-ahead. Nevertheless, it is possible that the spawned thread is corrupt and this policy only delays the inevitable. Table 5.3 shows that this is not the case. The first row of numbers indicate how often a spawned thread successfully merged with the rebooted main look-ahead thread, indicating the spawned thread is still on the right path. The next two rows show how many of these cases the spawned thread is still live (has not encountered its recovery) after 200 and 1000 instructions. In many applications, almost all instances of the spawned thread are alive and well pass 1000 instructions, indicating that indeed they deserved to be kept alive at the recovey point. Also, the difference between the number of instances alive at 200 and 1000 instructions point is very small, indicating that those that do not survive long actually terminate rather early. All in all, it is clear that keeping these spawned threads live has low risks and can achieve up to 3% performance improvement.

	gap	mcf	pbm	twf	vor	vpr	ammp	eqk	fac	fma	gal
Recv-Merge	1116	1309	6408	57910	4862	12314	4252	9174	925	4062	291
Live 200	1031	1308	3642	39305	4801	11284	2970	4347	526	4055	290
Live 1000	917	1306	3355	17601	3774	10178	2626	3495	522	4041	290

Table 5.3: Partial recoveries in speculative look-ahead. Recv-Merge are the instances when a recovery happens in the main look-ahead thread and spawned thread merges later. Live 200 and Live 1000 are the instances when a Recv-Merge occurced and merged thread didn't experience a new recovery for at least 200 and 1000 instrctions respectively.

Spawns

Table 5.4 shows statistics about the number of spawns in different categories. The top half shows the cases where a spawn happens on the right path. They are predominantly successful. Only a handful of spawns had to be killed because the main thread has not merged with the

spawned thread after a long time. The bottom half shows spawns on the wrong path due to branch mispredictions or because the primary look-ahead thread veers off the right control flow.

	mcf	pbm	twf	vor	vpr	ammp	art	eqk	fma	gal	lucas
Spawns invoked under correct path											
Successful	2297	26873	21067	1273	42082	6328	29598	16676	9687	20997	24022
Runaway	257	245	1738	37	409	3542	363	0	3965	0	1
Spawns invoked under incorrect path											
No disp.	11	707	2837	96	1633	26	29	245	363	1	0
Few disp.	28	69	1803	6	273	45	116	10	1	0	0
WP	11	184	2997	152	111	339	6	62	4	17	0

Table 5.4: Breakdown of all the spawns. WP refers to the case the spawn happens when the primary look-ahead thread has veered off the right control flow and will eventually encounter a recovery. For crafty, eon, and gzip we do not see any spawns in our experiments.

We can see that most of these spurious spawns happen because of branch misprediction that would be subsequently corrected. Recall that the spawned thread does not execute immediately after the spawn, but wait for a small period of time (to minimize unnecessary execution due to branch misprediction-triggered spawns and also to reduce violation of dependence). As a result of this small delay, many spawned threads have not dispatched any instruction before the branch is resolved and the spawn squashed as well. Almost all of these spurious spawns are short lived even for those cases where some instructions on the spawned thread have been dispatched. In summary, speculative parallelization does not significantly increase the energy cost as the waste is small. Our speculatively parallel look-ahead executes on average 1.5% more instructions than sequential lookahead due to very few failed spawns.

Effect of speculation support

Because look-ahead thread is not critical for correctness, supporting speculative parallelization can be a lot less demanding than otherwise – in theory. In practice, there is no appeal for complexity reduction if it brings disproportionate performance loss. Section 5.2.4 described a design that does not require full-blown versioning and has no dependence violation tracking. The cache support required is a much more modest extension of cache for multithreaded core. In Figure 5.12, we compare this design to one that is even more relaxed: the data cache has

no versioning support and in fact is completely unaware of the distinction between the primary look-ahead thread and the spawned one. As the figure shows, two applications (*vpr* and *equake*) suffer significant performance losses. However, for all other applications the degradation is insignificant. Since L1 caches are critical for performance and is often the subject of intense circuit timing optimization, any significant complication can create practical issues in a high-end product. Speculative parallelization in look-ahead, however, gives the designers the capability to choose incremental complexity with different performance benefits, rather than an all-or-nothing option as in conventional speculative parallelization.

Another example of the design flexibility is about whether to detect dependence violations. Dependence violation detection also requires intrusive modifications. The flexibility of not having to support it is thus valuable. Figure 5.12 also compares our design to another one where accesses are carefully tracked and when a dependence violation happens, the spawned thread is squashed. This policy provides no benefit in any application window we simulated and degraded performance significantly in one case (*vortex*). Intuitively, the look-ahead process is somewhat error tolerant. Being optimistic and ignore the occasional errors is, on the balance, a good approach.

