Summer 8-13-2010

Energy Efficient Load Latency Tolerance: Single-Thread Performance for the Multi-Core Era

Andrew D. Hilton
University of Pennsylvania, adhilton@cis.upenn.edu

Energy Efficient Load Latency Tolerance: Single-Thread Performance for the Multi-Core Era

Abstract

Around 2003, newly activated power constraints caused single-thread performance growth to slow dramatically. The multi-core era was born with an emphasis on explicitly parallel software. Continuing to grow single-thread performance is still important in the multi-core context, but it must be done in an energy efficient way.

One significant impediment to performance growth in both out-of-order and in-order processors is the long latency of last-level cache misses. Prior work introduced the idea of load latency tolerance---the ability to dynamically remove miss-dependent instructions from critical execution structures, continue execution under the miss, and re-execute miss-dependent instructions after the miss returns. However, previously proposed designs were unable to improve performance in an energy-efficient way---they introduced too many new large, complex structures and re-executed too many instructions.

This dissertation describes a new load latency tolerant design that is both energy-efficient, and applicable to both in-order and out-of-order cores. Key novel features include formulation of slice re-execution as an alternative use of multi-threading support, efficient schemes for register and memory state management, and new pruning mechanisms for drastically reducing load latency tolerance's dynamic execution overheads.

Area analysis shows that energy-efficient load latency tolerance increases the footprint of an out-of-order core by a few percent, while cycle-level simulation shows that it significantly improves the performance of memory-bound programs. Energy-efficient load latency tolerance is more energy-efficient than---and synergistic with---existing performance technique like dynamic voltage and frequency scaling (DVFS).

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Amir Roth

Keywords
BOLT, iCFP, Computer Architecture, Latency Tolerance, Speculative Retirement

Subject Categories
Other Computer Sciences

ENERGY EFFICIENT LOAD LATENCY TOLERANCE: SINGLE-THREAD PERFORMANCE FOR THE MULTI-CORE ERA

Andrew D. Hilton

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2010

Amir Roth, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Jianbo Shi, Associate Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Milo Martin, Associate Professor of Computer and Information Science

José Martínez, Associate Professor of Electrical and Computer Engineering

Andre DeHon, Associate Professor of Electrical and System Engineering

Rahul Mangharam, Assistan Professor of Electrical and System Engineering

Energy Efficient Load Latency Tolerance: Single-Thread Performance for the Multi-Core Era

Acknowledgements

Graduate school has been a long journey with many ups and downs, successes and frustrations, joys and sorrows. There are many people who I would like to thank for their help along the way.

First, I would like to thank my advisor, Amir Roth. If not for my enjoyment of your class, I would probably have never become an architect. Your help along the way has been invaluable—advice, refinements to ideas, and encouragement.

I would like to thank my committee—Milo Martin, Jose Martinez, Andre DeHon, and Rahul Mangharam—for their useful advice and suggestions along the way.

My academic siblings—Vlad Petric, Anne Bracy, and Tingting Sha—have provided help and encouragement, and have been with me through many all-nighters. I would also like to thank the other members of my research group, past and present—Marc Corliss, Colin Blundell, Arun Raghavan, Santosh Nagarakatte, Neeraj Eswaran, and Vivek Rane.

I have had many outstanding friends here, who provided moral support and encouragement, as well as fun. Jeff Vaughan and Nick Taylor were the best house-mates one could ask for. I'd also like to thank Karl Mazurak, Kuzman Ganchev, Brian Aydemir, Emily Pitler, and Jenn Wortman Vaughan for their friendship and support. I'd like to thank Maestro Mark Masters and everyone at the Fencing Academy of Philadelphia for all the enjoyment I had fencing during my time in graduate school.

I'd like to thank my family for all they have done for me throughout my life. I'd specifically like to thank my parents for pushing me to excel academically throughout my entire childhood.

Finally, I'd like to thank my girlfriend, Margaret Foster for all of her support, encouragement, and love.

ABSTRACT

ENERGY EFFICIENT LOAD LATENCY TOLERANCE: SINGLE-THREAD PERFORMANCE FOR THE MULTI-CORE ERA

Andrew D. Hilton

Amir Roth

Around 2003, newly activated power constraints caused single-thread performance growth to slow dramatically. The multi-core era was born with an emphasis on explicitly parallel software. Continuing to grow single-thread performance is still important in the multi-core context, but it must be done in an energy efficient way.

One significant impediment to performance growth in both out-of-order and in-order processors is the long latency of last-level cache misses. Prior work introduced the idea of *load latency tolerance*—the ability to dynamically remove miss-dependent instructions from critical execution structures, continue execution under the miss, and re-execute miss-dependent instructions after the miss returns. However, previously proposed designs were unable to improve performance in an energy-efficient way—they introduced too many new large, complex structures and re-executed too many instructions.

This dissertation describes a new load latency tolerant design that is both energy-efficient, and applicable to both in-order and out-of-order cores. Key novel features include formulation of slice re-execution as an alternative use of multi-threading support, efficient schemes for register and memory state management, and new pruning mechanisms for drastically reducing load latency tolerance's dynamic execution overheads.

Area analysis shows that energy-efficient load latency tolerance increases the footprint of an out-of-order core by a few percent, while cycle-level simulation shows that it significantly improves the performance of memory-bound programs. Energy-efficient load latency tolerance is more energy-efficient than—and synergistic with—existing performance technique like dynamic voltage and frequency scaling (DVFS).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Historically, efforts to extract single-thread performance at almost any cost have been a significant theme in computer architecture. Starting around 2005, this trend changed as energy became a primary concern in a wide range of domains—from embedded devices in which battery life is critical to large server rooms where power supply and cooling costs dominate budgets.

The promotion of energy to a first-order design constraint has slowed single-thread performance growth. One way in which this has happened is by slowing the scaling of clock frequency. Increasing clock frequency increases performance at most linearly. In a limited range it can be done by increasing power linearly as well. Increasing clock frequency in a broader range requires increasing the voltage as well, which leads to a quadratic increase in energy consumption. The 1980s and 1990s *modus operandi* of obtaining performance by increasing clock frequency is not suited to modern energy-constrained designs. Improving performance by increasing instruction level parallelism (ILP)—executing more instructions in a single clock cycle—is also constrained by energy. Techniques that increase ILP typically require increasing superscalar width, or making pipeline structures larger, or both. Both of these increase energy consumption but increase performance sublinearly.

The need for more energy-efficient performance has led to the proliferation of multi-core designs. Multiple independent cores can deliver high performance if programs are written to take advantage of them—explicitly multi-threaded, so that threads can run on different cores. The energy efficiency of multi-core designs comes from avoiding the quadratic costs associated with many aspects of single-thread performance. Four structures of size N consume less energy than one structure of size 4N. Similarly, four structures with one port each consume less energy than one structure with four ports, and four cores operating at 1GHz consume less energy than one core operating at 4GHz.

Future multi-cores are likely to be heterogeneous and will include a small number of large high-performance cores (*i.e.*, superscalar, out-of-order cores similar to Intel's Core i7) and a large number of small simple cores (*i.e.*, scalar, in-order similar to Intel's Atom). The simple cores will execute parallel regions of code, while the large cores will execute serial regions. One aspect of exploiting multi-cores is parallelizing important applications. However, an equally important aspect is single-thread performance of both the small and large cores. The large cores will have to deliver high performance to prevent the serial portions of the workload from dominating performance [29]—this is Amdahl's Law. While the small cores will exploit parallelism where it exists, their individual performance will be important because parallelism is not perfectly elastic—many applications will not scale to arbitrarily large numbers of threads. Because energy considerations will limit the number of cores which may be active at any given time, the performance techniques used in both the high-performance and simple cores must be energy efficient.

1.1 DVFS: Accelerating Computation-Bound Programs

One simple and common technique for increasing the performance of both out-of-order and in-order cores is Dynamic Voltage and Frequency Scaling (DVFS), also known as "TurboBoost" [1]. When single-thread performance is needed on a given core, the voltage

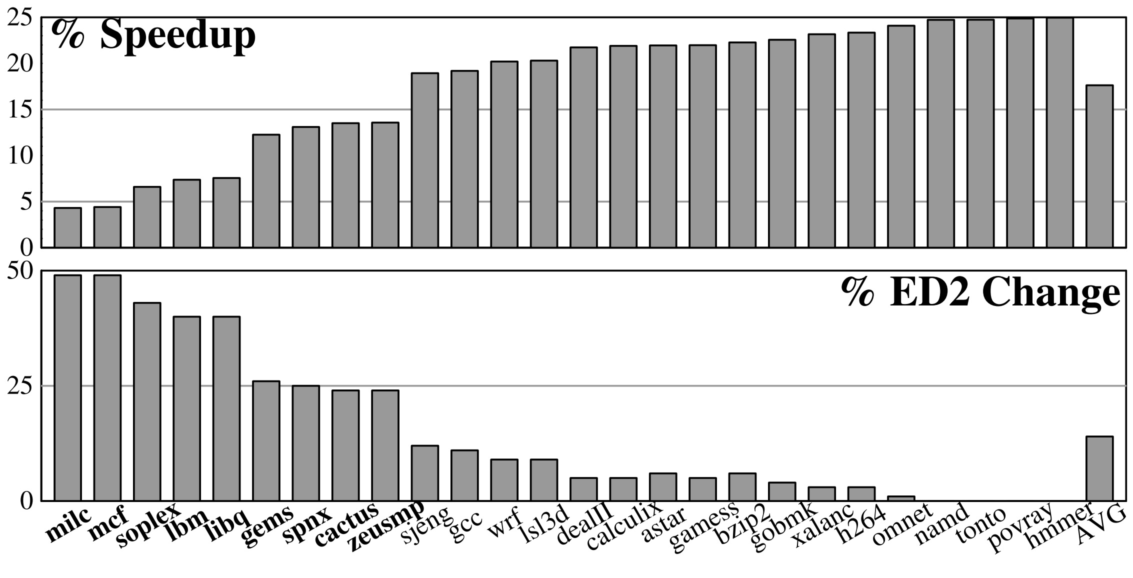


Figure 1.1: Top: Performance impact of using DVFS to increase clock frequency from 3.2 GHz to 4.0 GHz on a simulated design similar to Core i7. Bottom: Energy efficiency as measured by percent change in Energy × Delay^2.

and clock frequency of that core are increased. DVFS increases the energy consumption of the core quadratically, so it cannot be applied to all cores at once, and it most likely requires other cores to be powered down in order to meet the power budget. This quadratic increase in energy consumption corresponds to a linear increase in performance in the ideal case, when a program experiences no cache misses and its performance is constrained by in-core computation. It corresponds to almost no increase in the worst case, when a program misses the cache frequently and its performance is constrained by latencies outside the core.

The top graph in Figure 1.1 shows the performance impact of a 25% increase in clock frequency on a simulated configuration similar to Core i7. For more details about the simulated configuration, benchmarks used, and simulation methodology, see Chapter 5. In this graph, the benchmarks on the right (*namd*, *hmmer*, *povray*) obtain the ideal 25% speedup. Under these ideal conditions, DVFS keeps Energy × Delay^2 (ED^2) constant, making ED^2 a useful metric for energy efficiency [47]. The bottom graph of Figure 1.1 shows the percent change in ED^2 corresponding to the speedups in the top graph. While

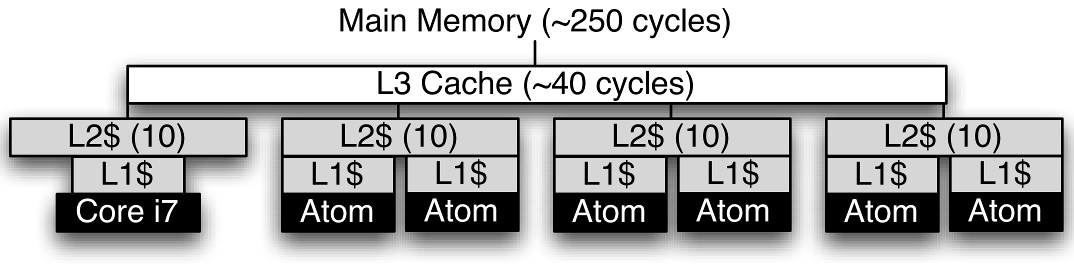


Figure 1.2: An example cache hierarchy for a hypothetical future heterogenous multi-core chip.

the benchmarks on the right side of the graph obtain ideal speedups, other benchmarks do not. In fact, the left-most benchmarks (*mcf*, *milc*, *soplex*, etc.) on the graph obtain little speedup. Correspondingly, these benchmarks have significant increases in ED^2—almost 50%—indicating that their performance gains are highly energy *inefficient*. The problem with these programs is that they are memory-bound—their performance is dominated by misses to memory or to the last level cache, which are not in the core's clock domain and are not affected by DVFS.

1.2 Load Latency Tolerance: Accelerating Memory-bound Programs

Figure 1.2 shows an example cache hierarchy for a (hypothetical) heterogenous multi-core. The cache hierarchy itself is modeled on the cache hierarchy of current 4-core Core i7s, but some of the Core i7s have been replaced with in-order Atom cores. Accesses which hit the first level (L1) cache incur no latency penalty, while those which miss incur latency penalties ranging from 10 cycles (hitting the L2) to 250 cycles (missing all the way to memory). A program is "memory-bound" when its performance is dominated by long-latency memory accesses, typically loads. When a long-latency load occurs, a processor may be able to execute a few independent instructions, but its critical execution resources—physical registers and issue queue entries for out-of-order; pipeline latches for in-order—quickly fill and executions halts. Latencies that are long enough to pose significant performance issues can be incurred not only by accesses which go all the way to main memory, but also by sufficiently large numbers of accesses which are serviced by the L3 cache. For an in-order core, a program may be memory-bound due to a sufficiently large number of L1 misses, even if they hit in the L2.

Increasing the core's clock frequency does little to help memory-bound programs. Only the first two levels of cache (shown in light grey) operate on the same clock as the core. The L3 cache and main memory, are in different clock domains. Accesses handled at these levels take the same amount of time no matter what the core's clock frequency is. Increasing the clock frequency however, makes these accesses take relatively more clock cycles, meaning that the core consumes more energy while waiting for the data.

Previous work [44, 58, 60, 76] introduced the technique of *load latency tolerance*—long-latency misses and their dependents temporarily release their resources, wait on the side, then re-execute after the miss returns. Load latency tolerant processors improve the performance of memory-bound programs by allowing the pipeline to continue to execute younger instructions while waiting for the long-latency miss to return. However, previous latency-tolerant processor designs are not energy-efficient. They used too many additional structures and deferred and re-executed too many miss-dependent instructions to justify their performance gains.

1.3 Energy Efficient Load Latency Tolerance: BOLT and iCFP

To achieve industry adoption, a load latency tolerant design must be energy efficient both statically—it must re-use existing structures when possible and ensure that new structures impose little energy overhead—and dynamically—it must minimize the energy overhead introduced by re-executing instructions. This dissertation describes BOLT (Better Out-of-order Latency Tolerance) [35] and iCFP (In-order Continual Flow Pipeline) [31] respectively out-of-order and in-order load latency tolerant designs which achieve energy efficiency. BOLT and iCFP implement the re-execution of miss-dependent instructions as an alternative use of multi-threading hardware, maximizing re-use of existing structures. They avoid associative structures for load and store management, favoring simple, low-overhead indexed structures. Additionally, they employ simple pruning techniques to minimize the re-execution overheads introduced by load latency tolerance. This dissertation focuses primarily on BOLT. It includes a detailed description of all mechanisms and a comparative, simulation-driven performance evaluation. In the interest of completeness, it also describes iCFP and qualitatively compares it to other in-order load latency tolerant designs. However, it does not include a simulation-driven performance/energy evaluation of iCFP[1].

The novel mechanisms presented in this dissertation include:

- **Out-of-order register management for load latency tolerance.** This dissertation describes an efficient implementation of a new hybrid physical register management algorithm for out-of-order load latency tolerant processors. This new algorithm combines the benefits of conventional register management for the youngest in-flight instructions with the benefits of aggressive register reclamation for the oldest miss-dependent instructions. This register management scheme also allows re-renaming of miss-dependent instructions to be framed as an alternative use of existing multi-threading hardware.

- **In-order register management for load latency tolerance.** This dissertation describes efficient register management for an in-order load latency tolerant processor. Like the out-of-order design, the in-order design frames re-execution of miss-dependent instructions as an alternative use for existing multi-threading support.

[1] Simulated performance results for iCFP are available in previously published work [31], however, that work uses a different ISA, simulation environment, baseline configuration, and set of benchmarks than the rest of this dissertation.

- **Pruning mechanisms for load latency tolerance.** This dissertation describes pruning mechanisms which reduce the number of instructions requiring re-execution. These mechanisms address two different forms of inefficiency—avoiding unnecessary re-execution and avoiding applying load latency tolerance to "pointer chasing" code, where it is ineffective.

- **Chained Store Buffer.** This dissertation describes a chained store buffer—a novel scalable, searchable store buffer design. Scalable search is obtained by overlaying an address-indexed hash-table on top of an indexed store buffer, allowing loads to search using iterative indexed access.

- **SVW-MT.** This dissertation describes SVW-MT—the inter-thread verification aspects of SVW—and two optimizations needed to make it work efficiently.

- **Decoupled Store Completion/Silent Deterministic Replay.** This dissertation describes Silent Deterministic Replay (SDR), a technique which permits non-speculative retirement of instructions in between checkpoints in a speculative retirement setting. SDR enables Decoupled Store Completion (DSC), the ability to non-speculatively complete stores to the data cache before the checkpoint they reside in is completed.

Figure 1.3 highlights the results presented in this dissertation. It compares Continual Flow Pipelines [76], the prior state-of-the-art out-of-order load latency tolerant design, to BOLT in terms of performance (top, higher is better) and energy efficiency (bottom, lower is better). A more detailed comparison of the two micro-architectures is presented in Section 5.4. These results show three significant advantages for BOLT:

- **Performance.** BOLT out-performs CFP by a significant amount. The difference here is most noticeable in the average across the memory-bound subset of the SPEC 2006 benchmarks where BOLT's performance gains are almost *twice* those CFP (29% vs 15%).

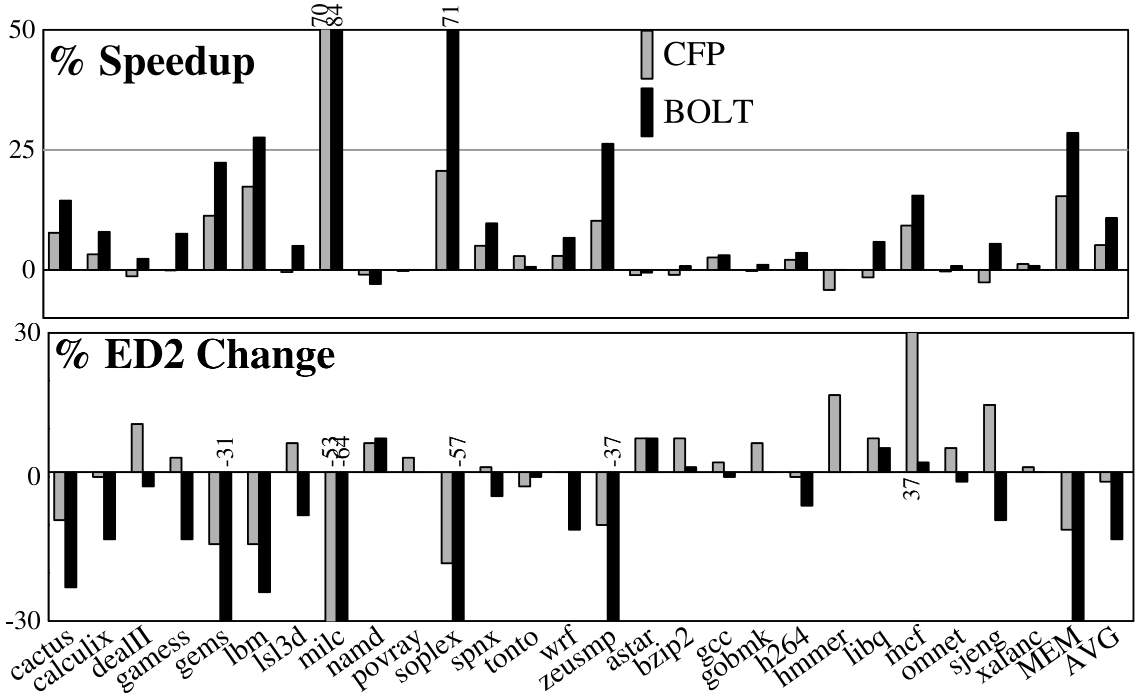


Figure 1.3: Top: The performance improvements of Continual Flow Pipelining (prior state-of-the-art load latency tolerant design), and BOLT. Bottom: The corresponding percent change in ED^2.

- **Energy efficiency.** BOLT has significantly better energy efficiency (lower ED^2) than CFP. While CFP averages only 1% ED^2 improvement overall and 11% on the memory-bound subset of SPEC, BOLT averages 13% overall and 30% on the memory-bound programs.

- **Performance and energy efficiency robustness.** BOLT more closely adheres to the architectural "Hippocratic Oath." CFP suffers slowdowns of more than 1% on five programs, compared to one for BOLT. In terms of energy efficiency, CFP increases (harms) ED^2 by more than 1% on 16 programs—more than half—including four programs which it harms by more than 10%. By contrast, BOLT only increases ED^2 by more than 1% on five programs and its worst case ED^2 increase is 8%.

1.4 Contributions

The main contributions of this dissertation are:

- **Similarities of in-order and out-of-order load latency tolerance.** This dissertation shows that both in-order and out-of-order processors can tolerate load latency using almost identical mechanisms.

- **Framing re-execution as multi-threading.** This dissertation shows that the instructions re-executed by load latency tolerant designs behave similarly to a thread. BOLT and iCFP re-use existing multi-threading support for this re-execution.

- **Hybrid register management in out-of-order processors.** This dissertation shows how different register management schemes may be hybridized in an out-of-order processor using reference counting. In BOLT, this hybridization is used to attach a register-efficient post-processing step to a conventional out-of-order processor.

- **Efficient, scalable load/store queue design.** This dissertation shows how SQIP, CSB, DSC/SDR, SVW, and SVW-MT can be combined into an efficient, scalable load/store queue design which is well-suited to load latency tolerance in an out-of-order processor. An in-order latency processor does not need SQIP and SVW, but can use the other mechanisms.

- **Deferral and re-execution pruning mechnaisms.** This dissertation shows that re-execution overheads can be a significant energy in-efficiency in a load latency tolerant design, and proposes mechanisms to address the inefficiency while still maintaining performance.

This dissertation presents a simulation-driven performance and energy evaluation of BOLT—the out-of-order design—including the impact of the register substrate, load and store structures, and the pruning mechanisms. iCFP is qualitatively compared to other in-order load latency tolerance designs. A quantitative evaluation of iCFP's performance may be found in previously published results [31].

1.5 Previously Published Work

Aspects of this dissertation have been published already. These publications include:

- **iCFP: Tolerating All-Level Cache Misses in In-Order Processors.** This 2009 publication appeared in the 15^{th} International Symposium on High Performance Computer Architecture [31]. It describes the in-order register state management in iCFP, as well as the chained store buffer. iCFP also appeared in MICRO's Top-Picks [32]. That manuscript also included a qualitative comparison of iCFP and Sun's Rock processor.

- **DSC/SDR: Enabling Scalable Data Memory for CPR/CFP Processors.** This 2009 publication appeared in the 36^{th} International Symposium on Computer Architecture [34]. It describes Decoupled Store Completion and Silent Deterministic Replay.

- **BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Execution.** This 2010 publication appeared in the 16^{th} International Symposium on High Performance Computer Architecture [35]. It describes BOLT— the out-of-order load latency tolerance design—including its register management and pruning mechanisms.

This dissertation differs from the previously published work in the following ways:

- BOLT is evaluated more thoroughly.

- SVW-MT is explained in detail, including optimizations which make it work efficiently.

- The "pointer chasing" pruning mechanism presented in this dissertation (in Section 4.3) is different from the one presented in the BOLT paper.

1.6 Document Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents background on previously proposed out-of-order load latency tolerance designs. Chapter 3 describes the BOLT micro-architecture. Chapter 4 describes the pruning mechanisms. Chapter 5 analyses the performance and energy efficiency of BOLT and compares it to other designs. Chapter 6 describes iCFP and other in-order load latency tolerance designs. Chapter 7 concludes. Appendix A provides an evaluation of SVW-MT. Appendix B examines the sensitivity of the energy model to the assumptions it makes.

Chapter 2

Background

Long latency loads pose a significant performance problem, as the processor must stall for many cycles while waiting for data. This problem can be mitigated by a processor's ability to execute independent instructions while waiting for a long latency miss to return. Gains can come from one of two types of parallelism. Memory-level parallelism (MLP) occurs when the processor overlaps the memory latency of two or more independent loads. Instruction-level parallelism (ILP) occurs when the processor executes independent instructions under the miss, preserves that work, and does not revisit those instructions after the miss returns. Both forms of parallelism are limited by the processor's ability to re-order instructions and find independent operations. Alternatively, the processor may decrease its voltage and clock frequency while a long latency miss is pending to save energy [45].

An in-order processor has little to no ability to re-order instructions, and typically can not extract ILP or MLP under long latency cache misses. The left side of Figure 2.1 shows an example of how long latency operations inhibit performance in the in-order case. This example depicts a sequence of 8 instructions, A–H, shown on the far left. The diagram shows how each of these instructions executes in time. Loads A and H miss the last level cache, resulting in long latencies, as indicated by the wide bars in their rows. The other instructions (B–G) each have short latency and execute quickly. As this is an in-order processor, no re-ordering is performed, the misses serialize, and their latencies add. The

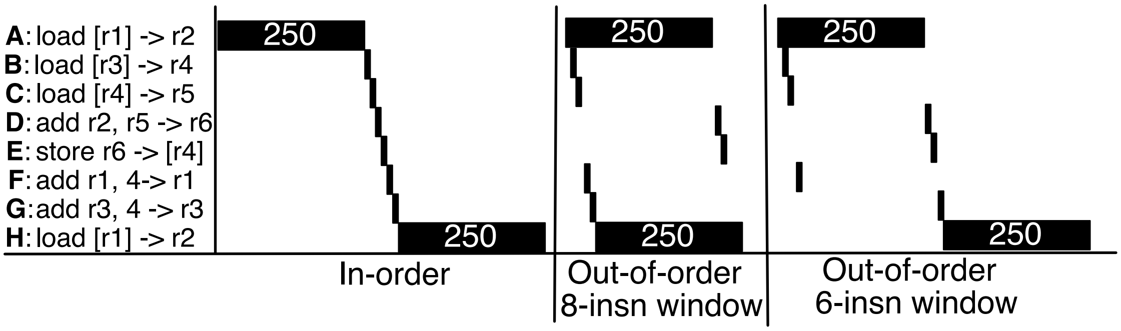


Figure 2.1: Execution behavior over time. Loads A and H miss the caches. Left: An in-order processor serializes all instructions. The misses dominate performance. Middle: An out-of-order processor can execute independent operations while waiting for A's miss to return. It may be able to overlap A's miss with H's, exposing MLP. Right: If the out-of-order processor's window is too small, it may not be able to overlap A and H, resulting in performance closer to the in-order case.

miss latencies dominate performance, accounting for 99% of execution time.

In many real processors, sequential accesses such as those in Figure 2.1 are often detected and prefetched—a dedicated hardware structure monitors cache accesses, detects simple patterns, and initiates misses before any load attempts to read the data [17, 18, 39, 40, 70, 71]. Prefetching improves performance, but is limited by the types of patterns it can discern. Many of the examples in this dissertation ignore prefetching to facilitate simple and clear examples. However, the simulated results in Chapter 5 include stream prefetching all configurations.

The middle of Figure 2.1 shows the performance gains that an out-of-order processor can achieve in this same situation. Here, the processor can execute independent instructions B, C, F, G, and H while it waits for A's miss to return. The most significant performance gain here comes from MLP—the out-of-order processor overlaps H's miss with A's, meaning their latencies no longer add. The processor also obtains some performance gains from ILP—the execution of B,C, F, and G—however, in this example, those gains are a small component of the overall performance improvement.

While an out-of-order processor is able to perform some re-ordering of independent operations to improve performance, it has a limited *window*—the range of instructions in which the processor can seek re-orderings—in which is able to do so. The size of this window determines how much ILP and MLP an out-of-order processor can extract under a miss. The out-of-order processor in the middle of Figure 2.1 has a window of eight instructions, meaning it can examine all instructions from A to H to find independent instructions. By contrast, if the out-of-order window were only six instructions, as pictured on the right side of Figure 2.1, then the processor could not execute G until A finished and exited the window. The important consequence of this smaller window is that H cannot execute until after A exits the window, again serializing the misses and causing their latencies to add.

2.1 Non-scalability of the Out-of-order Window

While a large out-of-order window is desirable for performance, it is neither practical nor energy efficient. The out-of-order window is restricted by four non-scalable structures: the issue queue, the physical register file, the load queue, and the store queue. In modern, conventional out-of-order processors, these structures are sized to tolerate moderate latencies of about 20–30 cycles. Latencies of this magnitude correspond to last-level cache hits, serially dependent loads hitting the L2 cache, or the longest floating point operations. To tolerate the long latency of last-level cache misses—250 cycles—the capacities of these structures would have to increase by 4–10×. Such increases would drastically increase the access latency and energy consumption of these structures.