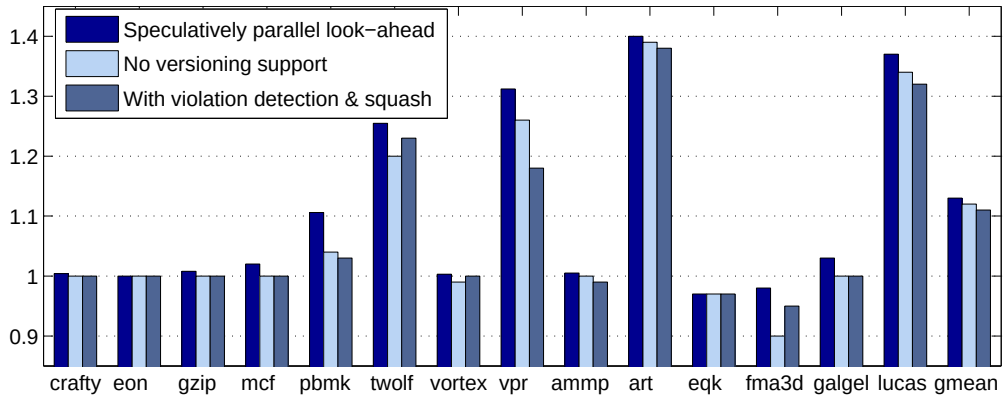


Figure 5.12: Speedup comparison of regular support and two other alternatives, one removing the partial versioning support altogether, the other adding dependence violation detection to squash the spawned thread.

5.5 Recap

In this chapter, we have proposed a mechanism to apply speculative parallelization to the look-ahead thread. This approach is motivated by two intuitions: 1. Look-ahead code contains fewer dependences and thus lends itself to (speculative) parallelization; and 2. Without correctness constraints, hardware support for speculative parallelization of the look-ahead thread can be much less demanding. We have presented a software mechanism to probabilistically extract parallelism and shown that indeed the look-ahead code affords more opportunities. We have also presented a hardware design that does not contain the array of support needed for conventional speculative parallelization such as dependence tracking and complex versioning. For an array of 14 applications where the speed of the look-ahead thread is the bottleneck, the proposed mechanism speeds up the baseline, single-threaded look-ahead system by up to 1.39x with a mean of 1.13x. Experimental data also suggest there is further performance potential to be extracted which would be investigated as future work.

Chapter 6

Summary and Future Work

In this chapter, I summarize the contributions of the dissertation and present directions for future work.

6.1 Summary

Improving single-thread performance for general-purpose code continues to be an important goal in the design of microprocessors. While designs often target the common case for optimization, they have to be correct under all cases. Consequently, while there are ample opportunities for performance optimization and novel techniques are constantly being invented, their practical application in real product designs faces ever higher barriers and costs, and diminishing effectiveness. Correctness concern, especially in thorny corner cases, can significantly increase design complexity and dominate verification efforts. The reality of microprocessor complexity, its design effort, and costs [BMMR05] reduces the appeal of otherwise sound ideas, limits our choice, and forces suboptimal compromises. In this thesis, we have introduced performance-correctness *explicitly-decoupled architecture* with two separate domains, each focuses only on one goal, performance optimization or correctness guarantee. By explicitly separating the two goals, both can be achieved more efficiently with less complexity.

Given the correctness guarantee, the performance domain can truly focus on the common case and allow the entire design stack to carry out optimizations in a very optimistic and efficient

manner, avoiding excessive conservatism. An effective performance domain allows designers to use simpler, throughput-oriented designs for the correctness domain and focus on other practical considerations such as system integrity. As a proof-of-concept, in this thesis we have demonstrated two concrete designs.

6.1.1 Decoupled Memory Dependence Enforcement

First, we presented a decoupled memory dependence enforcement approach aimed at simplifying one of the most complex and non-scalable functional blocks in modern high-performance out-of-order processors. The conventional approach uses age based queue to buffer and cross-compare memory operations to ensure that the out-of-order execution of memory instructions does not violate program semantics. In the design presented in Chapter 3, we move away from this conventional load-store handling mechanism that strives to perform most accurate forwarding in the first place and proactively detect and recover from any dependence violation. Instead, we adopt a very decoupled approach where memory instructions execute twice, first in a front-end execution where they execute only according to register dependences and access an L0 cache to perform an opportunistic communication. They then access memory a second time, in program order, accessing the non-speculative L1 cache to detect mis-speculations and recover from it.