2.1.1 Issue Queue

The issue queue holds all un-executed instructions and schedules instructions for execution as their inputs become available. Scheduling consists of two parts—*select* selects N instructions for execution from among the instructions whose inputs are available; *wakeup* updates the ready bits of all instructions that depend on the instructions just selected. Select is usually implemented using priority encoders. Wakeup is implemented using a content associative memory (CAM) that holds the input physical register names of all

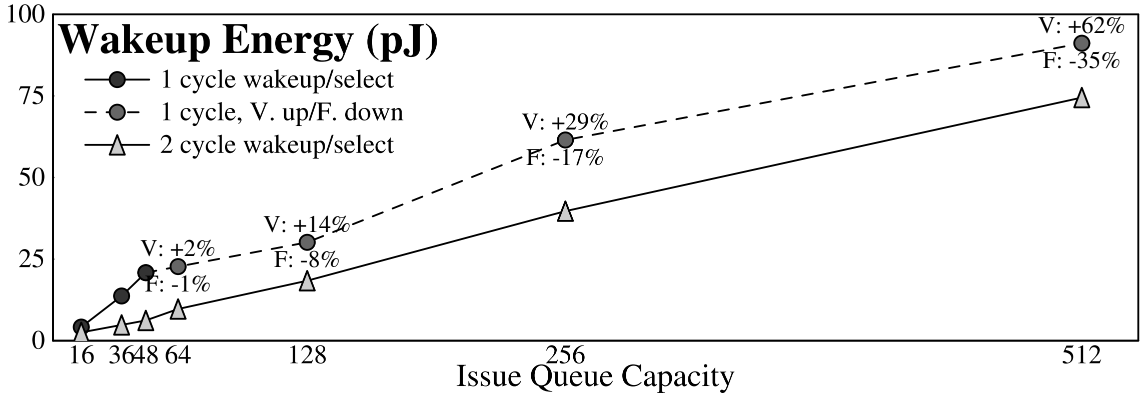


Figure 2.2: Energy scaling of the issue queue. The dashed line indicates designs where 1-cycle wakeup/select requires increasing voltage or decreasing clock frequency.

un-executed instructions. The issue queue in the Intel Core i7 processor has 36 entries, limiting the out-of-order window to 36 un-executed instructions.

Physical scalability. One difficulty with scaling the issue queue arises from the fact that its latency is critical. Scheduling that takes multiple clock cycles, prevents dependent instructions from executing in consecutive cycles. Essentially, the minimum execution latency in the processor becomes that of the scheduler's *wakeup/select* loop. The complexity (and therefore latency) of both wakeup and select increases as the size of the issue queue increases [56].

Figure 2.2 quantifies the scalability of issue queue wakeup. The graph plots the energy required for a single wakeup operation for different issue queue sizes. The data in this graph is derived from a modified version of CACTI 4.1 [79] assuming a 4-wide out-of-order processor. We model an I-entry issue queue as 2 I-word 8-bit CAMs, each with four write ports and 4 associative match ports—each 8-bit entry corresponds to one 8-bit physical register tag. Two CAMs are modeled as each instruction may have two register inputs. CACTI cannot model the select logic, so its latency is estimated from the baseline processor's issue queue. CACTI reports the baseline issue queue (36-entry) has a minimum latency of 284 ps. CACTI reports a design with a 292 ps latency which uses approximately two-thirds as much energy and about one-half the area of the minimum latency design. Assuming that a 3.2GHz Core i7 uses the second design, 20.5 ps remain

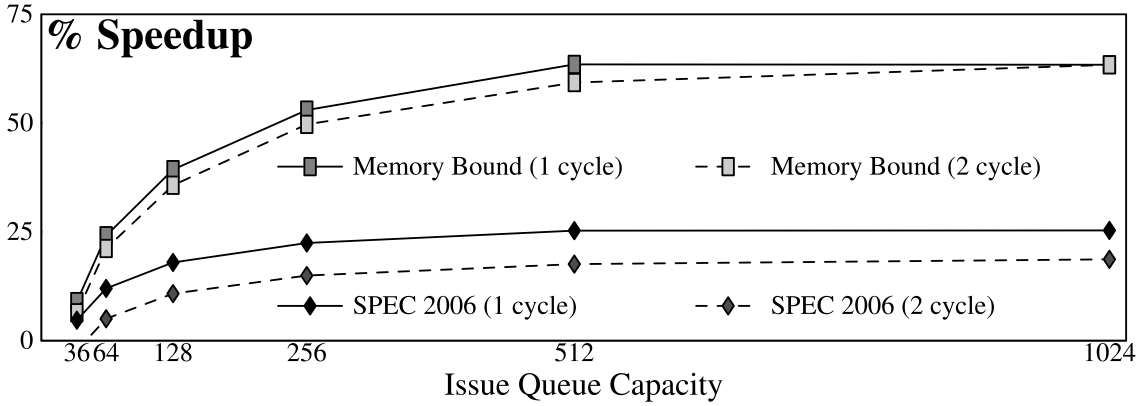


Figure 2.3: Performance scaling of the issue queue for both 1- and 2-cycle wakeup/select. All other window structures are infinite.

for the select logic. The data in the graph uses this estimation—20.5 ps—for all points and does not model the longer select latencies of larger issue queues.

The line marked by dark circles in Figure 2.2 plots the energy requirements for different issue queues sizes when the wakeup/select loop is constrained to a single cycle. For issue queue sizes of 64-entries and larger, CACTI is unable to produce any configuration which permits single-cycle wakeup/select at the baseline clock frequency and voltage. Issue queue designs of these sizes with single-cycle wakeup/select require either increasing voltage or decreasing clock frequency (or some of both). These designs are indicated with slightly lighter circles and a dashed line. Each point has the voltage increase/frequency decrease noted above/below it. The energy values plotted do not account for the voltage change *i.e.*, they are the energy required if frequency were changed. The line marked with light triangles shows the energy consumption when wakeup/select is relaxed to two cycles.

Performance impact. Figure 2.3 shows the geometric mean speedup over the baseline Core i7 configuration of different issue queue sizes and latencies for all of SPEC 2006 (diamonds) and the memory-bound subset (boxes). The solid lines represent single cycle wakeup/select, and the dashed lines represent two cycle wakeup/select. In all configurations, window structures other than the issue queue are made ideally large. This means that even in the configuration with the same size and latency issue queue as the baseline (36-entries, 1-cycle), the idealized window enjoys some speedups due to the benefits of

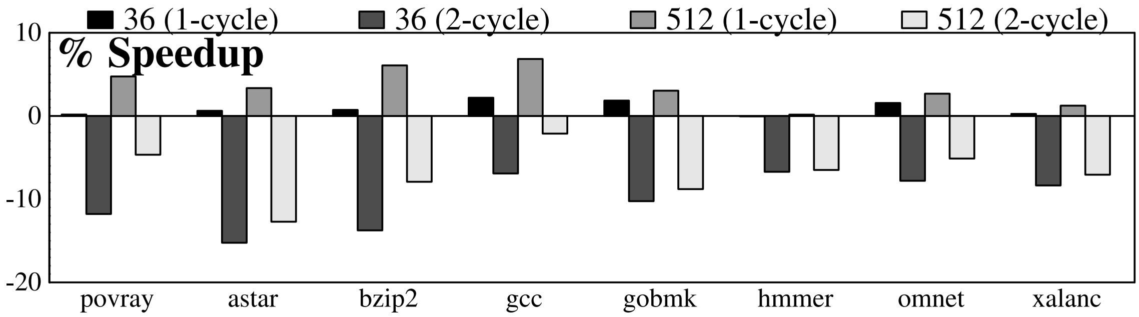


Figure 2.4: Slowdowns suffered by individual programs from 2-cycle wakeup/select.

the other structures. The clock frequency is un-modified in all experiments.

Maximizing IPC throughput on the memory-bound programs requires a 512-entry issue queue. Such a design requires a 6–7× increase in the per-search energy of the issue queue as well as either 2-cycle wakeup select, a 35% frequency decrease, or a 62% voltage increase. Neither drastically increasing voltage nor decreasing frequency are attractive options. The 2-cycle wakeup/select design is also not attractive [10]. Experiments show that programs suffer *average* performance losses of about 7%. However, the designs with 2-cycle wakeup/select suffer slowdowns *compared to the baseline* for individual benchmarks.

Figure 2.4 shows the impact of 2-cycle wakeup select on individual benchmarks. The graph shows four bars—36-entry/1-cycle, 36-entry/2-cycle, 512-entry/1-cycle, and 512-entry/2-cycle. The benchmarks selected for this graph are those which suffer a slowdown of greater than 2% in the 512-entry/2-cycle configuration. These programs suffer from the latency added to operations which are normally single-cycle. Some of these programs (*e.g.*, *povray*, *gcc*, and *bzip2*) mitigate this cost slightly by having large window, but the longer latency operations are the dominant effect here.

Other approaches. Prior research has attacked the non-scalability of the issue queue in various ways. Cyclone [24] replaces a traditional associative wakeup/select issue queue with a FIFO queue/ An extended rename stage places instructions into this queue according to calculated dataflow height. The Cyclone scheduler is scalable, but cache misses result in costly performance penalties. Forwardflow [27] represents register dependences as linked lists—the readers of a given register are chained in a list that starts at that register—and implements wakeup by following list pointers. Forwardflow scales the issue queue, but may impact performance for long dependence lists *i.e.*, register values with high fan-out.

Other techniques examine modifications to the traditional wakeup/select-based design. One approach exploits the fact that most instructions entering the window need to only wait for one input to become available. Putting such instructions into a CAM which only matches one physical register tag allows for lower latency and energy [23]. Other approaches create a large issue queue from multiple smaller structures [56, 63].

Other techniques attack the latency problem of wakeup/select fitting in one cycle, while allowing dependent instructions to issue in consecutive cycles. One approach speculatively wakes up grandchildren (*i.e.*, a dependent instruction's dependent instructions) [77]. This allows wakeup and select to take two cycles, but requires issue queue entries to hold more tags (*i.e.* the parents' inputs) or to speculate one which inputs will arrive last.

2.1.2 Physical Register File

Out-of-order processors rename registers—mapping logical register names to a larger set of physical registers—to eliminate false dependences. The physical register file limits the out-of-order window due to the in-order nature of register allocation and freeing. Registers are allocated in-order at the rename stage, in the front-end of the pipeline. As each instruction is renamed, it is allocated a new physical register for its destination, but also notes the previous physical register mapped to the same logical designation. When an instruction retires—exits the pipeline in program order and updates architected state—the processor reclaims the physical register which the instruction overwrites that was noted at rename. The only other way in which physical registers are reclaimed is during misprediction recovery, when the destination registers of squashed instructions are reclaimed.

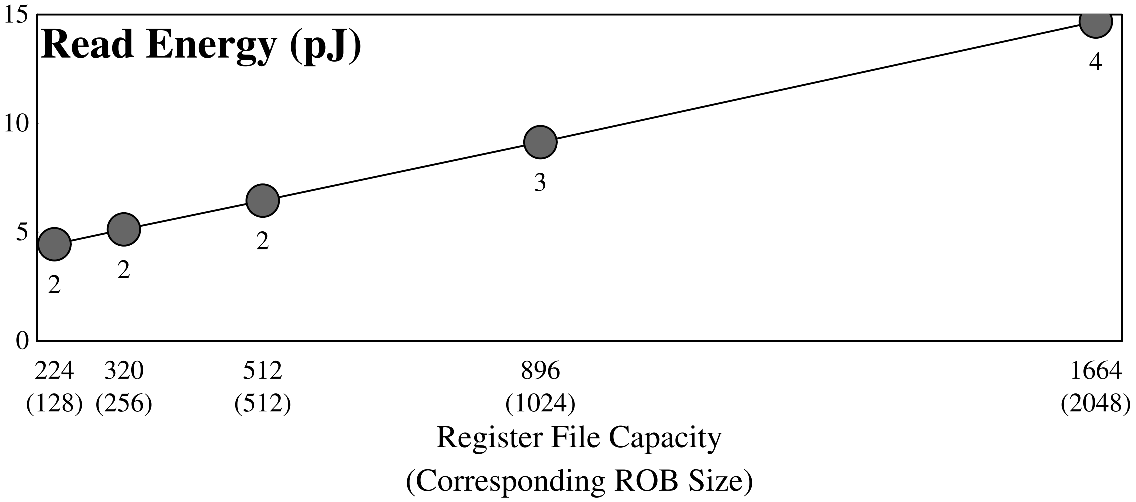


Figure 2.5: Energy and latency scaling of the register file. Numeric labels indicate access time in 3.2 GHz clock cycles. The x-axis is labeled with both the register file size, and the corresponding Reorder Buffer (ROB) size in parenthesis.

The consequence of this conventional register management scheme is that the total number of register writing instructions between rename and retirement is limited by the size of the physical register file. The physical register file is both latency-critical and high-bandwidth. The latency of the physical register file impacts the number of pipeline stages between issue and execution. Increasing the number of pipeline stages here increases the bypass complexity, scheduler speculation depth and complexity, and the branch misprediction penalty. The register file must also support 2N reads and N writes on an N-wide processor. On a 4-wide out-of-order machine, this means the register file requires 12 ports. This bandwidth requirement may be reduced slightly under the assumption that some instructions will receive inputs from the bypass network rather than the register file and that some instructions will not produce register output values. The combination of these two factors limits the register file to a modest number of entries—in Intel's Core i7, enough to support architected state plus about 128 in-flight instructions.

Physical scalability. Figure 2.5 shows the read energy and latency of register files of varying sizes. In CACTI, we model an R-register register file as an R-word 64-bit RAM with 6 read ports and 3 write ports. Under each register file size label on the x-axis, the

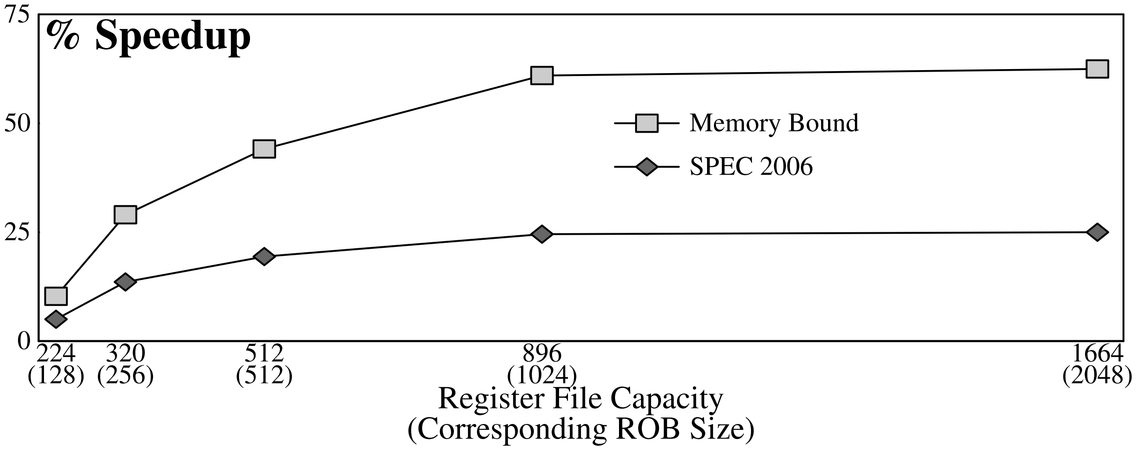


Figure 2.6: Performance impact of register file. Other window structures are infinite. The x-axis is labeled with both the register file size, and the corresponding Reorder Buffer (ROB) size in parenthesis.

corresponding Reorder Buffer (ROB) size is listed in parenthesis. Specifically, this is the ROB size appropriate for the given register file size, assuming 128 architectural registers (64 per-thread with two threads), and that three-quarters of instructions require an output register. The register file access latency in clock cycles is listed under each point. Unlike the other window structures, CACTI does not produce significantly different numbers for the smaller register files with relaxed cycle time constraints.