We have shown that even a rudimentary implementation of this decoupled approach rivals a conventional disambiguation logic with optimistically sized load queue and store queue. When two cost-effective optimization techniques are employed, the improved design offers performance close to that of a conventional system with ideal load-store queue. Another advantage of the design is that when scaling up the in-flight instruction capacity, almost no change is needed and yet the performance improves significantly. With a fewer concerns for correctness in the front-end, we have demonstrated a simpler yet effective implementation of an optimization to delay premature loads from execution. In the rest of this thesis we tried to demonstrate many such cost-effective optimizations in the performance domain.

6.1.2 Decoupled Lookahead

Next, we widened our scope in an attempt to explore explicitly decoupled architecture. In Chapter 4, we presented ways explicitly decoupled architecture can improve upon traditional ILP lookahead. While lookahead techniques have the potential to uncover significant amount of ILP, conventional microarchitectures impose practical limitations on its effectiveness due to their monolithic implementation. Correctness requirement limits the design freedom to explore probabilistic mechanisms and makes conventional lookahead resource-intensive: registers and various queue entries need to be reserved for every in-flight instruction, making deep lookahead very expensive to support.

We elevated these challenges by separating lookahead engine from architectural execution using two independent cores: the optimistic and the correctness core. Executing dedicated code to run ahead of program execution to help mitigate performance bottlenecks from branch mispredictions and cache misses is a promising approach. We showed that both the microarchitecture of the optimistic core and its software can be designed optimistically and allow much simpler methods for optimizations. Such a design enables efficient, deep lookahead and produces a significant performance boosting effect. It also allows the correctness core to be less aggressive in its implementation with less impacts on performance than such simplifications would have on a conventional microarchitecture.

This approach, in some cases, allows the program to nearly saturate a high-performance out-of-order pipeline. In other cases, the speed of the look-ahead thread becomes the bottleneck. To further enhance the performance of the look-ahead thread, we also explored cost-effective and efficient speculative parallelization in performance domain.

6.1.3 Speculative Parallelization in Decoupled Lookahead

In Chapter 5, we proposed a mechanism to apply speculative parallelization to the look-ahead thread. This approach is motivated by two intuitions: 1. Look-ahead code contains fewer dependences and thus lends itself to (speculative) parallelization; and 2. Without correctness constraints, hardware support for speculative parallelization of the look-ahead thread can be much less demanding. We have presented a software mechanism to probabilistically extract

parallelism and shown that indeed the look-ahead code affords more opportunities. We have also presented a hardware design that does not contain the array of support needed for conventional speculative parallelization such as dependence tracking and complex versioning. For an array of 14 applications where the speed of the look-ahead thread is the bottleneck, the proposed mechanism speeds up the baseline, single-threaded look-ahead system by 1.14x (geometric mean). The speedup for the 8 integer applications is 1.16. The experimental data also suggest there is further performance potential to be extracted.

In the end, we have demonstrated with two concrete designs that by explicitly decoupling performance from correctness, we can design both the domains more efficiently. This relatively simpler implementation of both the domains is also more effective in improving overall system performance. While, we have proposed many such optimizations in this thesis, there are a lot more remains to be explored. We will look at few such opportunities in the next section.

6.2 Future Work

We summarize future work on further opportunities in performance and correctness domain, and new opportunities to solve existing problems.

6.2.1 Performance Domain Optimizations

With no requirement to keep the optimistic core absolutely correct, it is possible to improve its performance in many ways, and we have already explored a few of them in this thesis.

Frequency improvements

A straightforward technique to boost optimistic core's performance is to improve its operating frequency. In Chapter 4, we discussed how conservative timing margins are built into circuit to guard against extreme fabrication, PVT, and environmental variations. However, errors due to these variations are far from common and, in-fact, our studies in Section 4.5 shows that the optimistic core is immune to some of these type of errors. Further studies are needed to understand the impact to the performance domain and frequency gains expected when these

timing margins are relaxed. Infact, some of the critical paths in the design can be studied and probabilistic architectural solutions can be explored to further prune them for overall frequency gains.

Control flow independence

A main difference between a conventional thread and our skeleton is the output. Skeleton only produces a stream of branch outcomes. This has much less information content and therefore skeleton execution is naturally more tolerant to mis-speculation. And even these outcomes need not always be right. Therefore, instead of always relying on full-blown squash-and-recover methods used in existing designs, we can employ light-weight partial repair mechanisms when appropriate. Since any serious impact can be eventually repaired via a recovery from the correctness engine, a common-cases repair mechanism does not need to be always thorough. In certain cases, it may be more efficient to simply fix the recorded branch outcome (and ignore any potential impact on data flow) then to throw away work already done and rewind. Of course, since the skeleton also powers other optimizations (directly or indirectly), partial repair will affect the quality of service (QoS) of other optimizations.