Performance impact. Figure 2.6 shows the performance effects (again, geometric mean speedup over the baseline configuration) of scaling the register file when all other window structures are ideally large. The ROB size corresponding to each register file size is listed for reference, but the ROB is treated as infinite in all simulations. These simulations also assume that the larger sizes have no impact on access latency, and model 2 cycle register reads for all configurations. An 896-entry register file (*i.e.*, one large enough to support a 1024 instruction window) is required to achieve maximum performance. A register file this large requires 3 cycles to access instead of 2, which reduces performance by up to 3%. The access energy for a register file of this size approximately doubles compared to the baseline 224-entry register file.

Other approaches. Prior research has attacked the non-scalability of the physical register file in various ways. Some approaches, such as *Checkpoint Processing and Recovery* [2], *Cherry* [48], and *Early Register Release* [22] improve register file utilization by freeing registers sooner than in a conventional processor. *Physical Register Inlining* [46] releases physical registers with small values, placing the values directly in the rename map table. Other approaches, such as *Virtual Physical Registers* [50] allocate physical registers only when they are needed by executing instructions. *Ephemeral Registers* [19] combine both late allocation and early release.

Some techniques scale the effective capacity of the register file by allowing certain instructions to share registers with older instructions that produce the same value [25, 41, 61, 62, 75] or by fusing multiple instructions into larger blocks and elimianting register storage for block-interior values [11]. Most of these techniques also amplify the capacity of the issue queue.

2.1.3 Store Queue

Processors must respect data dependences via memory as well as registers. Because cache writes cannot be undone in a conventional processor, a store may not complete to the data cache until after it retires in program order. A load may read the memory location written by an older store which has not yet completed to the data cache. To avoid stalling such loads until the relevant store completes, out-of-order processors allow loads to *forward* values from in-flight stores. To support store-to-load forwarding, out-of-order processors use a store queue that holds the addresses and data values of in-flight stores in program order. When a load executes, it searches the store queue for an older store to the same address. If a match is found, the load uses the value forwarded from the store queue. If not match is found, the load uses the value from the data cache.

Conventional store queues are implemented using content associative memory (CAM), which supports the search functionality required. The CAM is combined with age logic that determines which matching store is the youngest store older than the executing load. As with the issue queue, the use of a CAM prevents the store queue from scaling to large

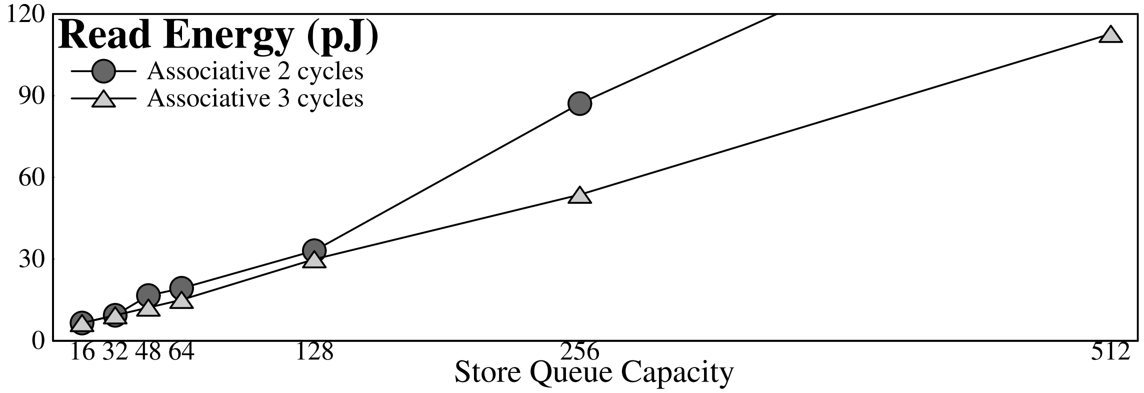


Figure 2.7: Energy scaling of the store queue.

sizes. However, the latency constraints of the store queue are somewhat more relaxed than those of the issue queue. The access latency of the store queue (the CAM plus the age logic) must fit within the data cache access latency, which is typically three or four clock cycles—in this dissertation, we assume a three-cycle data cache. The store queue in a Core i7 processor has 32 entries, limiting the window to 32 stores which have not yet completed to the data cache.

Physical scalability. Figure 2.7 shows the per-read (search) energy reported by CACTI for the CAM portion of the associative store queue. An S-entry store queue is modeled as an S-word 12-bit CAM with a 104-bit data array, 1 indexed write port, 1 indexed read port, and 1 associative read port. Each 116-bit entry includes a 48-bit physical address—the low-order 12-bits are searched[1]—a 64-bit value, a 3-bit size field, and a valid bit. CACTI does not model age logic. To account for the latency of the age logic across a reasonable range of possibilities, two models are used. In the first model (shown by the top line, marked with circles), the CAM is limited to designs which fit into two clock cycles. This model assumes a full cycle for the age logic. The second model (shown by the middle line, marked with triangles) constrains the designs to three clock cycles. This model assumes zero-latency age logic. If the actual latency for the age logic is between zero and one cycle, then the actual energy curve for the store queue will fall between these two.

[1] This allows store queue access in parallel with DTLB access.

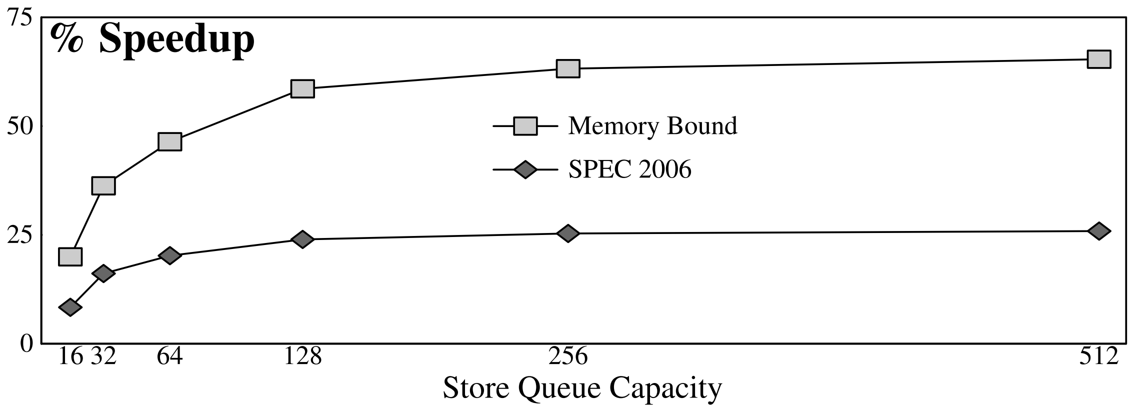


Figure 2.8: Performance impact of the store queue. Other window structures are infinite.

Performance impact. Figure 2.8 shows the performance impact of varying the store queue size when the other window structures are infinite. The top line (marked with light grey boxes) shows the geometric mean speedup for the memory-bound programs. The bottom line (marked with dark grey diamonds) shows the geometric mean speedup for all of SPEC. Memory-bound programs need at least a 128-entry store queue to fully utilize a large window. As Figure 2.7 shows, this requires a 3–5× increase in the per-read energy of the store queue.

Other approaches. Prior research has attacked the scalability of a conventional store queue in various ways. Store Queue Index Prediction (SQIP) [74] replaces the associative store queue with an indexed store queue. SQIP is discussed in detail in Section 2.7.2. Other techniques remove the store queue entirely [75, 78]. Other techniques reduce the number of entries searched in an associative design [7, 57, 73].

2.1.4 Load Queue

In many out-of-order processors, store-to-load forwarding is performed speculatively. A load may execute before the forwarding store, causing a *memory ordering violation*. Memory ordering violations can also occur due to stores from other threads, potentially on other cores. When either form of violation occurs, the processor must detect the situation, squash the incorrect load, and re-execute it.

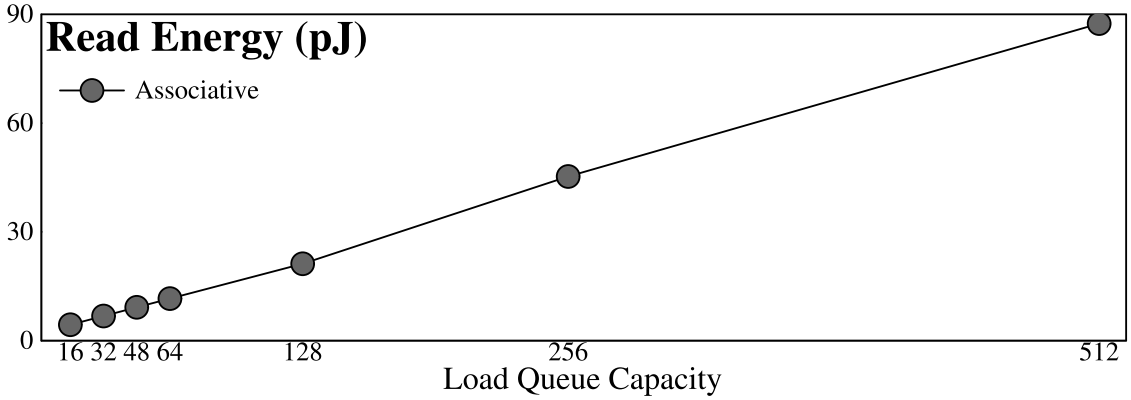


Figure 2.9: Energy scaling of the load queue.

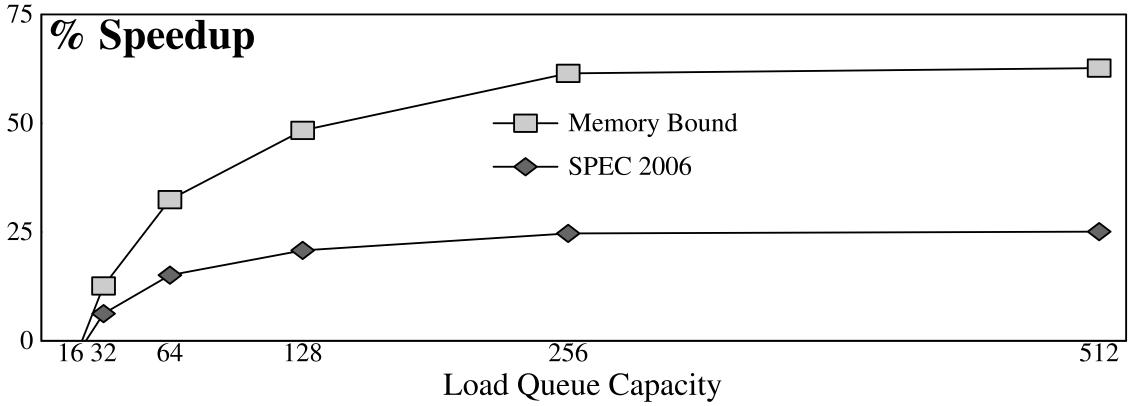


Figure 2.10: Performance impact of the load queue. Other window structures are infinite.

To support memory ordering violation detection, conventional out-of-order processors track the addresses of all in-flight loads in a program-order associative load queue (*i.e.*, a CAM). Executing stores and invalidations due to stores on other processors trigger load queue searches to find any load which may have executed prematurely and read an incorrect value. Unlike the store queue and issue queue, latency is not critical for load queue search.

Physical scalability. Figure 2.9 shows the energy scaling of an associative load queue. An L-entry load queue is modeled as an L-word 48-bit CAM with a 4-bit data array, one read port and one write port. Each entry contains a 48-bit physical address, a 3-bit size field, and a valid bit. As the load queue search latency is not critical, these are modeled with no specific latency constraints. Access latencies range from 2–4 clock cycles.

Performance impact. Figure 2.10 shows the performance impact of scaling the load queue when all other window structures are infinitely large. A 256-entry load queue is needed to maximize performance for the memory-bound programs. A load queue this large uses approximately 5× as much energy as the baseline 48-entry load queue.

Other approaches. Prior work has attacked the scalability of the load queue primarily by re-executing loads in-order prior to commit [12, 67, 68]. One of these approaches—Store Vulnerability Window—is discussed in detail in Section 2.7.1.

2.1.5 Load Latency Tolerance: Virtually Scaling the Window

The high-level idea of load latency tolerance is to *virtually* scale the issue queue and physical register file. Load latency tolerance designs remove miss-dependent instructions from these key window resources to allow younger instructions to enter the pipeline and execute. When a miss returns, the instructions which depend on it are re-injected into the pipeline—re-acquiring issue queue entries and physical registers—and re-execute.

The remove-and-re-inject technique works well for the issue queue and physical register file, but does not work for the load and store queues. This technique works for the issue queue because the issue queue is simply a scheduling structure and the miss-dependent instructions are known not to be ready. The physical register file, however, contains the register state of the program, so releasing physical registers early removes the instruction-granularity register state which processors typically maintain. Instead, a processor can track register state at a coarser granularity using register checkpoints. For memory state—tracked in the load and store queues—such an exchange is more difficult to make. Replacing instruction-granularity register state with checkpoints works for four reasons. First, the number of registers is small. Second, register values are not vulnerable to actions from other threads and processes. Third, the register checkpoint captures all dependences from the point at which it was taken to younger instructions. Finally, register dependences are explicit, meaning that independent miss-independent inputs can be captured so that re-execution is decoupled from the main program. Memory state, however, has exactly the opposite behavior of register state in each of those dimensions. First, memory state is large. Second, loads are vulnerable to stores from other threads. Third, a finite-sized snapshot cannot capture all data dependences between stores and younger loads. Finally, store-to-load dependences are only known when the address of the load and communicating store are both known, making it impossible to capture inputs in the presence of loads with unknown addresses.

In this dissertation, we define a *true load latency tolerant design* as one that incorporates scaling for all four window structures (the issue queue, the physical register file, the load queue, and the store queue), and retains miss-independent work completed under a miss, exploiting ILP as well as MLP. Scaling techniques for all four structures must allow them to accommodate a large enough window to avoid dispatch stalls under long latency misses, without decreasing the clock frequency or drastically increasing energy consumption. These techniques may include virtual scaling—removal and re-injection to achieve a larger effective size—or replacement of non-scalable associative structures with scalable indexed structures.

Several prior works make important steps towards improving performance under long-latency cache misses, but not all of them qualify as true load latency tolerant designs—some of these works either only address some aspects of the design, or only obtain part of the benefits of load latency tolerance. Even among those designs that do qualify as true load latency tolerance, some provide the ability to tolerate dependent long latency misses—miss-dependent loads which are themselves long latency misses—while others do not.

Prior load latency tolerance work varies in its terminology. The following discussion of this prior work uses BOLT's terminology for consistency and simplicity.

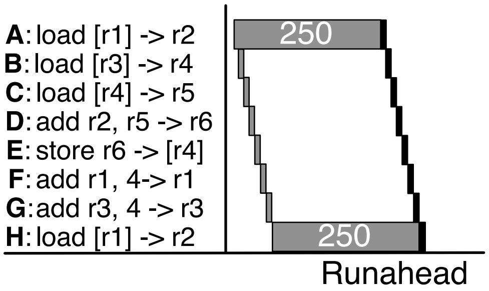


Figure 2.11: Execution of the example code sequence under Runahead Execution. Grey boxes indicate executions in Runahead mode which are discarded. Black boxes indicate actual executions.

2.2 Runahead Execution

An out-of-order processor uses Runahead Execution [20, 54] to expose MLP in the presence of last-level cache misses. When a pending last-level cache miss reaches the head of the re-order buffer, Runahead checkpoints architected register state, and begins Runahead execution mode. In this mode, loads which miss the last level cache undergo a pseudo-execution which produces "poison" rather than an actual output value. This poison is indicated by an extra bit in the destination register which indicates the value is not known. The pseudo-execution of the missing load causes the load to release its issue queue entry and wake up dependent instructions as if it had executed. When dependent instructions read their input values, they ingest poison, pseudo-execute, and propagate poison to their own outputs. Runahead mode retirement processes executed instructions in program order, removes them from the ROB, and frees their physical registers but does not commit them to architected state—specifically, Runahead stores do not write the data cache.

When the miss that triggered the Runahead episode returns, the processor restores the register checkpoint and re-fetches and re-executes all instructions younger than the miss. Re-execution of the instructions that already executed in Runahead mode is accelerated because Runahead execution initiated parallel last-level cache misses and warmed up the caches. Runahead Execution is not a true load latency tolerant design as it discards all miss-independent work after the miss returns and does not exploit ILP under cache misses.

Figure 2.11 shows an example of Runahead execution. When Runahead encounters A's miss, it checkpoints register state and enters Runahead mode. Runahead mode exposes MLP by executing H's miss. However, it is unable to expose ILP—it discards and re-executes the independent instructions: B,C, F, and G.

2.2.1 Load and Store Queues

Runahead's approach to virtually scaling the load and store queues hinges on the fact that it discards all instructions from Runahead mode. Since Runahead instructions will be discarded, it is acceptable if load ordering violations are not detected for Runahead loads, so Runahead simply removes loads from the load queue when they exit the ROB. For stores, forwarding is performed in a best-effort fashion via a Runahead cache. When Runahead stores exit the ROB, they write the Runahead cache and release their store queue entries. Younger Runahead loads check the Runahead cache to see if there are any matching stores. When a Runahead episode ends, the forwarding cache is cleared. As stores may age out of the Runahead cache, a proper forwarding may be missed. This possibility is acceptable because all loads and stores re-execute when the Runahead episode ends.

2.2.2 Efficiency

If Runahead mode does not expose any MLP, it not only does not help performance, but also wastes energy. To avoid such wastes, Runahead can implement heuristics which predict whether a Runahead episode is likely to expose MLP or not [53]. One heuristic avoids useless periods of Runahead by learning which static loads have historically exposed MLP and which have not. Another heuristic avoids overlapping periods of Runahead by tracking how many instructions were pseudo-retired during Runahead mode, and preventing the processor from re-entering Runahead mode until at least that many instructions have been retired conventionally. These techniques occasionally harm Runahead's performance, but greatly improve its overall energy efficiency (see Section 5.3).

2.3 Checkpointed Early Load Retirement and Checkpoint Assisted Value Prediction

Two similar designs: Checkpointed Early Load Retirement (CLEAR) [42] and Checkpoint Assisted Value Prediction (CAVA) [14] use value prediction to tolerate long load latency. When a load misses the last-level cache, a predictor is used to guess the output value. The load then binds this value and dependent instructions execute normally, releasing their issue queue entries.

CLEAR and CAVA both couple value prediction with *speculative retirement* to scale the physical register file. Speculative retirement is not traditional retirement made speculative—instructions are not made globally visible and then pulled back. Instead it is a more resource efficient way of buffering speculative instructions. Specifically, it uses register checkpointing to buffer a large number of instructions without explicitly representing the register output of each instruction. In CLEAR and CAVA, speculative retirement begins when a value-predicted load reaches the head of the ROB. The processor checkpoints architected register state and begins retiring instructions speculatively. Speculatively retired instructions exit the processor and release their physical registers allowing younger instructions to enter behind them. When value-predicted load misses return, the actual output value is compared against the predicted value. If all predictions are correct, then the checkpoint is released, making speculative retirement non-speculative. If any mismatch occurs, then speculation is aborted to the checkpoint. Both CLEAR and CAVA use more than one checkpoint to reduce the number of instructions squashed on a mis-speculation.

As mis-speculations would result in squashes of many independent instructions, it is only beneficial to apply value prediction to those loads whose value is very predictable. Unfortunately, most loads have difficult to predict values (CLEAR reports that only 24.3% (integer) and 39.6% (floating point) of loads are high confidence [42]), meaning that value prediction is not a general enough solution for a processor to achieve true load latency tolerance.

2.3.1 Load and Store Queues

CAVA uses a *transactional cache* to buffer speculatively retired loads and stores. A transactional cache is a data cache which accepts speculative writes and can abort them if needed. The transactional cache handles the verification of loads relative to stores from other threads by tracking which cache lines have been speculatively read and signaling a violation if one of these lines must be evicted before the read is made non-speculative. The transactional cache works well with value prediction-based designs like CAVA, but is ill-suited to other forms of load latency tolerance which re-execute miss-dependent instructions. In designs which depend on the re-execution of miss-dependent instructions, a re-executing load may need a value which has been overwritten in the cache, preventing the load from executing properly.

CLEAR assumes large load and store queues, but notes that other solutions exist, including a transactional cache.

2.4 Waiting Instruction Buffer

Waiting InstructionBuffer (WIB) [44] differs from Runahead in that it attempts to obtain ILP as well as MLP under misses by retaining miss-independent work. To accomplish this, WIB places miss-dependent instruction in a new structure[2] called the waiting instruction buffer as they pseudo-execute. The process of removing the miss-dependent instructions and buffering them for later re-execution is termed *slicing out*. Unlike the issue queue, the WIB does not support associative search and is not latency critical, meaning it can be made large. When a long latency miss returns, it and its dependent instructions are *sliced in*—re-injected into the issue queue for re-execution.

[2] It is possible to simply let the ROB double as the WIB since there is a one-to-one correspondence between the structures. The WIB paper describes it as a separate structure, so that is how it is described here.

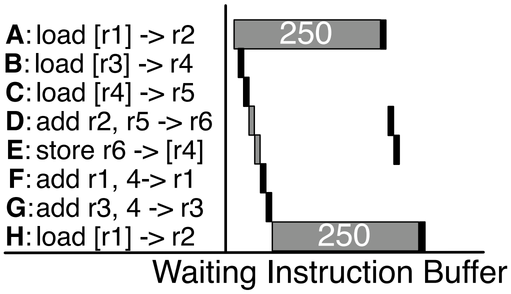


Figure 2.12: Execution under WIB. Black boxes represent actual executions, while grey boxes represent pseudo-executions to propagate poison.

Figure 2.12 shows an example of execution under WIB. Load A and its dependents, D and E undergo pseudo-execution (grey box) to propagate poison and exit the issue queue. Removing these instructions allows G and H to enter the pipeline and execute. Executing H exposes MLP, improving performance. WIB does not need to re-execute independent instructions B,C, F, or G after A's miss returns, providing some performance benefits from ILP.

WIB's pseudo-execution/poison propagation allows the issue queue to scale by removing miss-dependent instructions. However, the issue queue is only part of the problem—WIB does not scale the physical register file or load and store queues, and simply assumes they are very large. Since WIB only addresses part of the problem it is not, by itself, a complete load latency tolerant design.

2.5 Continual Flow Pipelines

The first true load latency tolerant design is Continual Flow Pipelines (CFP) [76], which includes solutions for scaling the issue queue, register file, and load/store queues. CFP's issue queue solution is similar to WIB—miss-dependent instructions propagate poison, exit the issue queue, and are placed in a *slice buffer*, which is similar to the WIB. Repeating the running example execution-over-time for CFP would look identical to the execution under WIB in Figure 2.12. The primary difference is that CFP incorporates a scalable register management algorithm, and addresses the load and store queue issues.

2.5.1 Register management: Checkpoint Processing and Recovery

CFP scales the register file by replacing conventional, ROB-based register management with Checkpoint Processing and Recovery (CPR) [2]. CPR operates at the granularity of checkpoints, which are created at rename. Instructions are committed one checkpoint at a time and recovery is permitted only to checkpoints. CPR leverages the recovery restriction to aggressively reclaim physical registers *between* checkpoints. At any point, CPR holds only those registers that are named in the active map-table or any map-table checkpoint, and those that are read or written by any un-executed in-flight instruction [52].

CPR reclaims registers out-of-order—a register is typically freed when the last instruction that reads it executes, and execution proceeds out-of-order. As a result, CPR cannot track free registers using a queue. Instead, it uses reference counting. Each register is associated with a reference count—a count of zero indicates the register is free.

CPR's reference counting algorithm is simple. A register's reference count is incremented when it is allocated to an instruction and written to the map-table, and decremented when it is overwritten and disappears from the map-table. It is incremented when an instruction that reads or writes the register is dispatched to the issue queue and decremented when that instruction executes or is squashed. The reference count of any register that appears in a map-table checkpoint is incremented when that checkpoint is created and decremented when the checkpoint is freed.

A simple and efficient implementation of reference counting uses "unary" matrices [28, 69]. In a reference count matrix, each column represents a physical register and each row represents a resource that can hold a physical register, *e.g.*, an issue queue entry or a checkpoint. A bit in the matrix is 1 if the given resource references the given physical register. A bitvector-style free-list is constructed by ORing together all the bits in a column. Registers are allocated from this free list using encoders.

While the reference counting machinery is logically a single matrix, it is physically comprised of multiple different structures. Some rows (*e.g.* those corresponding to the checkpoints) of the matrix do not need frequent manipulation of individual bits—only copy, restore, or clear operations. These rows are actually stored as rows in an SRAM matrix. Other rows (*e.g.* the one corresponding to the rename map table) require manipulation of individual bits and are implemented as latches.

2.5.2 Slice Management and Processing

While CFP's slice buffer is similar to a WIB, it has two major differences. First, it also captures miss-independent register inputs. This difference is significant as it decouples the miss-dependent slice from the rest of the program. This decoupling means that miss-dependent instructions can release their input registers as if they had executed, allowing CPR to reclaim them.

The second difference is that CFP's slice buffer is maintained in execution order, while WIB is maintained in program order. In CFP, instructions are allocated slice buffer entries as they slice out of the issue queue. A direct benefit of this allocation is that the slice buffer only needs space for miss-dependent instructions, making it smaller than WIB. The downside to tracking the slice in execution order arises when outstanding misses return and instructions must be re-injected into the window to re-execute.

When CFP re-injects instructions, it does so by traversing the slice buffer and re-renaming and re-dispatching instructions. Instructions must be re-renamed—including allocation of a new destination physical register, as well as re-mapping inputs—as the original registers have likely been reclaimed by the CPR substrate. The fact that the instructions are tracked in execution order means that the slice buffer cannot represent register dependences in terms of logical register names and must use physical register names instead. Consequently, conventional logical-to-physical renaming cannot be performed. Instead, CFP must introduce new physical-to-physical renaming hardware. In physical-to-physical renaming, the physical-to-physical map table has as many entries as the processor has physical registers. An instruction's input registers are renamed by indexing this map table with the physical register numbers that it received when it was originally renamed. A new destination register is allocated, and entered into the map table at the index corresponding to the destination physical register number originally assigned at rename.

Physical-to-physical renaming has several disadvantages. First, it requires the large physical-to-physical map table. Second, it holds registers live for artificially long durations. Registers are only clobbered in the physical-to-physical map table when the slice buffer traversal encounters a younger instruction which wrote the same *physical* register. It is possible for the slice map table to hold every available physical register, requiring a squash to break the deadlock[3].

2.5.3 Load and store management

CFP scales the load and store queues through the use of hierarchy. For the load queue, CFP uses a large set-associative load queue in conjunction with a conventional fully-associative load queue. In this hierarchical design, the youngest loads are placed in the fully-associative queue. When the fully-associative queue's capacity is exhausted, the oldest load in the queue is moved to the set-associative load queue.

Allocating an entry in the set-associative load queue requires the load's address to be known to determine the correct set. If the appropriate set is full, then the load remains in the fully-associative load queue, and dispatch stalls. If the load's address is not known, it is simply removed from the fully-associative queue and dispatch continues. In this situation, an entry in the set-associative queue is allocated when the load's address becomes known. At that time, if the appropriate set is full, the load must wait until older loads commit, making space available. If the stalling load is in the oldest checkpoint, this will never happen, so deadlock is avoided by squashing checkpoints until either the required set has space, or the load in question is itself squashed.

The problem with CFP's set-associative load queue design is that it can significantly inhibit performance. As the window grows and more loads occupy the set-associative load

[3] CFP proposes to reserve a pool of physical registers for re-renaming to avoid this problem. Unfortunately, this approach results in too many reserved registers to be practical.

queue, the liklihood of set-conflicts increases. These problems are magnified with miss-dependent loads of unknown addresses. These loads will only re-execute and determine their address after the miss they depend on returns. At this point, many younger loads will have executed, causing a high probability of a full set. If the set is in fact full, these old loads likely require squashes to make forward progress—not only hurting performance, but also wasting energy.

CFP also uses hierarchy to manage stores, with two proposed designs for secondary store queues. The first is a large associative queue with an access latency that matches that of the L2 [2]. To avoid accessing the secondary queue on every load—and effectively increasing load latency to L2 latency—the secondary queue is guarded by an address-indexed Bloom filter called the Membership Test Buffer (MTB). Load latency elongates to L2 latency only on an MTB "hit." This design performs well. Its disadvantage is the area and dynamic power cost of the secondary queue.

The follow-on design replaces the large associative store queue with two structures which activate on a last-level cache miss [26]. While the miss is pending, miss-independent stores drain into a small forwarding cache [54], supporting forwarding to miss-independent loads. In parallel, *all* stores drain into an age-ordered Store Redo Log (SRL). When the miss returns, the forwarding cache is flushed and the miss slice is re-injected into the window. Miss-dependent stores re-acquire first-level store queue entries for store-load forwarding within the slice. When they complete, they write their values to their pre-allocated positions in the SRL. When the miss slice completes execution, dispatch at the tail of the window resumes and the SRL drains to the data cache in program order. An address indexed Bloom filter—called the Loose Check Filter (LCF)—supports limited forwarding from the SRL during the draining process. Loads that cannot forward using the LCF stall until draining completes. SRL has both performance and algorithmic complexity disadvantages. Performance-wise, it provides a large store queue only in the shadow of a last-level cache misses and not in the general case. Complexity-wise, although the SRL, LCF, and forwarding cache are physically simple, their associated management algorithms—which are required to detect forwarding violations—are complex.

2.6 KILO- and Decoupled KILO-Instruction Processors

KILO-Instruction Processor [19] is another load latency tolerance design. KILO scales the issue queue in a manner similar to WIB and CFP—it buffers miss-dependent instructions in a non-associative structure until the miss they depend on returns. KILO calls this structure a Slow Lane Issue Queue (SLIQ).