A quantitative insights on the impact of mis-speculations and the effect of partial repairment needs to be developed. For instance, if a memory dependence is incorrectly enforced. Upon detection, how much state can be repaired by simply re-executing the load and its dependence chain? In general, how often can partial repairment work; whether we can use static analysis-based predictor or runtime predictor to judiciously decide between full and partial repairment; and whether we can develop simple heuristics to guide how far partial repairment will need to go.

Support for parallel applications

In Section 4.5, we performed a limited study on highly-tuned scientific SPLASH applications by ignoring any shared memory issues in the design of the optimistic core and showed that exploiting instruction level parallelism is as important for performance improvement as exploiting thread level parallelism. Further studies can be performed to understand some of the issues

related to parallel applications. *e.g.*, , how sharing issues impact the progress of the optimistic core and quality of predictions sent to the correctness core; can optimistic core take advantage of the lookahead to understand the sharing pattern. These insights can be used to mitigate the network latencies by pre-scheduling the communication of the shared cache lines. Lastly, an efficient implementation of explicitly decoupled architecture paradigm may need changes in the cache coherence protocol, *e.g.*, the optimistic core can perform instantaneous invalidation in L0 cache without waiting for an acknowledgment. Further, it can pass these invalidation hints to the correctness core. These hints can be used to pro-actively issue invalidations ahead of time.

6.2.2 Correctness Core Simplifications

In Chapter 4.5, we have discussed a few opportunities to simplify the correctness core. With a significantly fewer number of long latency operations performed by the correctness core, it fundamentally changes the way we may design its out-order execution core. For instance, it may not have to look for instruction level parallelism very aggressively. This change in behavior may simplify the design of otherwise complex circuit (*e.g.*, dispatch logic, load-store queue, issue, and broadcast logic) with little or no performance cost. We show that as long as the throughput is enough, proposed architecture is not as sensitive to the proposed simplifications as the baseline core. This study is, however, far from over and there is a potential for many more opportunities in the correctness core.

6.2.3 Opportunities to Solve Other Existing Problems

In this thesis, using explicitly decoupled architecture we explored solutions to two very important architectural problems. Many other opportunities exists in all the layers of the design stack, including software and circuits. Here we discuss another such opportunity in the circuit layer as a motivation for future work.

Inductive noise

Continuous improvements in the CMOS process technology to achieve higher levels of performance and circuit integration uncover many challenging circuit issues that become dominant

at a smaller feature size. For example, inductive noise in the power supply grid is one such issue. Reduction in feature size requires reduction in the operating voltage to control leakage current, effectively reducing the absolute noise margins. Noise margins are used to differentiate between logic levels for proper operation. At the same time, the demand for supply current either rises or remains the same due to higher circuit integration. Coupled with the use of low-power techniques such as clock gating, chip power supply sees a larger variation in demand for cycle-to-cycle current. These large variations in the processor current (di/dt) convert into large supply-voltage glitches due to inductive impedance present in the power-supply network. These supply-voltage glitches together with reduced noise margins is a big reliability concern in the modern high-performance microprocessors. Previous works try to solve this problem by the extensive use of decoupling capacitors [PPW02]. The decoupling capacitors try to reduce the inductive impedance, shielding the power-supply network from large variations in the current by acting as a current reservoir [PPWT99; GAT02]. Although this technique reduces the cycle-to-cycle inductive noise, still the supply network impedance changes with resonant frequency and the inductive impedance may be very high at the peaks which occur at resonant frequency. [PV03; JBM03; PV04; EEA04] uses architectural techniques to throttle the processor activity and bound the peak change in current around the resonant frequency.

Using a hierarchy of decoupled capacitors is effective in maintaining low impedance over a wide range of frequencies, but does not guarantee violation-free operation. Controlling a processor's activity, on the other hand, involves performance and energy trade-off with noise reduction. An explicitly decoupled architecture could be capable of handling the inductive noise problem in a better way. First, the optimistic core is immune to failures due to voltage under-shoots and over-shoots. Second, the correctness core could be more efficiently throttled with a much smaller energy and performance trade-off compared to the present schemes. Detailed studies are needed to understand the effect of inductive noise on explicitly decoupled architecture.

Bibliography

[AD98] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. Int'l Symp. on Microarch.*, pages 226–236, November–December 1998.

[AMW+07] M. Agarwal, K. Malik, K. Woley, S. Stone, and M. Frank. Exploiting Postdominance for Speculative Parallelization. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 295–305, February 2007.

[APD01] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proc. Int'l Symp. on Comp. Arch.*, pages 52–61, June 2001.

[ARS03] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. Int'l Symp. on Microarch.*, pages 423–434, December 2003.

[Aus99] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. Int'l Symp. on Microarch.*, pages 196–207, November 1999.

[BA97] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[BBH+04] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel™Pentium™4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1):1–17, February 2004.

[BC91] J. L. Baer and T. F. Chen. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proc. Int'l Conf. on Supercomputing*, pages 176–186, November 1991.

[BCGM07] M. Bohr, R. Chau, T. Ghani, and K. Mistry. The High-k Solution. *IEEE Spectrum*, 44(10):29–35, October 2007.

[BDA01a] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. In *Proc. Int'l Symp. on Comp. Arch.*, pages 26–37, June 2001.

[BDA01b] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *Proc. Int'l Symp. on Microarch.*, pages 237–248, December 2001.

[BIWB02] E. Brekelbaum, J. Rupley II, C. Wilkerson, and B. Black. Hierarchical Scheduling Windows. In *Proc. Int'l Symp. on Microarch.*, pages 27–36, November 2002.

[BKN+03] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. In *Proc. Design Automation Conf.*, pages 338–342, June 2003.

[BMMR05] C. Bazeghi, F. Mesa-Martinez, and J. Renau. μComplexity: Estimating Processor Design Effort. In *Proc. Int'l Symp. on Microarch.*, pages 209–218, December 2005.

[BNS+03a] R. Barnes, E. Nystrom, J. Sias, S. Patel N. Navarro, and W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proc. Int'l Symp. on Microarch.*, pages 387–399, December 2003.

[BNS+03b] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu. Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining. In *Proc. Int'l Symp. on Microarch.*, pages 387–398, December 2003.

[BS02] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 199–210, October 2002.

[BS03] S. Balakrishnan and G. Sohi. Exploiting Value Locality in Physical Register Files. In *Proc. Int'l Symp. on Microarch.*, pages 265–276, December 2003.

[BS04] J. Butts and G. Sohi. Use-Based Register Caching with Decoupled Indexing. In *Proc. Int'l Symp. on Comp. Arch.*, pages 302–313, June 2004.

[BS06] S. Balakrishnan and G. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. In *Proc. Int'l Symp. on Comp. Arch.*, pages 302–313, June 2006.

[BSI99] BSIM Design Group, `http://www-device.eecs.berkeley.edu/~bsim3/ftv322/Mod_doc/V322manu.tar.Z`. *BSIM3v3.2.2 MOSFET Model - User's Manual*, April 1999.

[BSP01] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *Proc. Int'l Symp. on Microarch.*, pages 204–213, December 2001.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. Int'l Symp. on Comp. Arch.*, pages 83–94, June 2000.

[BZ04] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. In *Proc. Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, October 2004.

[CB95] T. Chen and J. L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Theoretical Computer Science*, 44(5):609–623, May 1995.

[CCE+09] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor. In *Proc. Int'l Symp. on Comp. Arch.*, pages 484–295, June 2009.

[CCYT05] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, May/June 2005.

[CGVT00] J. Cruz, A. González, M. Valero, and N. Topham. Multiple-Banked Register File Architectures. In *Proc. Int'l Symp. on Comp. Arch.*, pages 316–325, June 2000.

[Che98] T. Chen. Supporting Highly Speculative Execution via Adaptive Branch Trees. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 185–194, January–February 1998.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 40–52, April 1991.

[CL04] H. Cain and M. Lipasti. Memory Ordering: A Value-based Approach. In *Proc. Int'l Symp. on Comp. Arch.*, pages 90–101, June 2004.

[CMCWm91] W. Chen, S. Mahlke, P. Chang, and W. Wen-mei. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proc. Int'l Symp. on Microarch.*, pages 69–73, November–December 1991.

[CMT00] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 13–24, June 2000.

[CNG+09] F. Castro, R. Noor, A. Garg, D. Chaver, M. Huang, L. Pinuel, M. Prieto, and F. Tirado. Replacing Associative Load Queues: A Timing-Centric Approach. *IEEE Transactions on Computers*, 58(4):496–511, April 2009.

[COLV04] A. Cristal, D. Ortega, J Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 48–59, February 2004.

[Com00] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, September 2000. Order number: DS-0027B-TE.

[CSCT02] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer cache assisted prefetching. In *Proc. Int'l Symp. on Microarch.*, pages 62–73, November 2002.