KILO scales the register file using a combination of *virtual registers* and reference counting. At rename, logical registers are mapped to virtual registers—a namespace larger than the set of physical registers. The virtual register name is then re-mapped to a physical register when the instruction is actually ready to execute. KILO reference counts these virtual register names to determine when they can be reclaimed, and reclaims the underlying physical register at the same time. KILO does not consider the implementation details of this register management scheme. If this scheme were implemented using a reference counting matrix, the matrix would be quite large. It would need columns equal to the number of virtual register names, and would need one row per slow lane issue queue entry as well per conventional "fast lane" issue queue entry.

For the load or store queues, KILO cites various prior proposals, including those used by CFP, and states that any would be satisfactory.

2.6.1 Decoupled KILO-Instruction Processor

KILO's register management scheme is quite complex. The follow-on design, Decoupled KILO-Instruction Processor (D-KIP) [60] addresses this problem by chaining execution between two different processors. In D-KIP, instructions start in the Cache Processor. The Cache Processor is a conventional out-of-order processor, except that miss-dependent instructions are forced out of its ROB after a certain number of cycles. As miss-dependent instructions leave the Cache Processor, they are placed into the Long Latency Instruction Buffer (LLIB)—a FIFO queue which chains the out-of-order Cache Processor to an in-order Memory Processor. LLIB instructions capture their ready register input values as they exit the Cache Processor, so that the Memory Processor's execution is self-contained. D-KIP maintains precise register state in the Cache Processor using a series of register checkpoints. These checkpoints contain register value (not mappings), and miss-dependent instructions whose values appear in them are flagged to update the checkpoint when they execute on the Memory Processor. D-KIP scales the load and store queues using hierarchy. However, it arranges its hierarchy differently from CFP, using miss-dependence or independence rather than age to determine hierarchy level [59].

On disadvantage of D-KIP is excessive copying of register values. Another disadvantage is the complexity of implementing register checkpoints that support incremental updates. A third disadvantage is that the only value communication from the in-order Memory Processor to the out-of-order Cache Processor comes in the form of restoring register checkpoints. Consequently, instructions executing in the Cache Processor may *appear* to be miss-dependent and re-execute on the second processor, even if the miss they depend on has already returned and their input values are known in the Memory Processor. Delaying this communication not only causes extra re-executions, but also can increase the branch mis-prediction penalty. Neither KILO nor D-KIP is capable of tolerating the latency of dependent load misses.

2.6.2 Scalability of Load Latency Tolerant Designs

The slice buffer of load latency tolerant designs like CFP has two scalability benefits compared to the issue queue and register file. The first benefit is that it only needs to hold miss-dependent instructions. Figure 2.13 shows the percentage of instructions which are miss-dependent (*i.e.*, ever poisoned).

The second advantage is the physical scalability of the slice buffer structure compared to the issue queue. The slice buffer is an indexed structure whereas the issue queue is

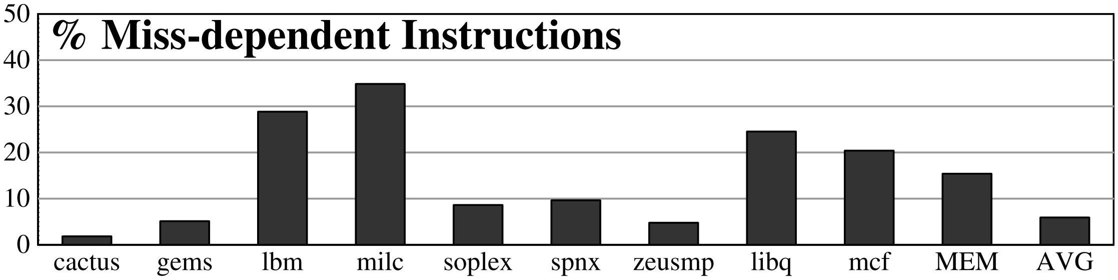


Figure 2.13: Percentage of instructions which are miss-dependent.

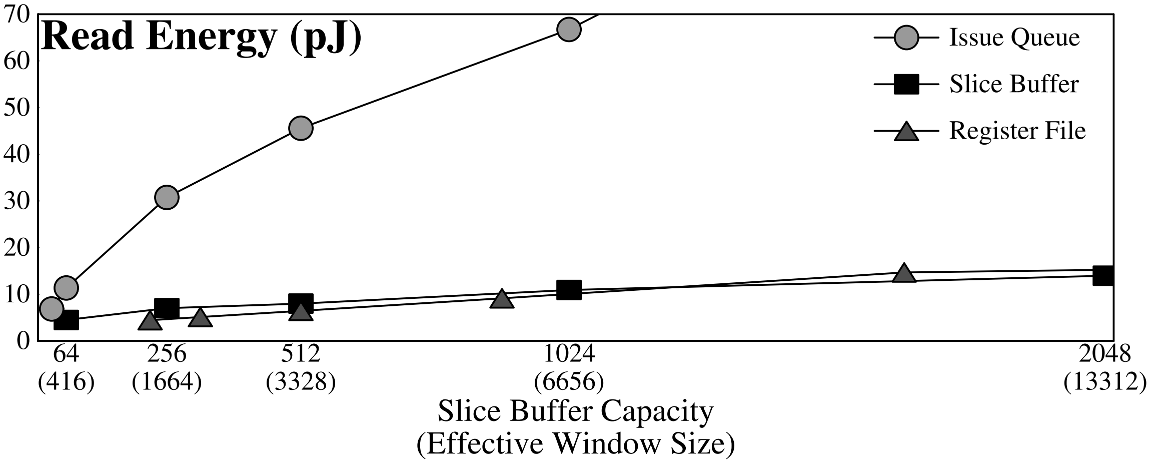


Figure 2.14: Scalability of a slice buffer versus the scalability of an issue queue and register file. In addition to the size of the structures, the x-axis is labeled with the effective window size given by that slice buffer capacity, assuming 15% of instructions are miss-dependent.

associatively searched. Figure 2.14 shows the per-instruction read energy of the slice buffer and the issue queue[4]. An S-entry slice buffer is modeled as four $S/4$-word 156-bit RAMs with one read port and one write port. Each entry holds a 64-bit instruction, a 64-bit (captured) data value, an 8-bit load/store queue index, an 8-bit physical register number, and a 12-bit instruction sequence number. The x-axis shows the capacity, but also lists the effective window size for the corresponding slice buffer size, assuming the average percentage of miss-dependent instructions (15.4%).

A 256-entry slice buffer requires about the same energy per read as a 36-entry issue queue requires per search. However, the slice buffer is only accessed by miss-dependent

[4] This graph ignores the fact that many of these issue queue sizes require changing the voltage or clock frequency, or using 2-cycle scheduling, as discussed in Section 2.1.1.

instructions. A 256-entry issue queue consumes significantly more energy for *all instructions*.

For the same number of entries, the slice buffer and register file require about the same read energy. Here, the key scalability difference is that the slice buffer only needs to hold miss-dependent instructions, whereas the register file must hold all in-flight instructions. Additionally, the slice buffer is not in the critical execution loop, so its access latency is not as important.

2.7 Scalable Load and Store Queues

The non-scalability of conventional associatively searched load and store queues has been addressed in work unrelated to latency tolerance as well. Two schemes proposed in the ROB-based context, Store Vulnerability Window (SVW) [67, 68] and Store Queue Index Prediction (SQIP) [74] provide scalable designs for the load and store queues respectively. Other schemes to scale the load and store queues have been proposed in the ROB-based context [7, 12, 57, 72, 73, 75, 78, 80], however, these two work well together and are incorporated in BOLT's design.

2.7.1 Store Vulnerability Window (SVW)

An alternative approach to conventional associative search is to re-execute loads in-order prior to commit, detecting a violation when the values obtained by (out-of-order) execution and (in-order) re-execution differ [12]. Re-execution is cheaper than associative search for large windows. It also does not signal spurious violations. Finally, it can detect violations in other forms of load speculation, not just store-load ordering speculation. Its primary drawback is that it consumes significant data cache bandwidth.

Store Vulnerability Window (SVW) is an address-based filter that reduces the data cache bandwidth requirements of load re-execution [67, 68]. The basic idea is that a load can skip re-execution if no store has written to its address in a sufficiently long time. SVW identifies dynamic stores using monotonically increasing sequence numbers (SSNs). A store's store queue position is the low order bits of its SSN. A small address-indexed table called the Store Sequence Bloom Filter (SSBF) contains the SSNs of the youngest store to write to each (hashed) address. An extended commit pipeline processes executed stores and loads in order. Stores write their SSNs at the SSBF index corresponding to their address. The global register $\text{SSN}_{\text{verify}}$ tracks the SSN of the youngest store to have updated the SSBF. Loads read the SSBF using their address and determine whether they need to re-execute based on the SSN they find. The re-execution test is specific to the type of speculation the load undergoes.

2.7.2 Store Queue Index Prediction (SQIP)

Conventional processors perform store-load ordering speculation. A more aggressive form of speculation is forwarding speculation in which the processor speculates for every load the precise identity of the forwarding store. Forwarding speculation supports a scalable store queue which is both age-ordered and non-associative [74], avoiding the scalability problems of fully-associative designs and the set-overflow problems of address-indexed ones. Forwarding speculation can be performed with very high accuracy (99.9% for most programs) because only a small fraction of loads forward, and because most forwarding behavior is stable and predictable. Forwarding prediction schemes that represent store-load dependences as dynamic store distances [75, 78, 84] mesh with SVW and typically yield the highest prediction accuracies.

2.7.3 SVW/SQIP Example

Figure 2.15 shows a working example of SQIP and SVW. In our notation, data addresses are lowercase letters, instruction PCs are uppercase letters, SSNs and store distances are numbers, and data values are irrelevant. The figure uses three instructions: stores W and X to addresses `b` and `a`, followed by load Z to address `b`. The figure shows four snapshots

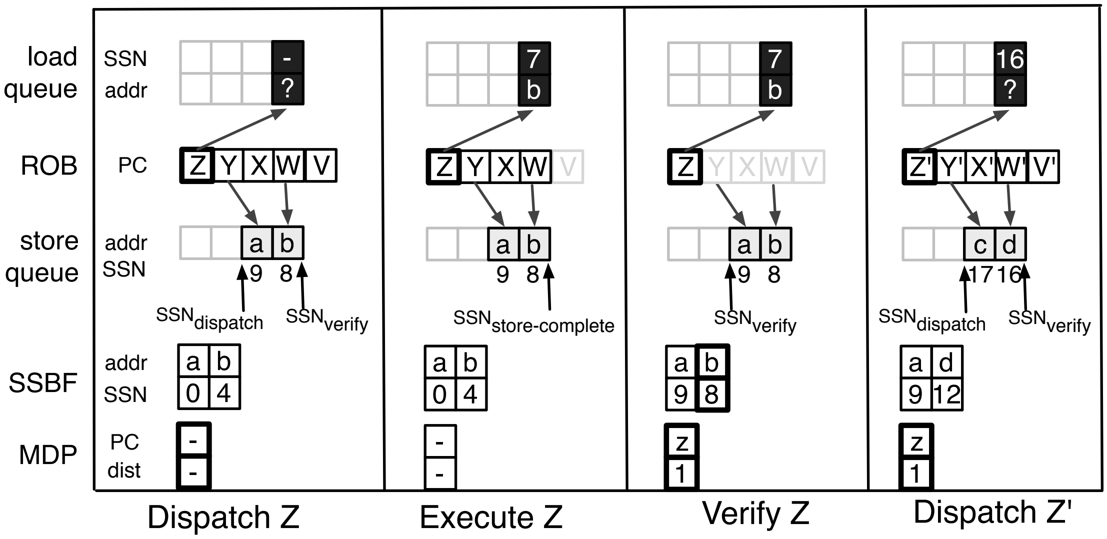


Figure 2.15: Example of SVW and SQIP

in time. Each snapshot shows a ROB (each entry contains a PC), a four entry load queue on top (each entry contains an address and the SSN of the predicted forwarding store), a four-entry store queue (each entry contains an address, the entry's SSN is implicit), two sets (corresponding to hashed addresses a and b) of a direct-mapped SSBF (each entry contains an address tag and an SSN), and a one-entry memory dependence predictor (each entry contains a PC tag and a store distance).

The dynamic instruction stream is tracked by three explicit global pointers. The position of these pointers in the dynamic store stream is marked by SSNs. We have already introduced $\text{SSN}_{\text{verify}}$. Stores older than $\text{SSN}_{\text{verify}}$ have updated the SSBF. They have also transitioned from the store "queue" to the store "buffer" and cannot be aborted. The corresponding instruction stream pointer is P_{verify}. Loads older than P_{verify} are verified and cannot be aborted as well. Global pointer $\text{SSN}_{\text{store-complete}}$ ($\text{P}_{\text{store-complete}}$) tracks the youngest store to write to the data cache. Global pointer $\text{SSN}_{\text{dispatch}}$ ($\text{P}_{\text{dispatch}}$) tracks the youngest dispatched store (instruction).

At T1, (Figure 2.15a) stores W and Y have dispatched (they are older than $\text{SSN}_{\text{dispatch}}$) and executed. Load Z dispatches and consults the memory dependence predictor. There is no matching entry so the load will only access the data cache. At T2 (Figure 2.15b) load Z executes, calculates its address (b), and reads the cache. Load Z is unaware of older store W to address b because it cannot search the store queue. Load Z records the current value of $\text{SSN}_{\text{store}-\text{complete}}$ (7)—it is "vulnerable" to all younger stores (*i.e.*, stores with higher SSNs). At T3 (Figure 2.15c), load Z is verified; notice stores W and Y have updated the SSBF. Load Z reads the SSBF set corresponding to address b and finds a store to address b with SSN 8. As load Z is only safe with respect to SSNs ≤ 7, it must re-execute. Re-execution shows it received the wrong value. Load Z is squashed and the memory dependence predictor is trained to forward from the appropriate store. The distance to the forwarding store is computed as $\text{SSN}_{\text{verify}}$–SSBF[Z.addr] (in this case 9–8=1). At T4 (Figure 2.15), load Z' (a future instance of load Z) predicts a dependence on Y' by computing $\text{SSN}_{\text{dispatch}}$–MDP[Z].

2.7.4 Multiprocessor Issues

Processors that verify intra-thread store-load ordering by snooping the load queue use the same mechanism to verify inter-thread store-load and load-load ordering. Load queue search is triggered by invalidation messages and (for directory protocols and some memory consistency models) local cache evictions as well. How does SVW track these events and "trip" loads that are exposed to them? SVW treats events that would require load queue snooping as "asynchronous stores" from the same thread. It tracks them using a second SSBF (SSBF-MT) which is organized at a cache line granularity. Verifying loads read both the SSBF and the SSBF-MT and re-execute if they hit in either. When an asynchronous store event occurs, SVW writes $\text{SSN}_{\text{store}-\text{complete}}$ +1 in the corresponding SSBF-MT entry. The value $\text{SSN}_{\text{store}-\text{complete}}$ +1 trips all exposed loads that had not been verified when the event occurred. For memory consistency models that disallow store buffers, *e.g.*, sequential consistency (SC). SVW can enforce SC by stalling P_{verify} at any load if $\text{P}_{\text{verify}} > \text{P}_{\text{store}-\text{complete}}$.