[CSK+99] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. Int'l Symp. on Comp. Arch.*, pages 186–195, May 1999.

[CST+04] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction. *IEEE TCCA Computer Architecture Letters*, 3, December 2004.

[CTWS01] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proc. Int'l Symp. on Microarch.*, pages 306–317, December 2001.

[CTYP02a] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *Proc. Int'l Symp. on Comp. Arch.*, pages 307–317, May 2002.

[CTYP02b] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. Microarchitectural Support for Precomputation Microthreads. In *Proc. Int'l Symp. on Microarch.*, pages 74–84, November 2002.

[CWT+01] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. Int'l Symp. on Comp. Arch.*, pages 14–25, June 2001.

[CYFM04] C. Chen, S. Yang, B. Falsafi, and A Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 276–287, February 2004.

[DM97] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. Int'l Conf. on Supercomputing*, pages 68–75, July 1997.

[DS98] M. Dubois and Y. Song. Assisted execution. Technical Report, Department of Electrical Engineering, University of Southern California, 1998.

[EEA04] W. El-Essawy and D. Albonesi. Mitigating Inductive Noise in SMT Processors. In *Proc. Int'l Symp. on Low-Power Electronics and Design*, pages 332–337, August 2004.

[FTEJ98] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. Int'l Symp. on Microarch.*, pages 59–68, November–December 1998.

[GAR+05] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 446–457, June 2005.

[GAT02] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural Simulation and Control of di/dt-induced Power Supply Voltage Variation. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 7–16, February 2002.

[GB05a] I. Ganusov and M. Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 350–360, September 2005.

[GB05b] I. Ganusov and M. Burtscher. On the Importance of Optimizing the Configuration of Stream Prefetchers. In *Proceedings of the 2005 Workshop on Memory System Performance*, pages 54–61, June 2005.

[GB06] I. Ganusov and M. Burtscher. Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 144–153, September 2006.

[GCH+06] A. Garg, F. Castro, M. Huang, L. Pinuel, D. Chaver, and M. Prieto. Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors. In *Proc. Int'l Symp. on Low-Power Electronics and Design*, pages 268–273, October 2006.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. Int'l Conf. on Parallel Processing*, pages I355–I364, August 1991.

[GGV98] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-Physical Registers. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 175–184, January–February 1998.

[GH08] A. Garg and M. Huang. A Performance-Correctness Explicitly Decoupled Architecture. In *Proc. Int'l Symp. on Microarch.*, November 2008.

[GJT+07] V. George, S. Jahagirdar, C. Tong, K. Smits, S. Damaraju, S. Siers, V. Naydenov, T. Khondker, S. Sarkar, and P. Singh. Penryn: 45-nm Next Generation Intel Core 2 Processor. In *Proc. IEEE Asian Solid-State Circuits Conf.*, pages 14–17, November 2007.

[GNK+01] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S. Mori. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. In *Proc. Int'l Symp. on Microarch.*, pages 225–236, December 2001.

[GT07] B. Greskamp and J. Torrellas. Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 213–224, September 2007.

[GVSS98] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 195–205, January–February 1998.

[HGH06] R. Huang, A. Garg, and M. Huang. Software-Hardware Cooperative Memory Disambiguation. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 248–257, February 2006.

[HMK91] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag Correlating Prefetchers. In *Proc. Int'l Conf. on Supercomputing*, pages 317–326, November 1991.

[HWO98] L. Hammond, M. Wiley, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 58–69, October 1998.

[JBM03] R. Joseph, D. Brooks, and M. Martonosi. Control Techniques to Eliminate Voltage Emergencies in High Performance Processors. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 79–90, February 2003.

[JG99] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *Theoretical Computer Science*, 48(2):121–133, February 1999.

[JMH97] T. Johnson, M. Merten, and W Hwu. Run-time Spatial Locality Detection and Optimization. In *Proc. Int'l Symp. on Microarch.*, pages 57–64, December 1997.

[JRB+98] S. Jourdan, R. Ronnen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *Proc. Int'l Symp. on Microarch.*, pages 216–225, November–December 1998.

[Kes99] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[KJM+06] H. Kim, J. Joao, O. Mutlu, , and Y. Patt. Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths. In *Proc. Int'l Symp. on Microarch.*, pages 53–64, December 2006.

[KKCM05] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed Early Load Retirement. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 16–27, February 2005.

[KL91] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proc. Int'l Symp. on Comp. Arch.*, pages 43–53, May 1991.

[KL04] I. Kim and M. Lipasti. Understanding Scheduling Replay Schemes. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 198–209, February 2004.

[KMSP05] H. Kim, O. Mutlu, J. Stark, and Y. Patt. Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In *Proc. Int'l Symp. on Microarch.*, pages 43–54, December 2005.

[KPG98] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the PolyPath Architecture. In *Proc. Int'l Symp. on Comp. Arch.*, pages 250–259, June–July 1998.

[KS02] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. In *Proc. Int'l Symp. on Comp. Arch.*, pages 195–206, May 2002.

[KV97] S. Kim and A. Veidenbaum. Stride-Directed Prefetching for Secondary Caches. In *Proc. Int'l Conf. on Parallel Processing*, pages 314–321, August 1997.

[KW98] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proc. Int'l Symp. on Comp. Arch.*, pages 357–368, June–July 1998.

[KY02] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 159–170, October 2002.

[LFF01] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proc. Int'l Symp. on Comp. Arch.*, pages 144–154, June 2001.

[LKL+02] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. Int'l Symp. on Comp. Arch.*, pages 59–70, May 2002.

[LL00] K. Lepak and M. Lipasti. On the Value Locality of Store Instructions. In *Proc. Int'l Symp. on Comp. Arch.*, pages 182–191, June 2000.

[LM96] C. Luk and T. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 222–233, October 1996.

[LSKR95] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. SPAID: Software Prefetching in Pointer- and Call- Intensive Environments. In *Proc. Int'l Symp. on Microarch.*, pages 231–236, November–December 1995.

[Luk01] C. Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 40–51, June 2001.

[LWW+02] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Conf. on Programming Language Design and Implementation*, pages 117–128, May 2002.

[McF93] S. McFarling. Combining Branch Predictors. Wrl technical note tn-36, DEC Western Research Laboratory, Palo Alto, California, June 1993.

[MDWB01] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. `alto`: A Link-Time Optimizer for the Compaq Alpha. *Software: Practices and Experience*, 31(1):67–101, January 2001.

[Mei03] S. Meier. Store Queue Multimatch Detection, February 2003.

[MHP05] O. Mutlu, K. Hyesoon, and Y. Patt. Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns. In *Proc. Int'l Symp. on Microarch.*, pages 233–244, December 2005.

[MHW03] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proc. Int'l Symp. on Comp. Arch.*, pages 182–193, June 2003.

[ML00] T. Mowry and C. Luk. Understanding Why Correlation Profiling Improves the Predictability of Data Cache Misses in Nonnumeric Applications. *IEEE Transactions on Computers*, 49(4):369–384, 2000.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 62–73, October 1992.

[MMHR06] F. Mesa-Martinez, M. Huang, and J. Renau. SEED: Scalable, Efficient Enforcement of Dependences. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 254–264, September 2006.

[MMR07] F. Mesa-Martinez and J. Renau. Effective Optimistic-Checker Tandem Core Design Through Architectural Pruning. In *Proc. Int'l Symp. on Microarch.*, pages 236–248, December 2007.

[MPB01] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice-processors: an Implementation of Operation-Based Prediction. In *Proc. Int'l Conf. on Supercomputing*, pages 321–334, June 2001.

[MPV93] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. In *Proc. Int'l Symp. on Microarch.*, pages 202–213, December 1993.

[MRH+02] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proc. Int'l Symp. on Microarch.*, pages 3–14, November 2002.

[MS97] A. Moshovos and G. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *Proc. Int'l Symp. on Microarch.*, pages 235–245, December 1997.

[MSWP03] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 129–140, February 2003.

[NS04] K. Nesbit and J. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, page 96, February 2004.

[PA99] V. Pai and S. Adve. Code Transformations to Improve Memory Parallelism. In *Proc. Int'l Symp. on Microarch.*, pages 147–155, November 1999.

[PCG+06] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 53–64, February 2006.

[PJS97] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 206–218, June 1997.

[PK94] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proc. Int'l Symp. on Comp. Arch.*, pages 24–33, April 1994.

[Por89] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[POV03] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proc. Int'l Symp. on Microarch.*, pages 411–422, December 2003.

[PPW02] M. Pant, P. Pant, and D. Wills. On-Chip Decoupling Capacitor Optimization Using Architectural Level Prediction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):319–326, June 2002.

[PPWT99] M. Pant, P. Pant, D. Wills, and V. Tiwari. An Architectural Solution for the Inductive Noise Problem due to Clock-Gating. In *Proc. Int'l Symp. on Low-Power Electronics and Design*, pages 255–257, August 1999.

[PSR00] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. In *Proc. Int'l Symp. on Microarch.*, pages 269–280, December 2000.