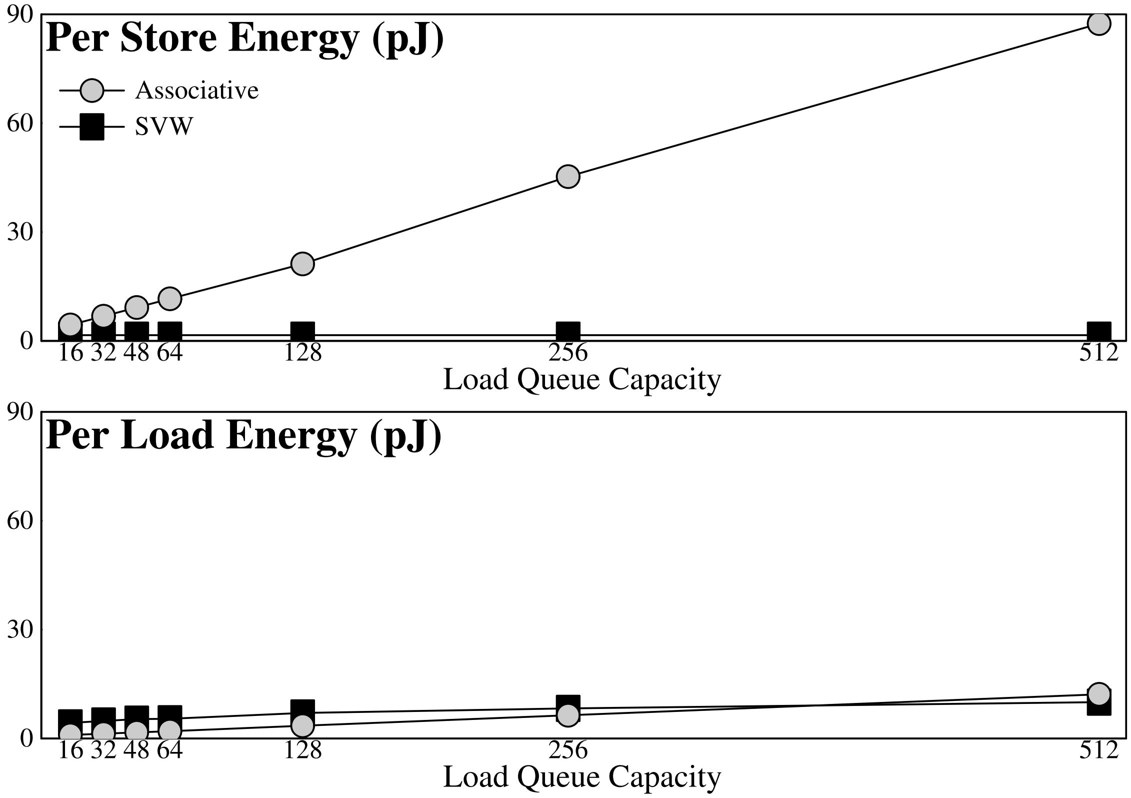


Figure 2.16: Comparison of the energy scaling of an SVW load queue versus the energy scaling of an associative load queue.

2.7.5 Scalability

Load queue. Figure 2.16 revisits the scalability of the load queue for SVW. Because associative and indexed load queues are used in different ways, simply comparing read energy is insufficient. Instead, the top graph shows the energy consumed per store in each design, and the bottom graph shows the energy consumed per load. The SVW load queue modeled here assumes the design used in BOLT (see Section 3.4) and a 128-entry chained store buffer (see Section 3.3). The SVW load queue is modeled with the address and data divided into two physical structures. For an L-entry load queue, the data portion is modeled a L-word 64-bit RAM with one read port and one write port. Each entry holds only the 64-bit data value. The address portion is modeled by two $L/2$-word 52-bit RAMs, each with one read port and one write port. Each entry holds a 48-bit physical address, a

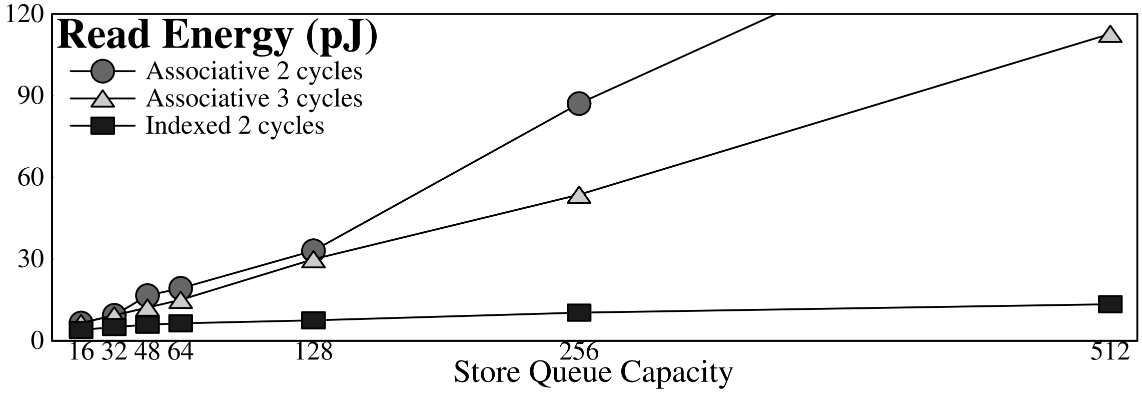


Figure 2.17: Energy scaling of a SQIP store queue versus the energy scaling of an associative store queue.

3-bit size field, and a valid bit. The SSBF is modeled as a 256-word 16-bit RAM with one read port and one read/write port. The SSBF-MT is modeled as an 8-word 16-bit RAM with one read port and one write port. The SMT filter is modeled as a 512-word 16-bit RAM with one read/write port. The re-execution rate is assumed to be 1%.

The per-load energies of both designs are relatively similar. The SVW design consumes slightly more energy per load at the lower sizes due to the cost of accessing the SSBF and re-executing loads, as well as the fact that an executing load must write its address *and data* to the load queue. At larger sizes, the efficiency of an indexed structure overcomes the constant overheads, giving SVW a slight per-load energy advantage over the associative load queue. The more significant factor is the per-store energy. Here, the associative load queue's search energy scales roughly linearly. By contrast, SVW's per-store energy is constant. Stores do not access the load queue at all, instead they update the SSBF, a small RAM whose size does not grow with the window size.

Store queue. Figure 2.17 shows the associative configurations from Figure 2.7, but also adds the read energy for an indexed store queue (bottom line, marked with black boxes). The graph also shows the read energy of an indexed store queue (bottom line, marked with black boxes). The indexed designs do not benefit from relaxing the timing constraints to three cycles.

The indexed store queue's energy curve has a much smaller slope than the associative's. For the indexed store queue, increasing the capacity from 32 entries to 128 entries corresponds to slightly less than a 50% increase in per-read energy. By contrast, the same capacity increase for the associative store queue results in slightly more than a *factor of three* increase for the three cycle design and slightly more than a *factor of five* increase for the two cycle design.

Chapter 3

BOLT Microarchitecture

Better Out-of-order Latency Tolerance (BOLT) is an out-of-order load latency tolerant micro-architecture based on three design principles. First, BOLT should extend a conventional micro-architecture and fall back to conventional execution in the absence of long-latency cache misses. This reduces design effort and ensures that BOLT does not harm the performance of any program it does not actively help. Second, in the presence of a long-latency cache miss, BOLT should scale all window structures—issue queue, physical register file, load queue, and store queues—and do so in a structurally efficient way. In BOLT, structural efficiency means the use of a single unified physical register file, large, flat, non-associative load and store queues, and the use of the additional map tables normally used to support simultaneous multithreading to re-rename deferred instructions. The latter requires that BOLT use a slice buffer that tracks miss-dependent instructions in program order. Third and finally, BOLT should support dynamically efficient load latency tolerant execution. In BOLT, this means making both slice deferral and re-execution as non-speculative as possible to avoid both expensive and unnecessary squashing and to eliminate the need to re-perform various execution verification tasks. It also means supporting optimizations that minimize the number of instructions that defer and re-execute in response to any miss and miss-return, respectively. This particular aspect is described in detail in the next chapter.

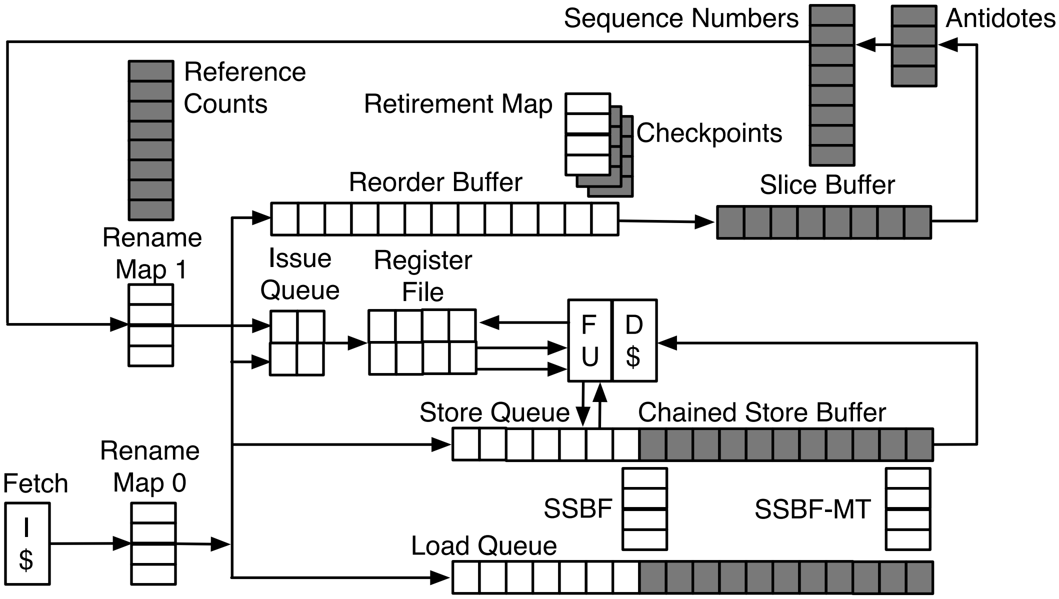


Figure 3.1: Structural diagram of BOLT. BOLT-specific structures are shaded grey.

3.1 BOLT Overview

To achieve these goals, BOLT draws from three previous designs: WIB, CFP, and D-KIP. At a high level, BOLT resembles D-KIP in that it divides the dynamic instruction window into two execution regimes. One regime executes miss-independent instructions and "defers" the execution of instructions that depend on long-latency cache misses. A second regime trails the first and essentially cleans up by re-executing deferred instructions as the misses they depend on return. However, the similarity largely ends here. D-KIP implements the two regimes by physically chaining together two different processors, a conventional out-of-order "cache" processor executes miss-independent instructions and an in-order processor re-executes miss-dependent instructions. The two processors have separate execution engines and separate register spaces (which are kept coherent). In contrast, BOLT *logically* chains together a WIB processor to execute miss-independent instructions with a CFP processor that re-executes miss-dependent instructions, but *physically* implements no chaining at all. BOLT unifies the register spaces of the two regimes in a single physical register file and executes both miss-independent and miss-dependent

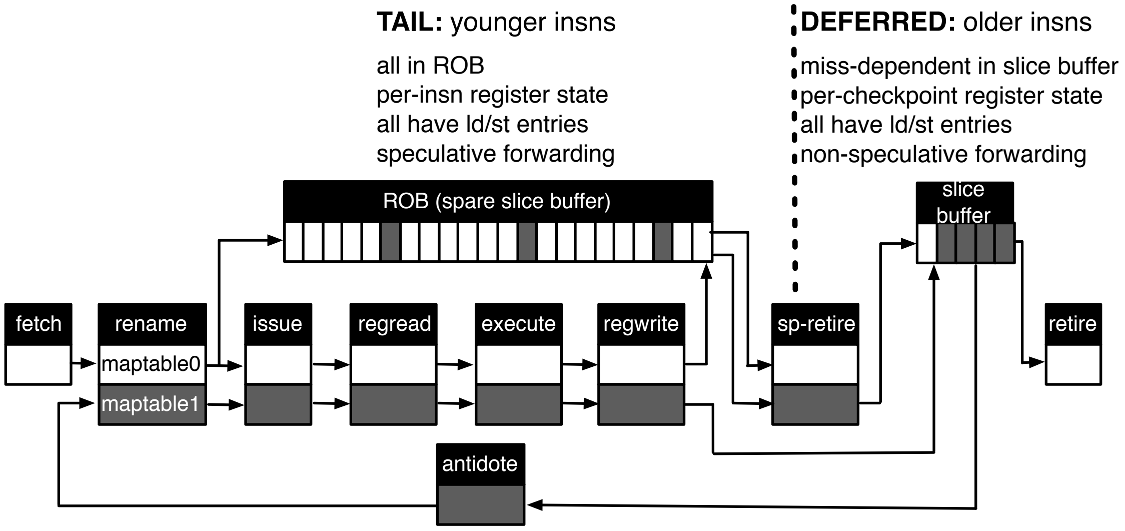


Figure 3.2: BOLT Pipeline.

instructions in the out-of-order engine in parallel, using SMT.

Figure 3.1 shows a high-level structural diagram of BOLT. White structures exist in a conventional out-of-order core. Structures shaded grey are new to BOLT. Only the execution paths used when BOLT is active are shown. Specifically, the front end paths used by the second thread when multi-threading is active are omitted.

Figure 3.2 shows the pipeline stages in a BOLT processor. Here, shading distinguishes miss-independent instructions (white) from miss-dependent instructions (gray). Re-executing instructions pass through a BOLT-specific *antidote* stage (which is discussed in Chapter 4) prior to re-renaming.

In BOLT, the distinction between the two execution regimes is primarily logical rather than physical so the following discussion distinguishes between two dynamic instruction stream regions, *tail* and *deferred*, rather than two processors.

Tail region. The portion of the window containing younger instructions is called the tail region. In this portion of the window, instructions are tracked using a conventional re-order buffer (ROB). As in a conventional out-of-order processor, tail region execution is characterized by branch and load speculation, instruction granularity register management, and the ability to precisely recover to any instruction on a mis-speculation. For loads, BOLT uses SQIP [74] to perform speculative store-to-load forwarding and SVW filtered load re-execution [67] to verify both this speculation and speculation relative to stores from other threads. BOLT uses SQIP and SVW because they use indexed store and load queues, respectively, which can be made large enough to accommodate the window sizes required for latency tolerance.

In the absence of long-latency misses, the tail region is the only active region and all instructions execute as they do in a conventional out-of-order processor. When a long-latency miss occurs, the miss and its dependent instructions *slice out*—capture ready input values, pseudo-execute, propagate poison, and release their issue queue entries. However, they do not immediately release their registers and enter the slice buffer as they do in CFP. Instead, they use the ROB as a sparse slice buffer and record captured miss-independent values in their slice buffer entries as in WIB.