[PV03] M. Powell and T. Vijaykumar. Pipeline Damping: A Microarchitectural Technique to Reduce Inductive Noise in Supply Voltage. In *Proc. Int'l Symp. on Comp. Arch.*, pages 72–83, June 2003.

[PV04] M. Powell and T. Vijaykumar. Exploiting Resonant Behavior to Reduce Inductive Noise. In *Proc. Int'l Symp. on Comp. Arch.*, pages 288–299, June 2004.

[RMS98] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 115–126, October 1998.

[Rot04] A. Roth. A High-Bandwidth Load-Store Unit for Single- and Multi- Threaded Processors. Technical Report (CIS), Development of Computer and Information Science, University of Pennsylvania, September 2004.

[Rot05] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. Int'l Symp. on Comp. Arch.*, pages 458–468, June 2005.

[RS99] A. Roth and G. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proc. Int'l Symp. on Comp. Arch.*, pages 111–121, May 1999.

[RS01] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 37–48, January 2001.

[RSC+06] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Energy-Efficient Thread-Level Speculation on a CMP. *IEEE Micro*, 26(1):80–91, January/February 2006.

[SBV95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 414–425, June 1995.

[SC00] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical report, IBM T. J. Watson Research Center, October 2000.

[SDB+03] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proc. Int'l Symp. on Microarch.*, pages 399–410, December 2003.

[Sel92] C. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[SJT02] Y. Solihin, L. Jaejin, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proc. Int'l Symp. on Comp. Arch.*, pages 171–182, May 2002.

[SM98] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 2–13, January–February 1998.

[Smi84] J. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[SMR05] T. Sha, M. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *Proc. Int'l Symp. on Microarch.*, pages 159–170, December 2005.

[SP88] J. Swensen and Y. Patt. Hierarchical Registers for Scientific Computers. In *Proc. Int'l Conf. on Supercomputing*, pages 346–354, June 1988.

[SPR00] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 257–268, November 2000.

[SRA+04] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, , and M. Upton. Continual Flow Pipelines. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, October 2004.

[SSH+03] K. Skadron, M. Stan, W. Huang, S. Velusamy, and K. Sankaranarayanan. Temperature-Aware Microarchitecture. In *Proc. Int'l Symp. on Comp. Arch.*, pages 2–13, June 2003.

[SWA+06] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A Moshovos. Spatial Memory Streaming. In *Proc. Int'l Symp. on Comp. Arch.*, pages 252–263, June 2006.

[SWF05] S. Stone, K. Woley, and M. Frank. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding. In *Proc. Int'l Symp. on Microarch.*, pages 171–182, December 2005.

[TA03] J. Tseng and K. Asanovic. Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 62–71, June 2003.

[TDF+02] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[TIVL05] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *Proc. Int'l Symp. on Comp. Arch.*, pages 469–480, June 2005.

[TNM99] A. Tyagi, H. NG, and P. Mohapatra. Dynamic Branch Decoupled Architecture. In *Proc. Int'l Conf. on Computer Design*, pages 442–450, October 1999.

[TNN+04] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing. In *Proc. Int'l Symp. on Performance Analysis of Systems and Software*, pages 78–87, March 2004.

[Tom67] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[USH95] A. Uht, V. Sindagi, and K Hall. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. Int'l Symp. on Microarch.*, pages 313–325, November–December 1995.

[WB96] S. Wallace and N. Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 179–184, October 1996.

[WCT98] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. In *Proc. Int'l Symp. on Comp. Arch.*, pages 238–249, June–July 1998.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. Int'l Symp. on Comp. Arch.*, pages 24–36, June 1995.

[YP92a] T. Yeh and Y. Patt. A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History. In *Proc. Int'l Symp. on Comp. Arch.*, pages 257–266, May 1992.

[YP92b] T. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proc. Int'l Symp. on Comp. Arch.*, pages 124–134, May 1992.

[YR95] R. Yung and N. Registers. Caching Processor General Registers. In *Proc. Int'l Conf. on Computer Design*, pages 307–312, October 1995.

[Zho05] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 231–242, September 2005.

[ZS00] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. Int'l Symp. on Comp. Arch.*, pages 172–181, June 2000.

[ZS01] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proc. Int'l Symp. on Comp. Arch.*, pages 2–13, June 2001.

[ZS02] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proc. Int'l Symp. on Microarch.*, pages 85–96, November 2002.

[ZTC07] W. Zhang, D Tullsen, and B. Calder. Accelerating and Adapting Precomputation Threads for Efficient Prefetching. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 85–95, February 2007.