Speculative retirement. To prevent conventional ROB-based register reclamation from blocking tail execution in the presence of a long-latency miss, BOLT employs speculative retirement. If the oldest instruction in the ROB is a pending cache miss, BOLT checkpoints register state at the head of the ROB and begins a speculative retirement episode. Backed by this checkpoint, BOLT speculatively retires instructions by removing them from the ROB and releasing their over-written registers. Speculative retirement occurs in program order and marks the boundary between the tail and deferred regions. Speculative retirement blocks at uncompleted miss-independent instructions and simply "discards" completed miss-independent instructions. It *defers* miss-dependent instructions by copying their ROB entry—complete with any captured miss-independent input—to a slice buffer. The combination of WIB like actions in the tail region and speculative retirement produce a slice buffer that tracks miss-dependent instructions in program order. In the tail region, this slice buffer is sparse with miss-dependent instructions co-mingled with miss-independent ones. In the deferred region, the slice buffer is dense explicitly containing only miss-dependent instructions. The speculative retirement stage performs the compaction.

BOLT's speculative retirement stage also uses SVW [67] to verify tail loads relative to older stores from the same thread. Combined with non-speculative forwarding for re-executing deferred loads, this means that deferral—the identification of instructions that depend on a pending miss and must be copied into the slice buffer—is itself non-speculative. This simplifies the implementation and supports dynamic efficiency by reducing squashes and elminating the need to re-verify loads relative to same thread stores after speculative retirement.

Stalling speculative retirement on poison branches. One potential source of dynamic inefficiency in BOLT comes from the checkpoint overhead incurred when a miss-dependent mis-predicted branch causes a squash to an older checkpoint. This penalty is the tradeoff for scaling the register file to accommodate a large window, but the extra registers are not always needed before the branch resolves. BOLT employs a simple policy to reduce these squashes. When speculative retirement encounters a miss-dependent branch, it stalls unless physical registers or ROB space are exhausted. If both resources are available, there is no benefit to speculatively retiring the branch—only the risk that it might be mis-predicted. If either resource is exhausted, then BOLT speculatively retires the branch—stalling on the chance that it *might* is not likely to be the best course of action.

Deferred region and slice re-execution. The deferred region resembles CFP. Only miss-dependent instructions are explicitly represented and register state is tracked in widely separated checkpoints. The deferred region does not support instruction-granularity precise state and any mis-speculation requires aborting to the checkpoint and discarding many instructions that are older than and independent of the mis-speculation. Fortunately, instructions in the deferred region are less speculative than those in the tail region—specifically, instructions are speculative only relative to miss-dependent branches and loads are speculative only relative to stores from other threads—so deferred region mis-speculations are rare.

When a miss returns, deferred instructions that depend on it are re-renamed, re-dispatched to the window where they re-acquire issue queue entries, and re-execute.

BOLT's program-order slice buffer combined with the capture of miss-independent input values make miss slices look like self-contained program fragments. This allows BOLT to treat slices as small software threads and to execute them in parallel with the main program thread on a Simultaneous Multi-Threading (SMT) processor [81]. The most important aspect of treating a slice as a software thread is the ability to re-rename it using a conventional map table. This represents a significant structural efficiency advantage over CFP, whose execution-order slice buffer requires new physical to physical renaming hardware (see Section 2.5.2). Slice re-execution does differ from software multi-threading in two subtle ways. First, the re-executing slice may write register values that have to be communicated back to the main thread. Second, the slice and main program "threads" share in-flight memory state. The handling of these differences is discussed in sections 3.2.3 and 3.3, respetively. Re-executing deferred loads forward non-speculatively from stores using a new structure called a Chained Store Buffer (CSB) which is discussed in Section 3.3.1.

Non-speculative retirement. BOLT can retire deferred region instructions *non-speculatively* in program order, once those instructions have executed. Although deferred region loads are non-speculative relative to same-thread stores, they are still vulnerable to stores from other threads. BOLT verifies loads with respect to stores from other threads as they retire in program order using SVW-MT, essentially extracting the inter-thread verification mechanisms of SVW into a separate step. SVW-MT is discussed in Section 3.4.2.

To reconcile instruction granularity load verification and non-speculative retirement with checkpoint-based register management, BOLT uses a complementary pair of new techniques called Decoupled Store Completion and Silent Deterministic Replay (SDR). These techniques are discussed in Section 3.5.1.

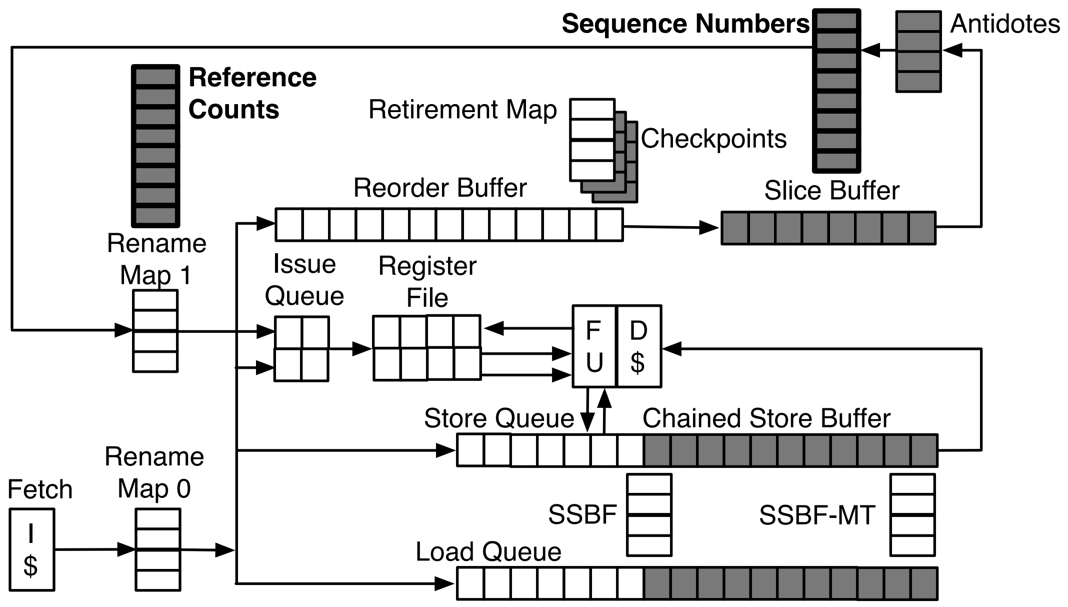


Figure 3.3: Structural diagram of BOLT with register management related structures highlighted.

3.2 Register Management

BOLT's hybrid register management scheme allows each region of the dynamic instruction window to be managed by the most appropriate algorithm. Tail instructions are most likely to experience mis-predictions, and benefit from conventional ROB-style instruction-granularity management that supports minimal recovery to any instruction. Deferred instructions are much less likely to experience mis-speculations, and when they do, they squash many younger instructions, making checkpoint overhead relatively less important. Deferred instructions, however, benefit from the aggressive CPR-style management, as the window needs to grow to accommodate one or more long latency misses. Figure 3.3 highlights the structures involved in BOLT's register management algorithms.

To implement hybrid register management, BOLT unifies the ROB and CPR register reclamation algorithms such that they operate together on a single physical register file. BOLT accomplishes this by formulating ROB-style register management as reference counting. Because CPR-style management is already formulated as reference counting, BOLT can combine the formulations in a straightforward way—a register is held if it

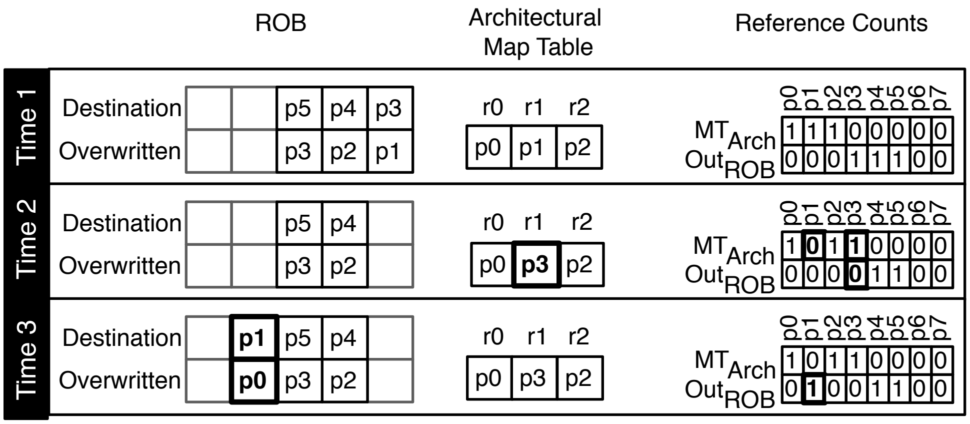


Figure 3.4: ROB-style Register Management as Reference Counting.

is held in either scheme (*i.e.* if its combined reference count is positive) and free if it is free in both (*i.e.* its combined reference count is zero).

3.2.1 ROB-style Register Management as Reference Counting

In conventional register management, every register in the retirement map table (the registers "live in" to the head of the ROB), and every output register in the ROB is live. Conventional processors implement this algorithm using a circular queue. When an instruction retires, the physical register it overwrites is reclaimed by adding it to a circular queue.

Conventional register management can also be implemented using a two row reference counting matrix—one row for the retirement map table (MT$_{\text{Retire}}$), and one row for the output registers in the ROB (Out$_{\text{Rob}}$). Both are implemented using latches because they require active manipulation. When an instruction is allocated a new destination physical register at rename, the corresponding bit in the Out$_{\text{Rob}}$ row is set. When an instruction (speculatively) retires, its output register no longer appears in the ROB, but does appear in the retirement map table. The bit corresponding to the retiring instruction's output register is set in MT$_{\text{Retire}}$, and cleared in Out$_{\text{Rob}}$. Likewise, the bit corresponding to the retiring instruction's overwritten register is cleared MT$_{\text{Retire}}$.

Figure 3.4 shows a three step example of ROB-style register management implemented as reference counting. In each time step, the output and overwritten physical registers of each instruction in the ROB are shown on the left. The retirement map table is shown in in the middle. The corresponding reference counting matrix is shown on the right. In each time step after the first, the changes since the previous are shown in boldface. Between the first and second time steps, the oldest instruction in the ROB has retired. Retiring this instruction replaces p1 with p3 in the retirement map table, and updates $\mathrm{MT}_{\mathrm{Retire}}$ and $\mathrm{Out}_{\mathrm{Rob}}$ accordingly. Between the second and the third time step, a new instruction which writes p1 is renamed, resulting in its output register in p1's bit being set in $\mathrm{Out}_{\mathrm{Rob}}$.

Supporting recovery. In a conventional, ROB processor there are two forms of recovery—serial recovery and recovery to an in-ROB checkpoint. When a ROB-based processor performs serial recovery, it traverses the ROB in reverse, freeing destination physical registers, and reverting the map table so that the entry holds the immediately previous mapping for the logical register being recovered. Updating the reference counts during this recovery is simple—as each instruction is removed from the ROB, the bit corresponding to its output register is cleared in $\mathrm{Out}_{\mathrm{Rob}}$.

ROB processors also typically support a few map table checkpoints to accelerate mis-speculation recovery. On a mis-speculation, the processor recovers to the immediately younger checkpoint, then performs serial recovery as needed to the point of the mis-speculation. Supporting this form of mis-speculation recovery requires checkpointing $\mathrm{Out}_{\mathrm{Rob}}$ whenever a rename map table checkpoint is taken. This can be implemented using either a few latch ranks or a small SRAM. Whenever an instruction (speculatively) retires, a column clear operation is performed across all of these checkpointed rows for the retiring instruction's destination register. If the checkpoint is restored, the updated row is restored along with it to reflect the registers currently in the ROB. These checkpointed rows do not need to participate in the column-ORs which are used to determine what registers are free, as the bits in each checkpoint are a subset of those in the row for the ROB's output registers.

3.2.2 Combining ROB-style and CPR-style Register Management

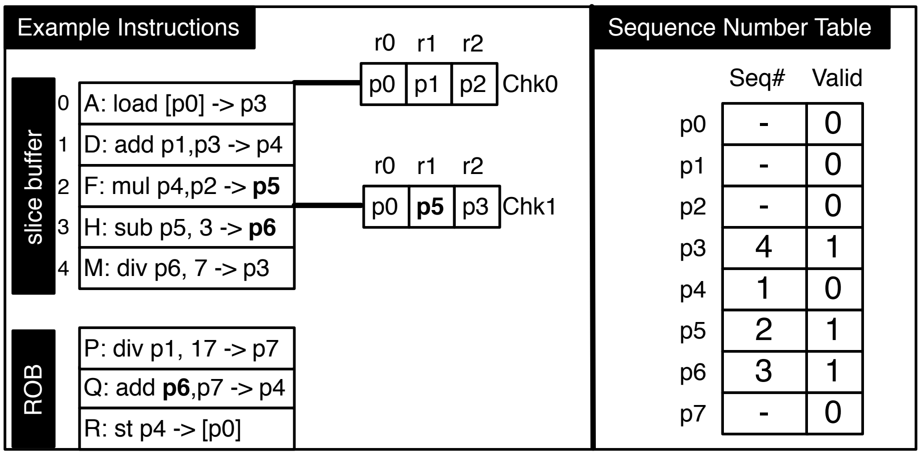
With ROB-style register management encoded as a two row reference counting scheme, BOLT's hybrid algorithm can be implemented by adding the rows needed for CPR-style register management of deferred instructions. BOLT needs two types of rows for deferred instructions. First, BOLT needs one row per issue queue entry so that re-executing deferred instructions can hold live the registers they read and write. Second, BOLT needs one row per retirement map table checkpoint. Unlike rename map table checkpoints, retirement map table checkpoints behave like CPR checkpoints—their reference count rows participate in the column-ORs and are not incrementally updated. These retirement map table checkpoint rows are copied to or from the retirement map table row when a checkpoint is restored or created respectively.

One potentially advantageous difference between CPR's and BOLT's reference counting matrix is that BOLT only needs the issue queue entry rows for re-executing deferred instructions, while CPR needs these rows for *all* instructions. In BOLT, tail instructions do not need to explicitly reference count the registers they use in the issue queue rows because these registers are held live by the reference counts used to implement ROB-style reclamation. This may allow BOLT to save energy—tail instructions do not need to access it, and it may be possible to power it down entirely when latency tolerance is not active.

3.2.3 Tracking Live-Out Destination Registers of Deferred Instructions

One register management detail which both CFP and BOLT must handle properly is ensuring that a re-executing slice instruction is re-allocated its original destination physical register if that register is still live either in the tail region or in a deferred region checkpoint. In this situation, the re-renaming instruction must be re-allocated the same output register so that its result will be properly communicated to younger instructions.

The left side of Figure 3.5 illustrates the need to detect live-out destination registers



	Seq#	Valid
p0	-	0
p1	-	0
p2	-	0
p3	4	1
p4	1	0
p5	2	1
p6	3	1
p7	-	0

Figure 3.5: Slice Live-out Destination Registers. Left: example contents of the slice buffer and ROB. Bold face registers highlight the need to detect slice live-outs, as they may appear in checkpoints (p5) or as inputs to younger instructions in the ROB (p6). Right: BOLT's sequence number table detects slice live-outs. A re-executing register is re-assigned its original output physical register if the sequence number matches the instruction's slice buffer position and the valid bit is set.

in a re-executing slice. In this example, the contents of the slice buffer and ROB are shown with their original input and output physical registers. Here, deferred instruction F's output register, p5, appears in checkpoint 1. When F re-executes, it must be re-allocated p5 so that checkpoint 1 will represent the correct architected state when checkpoint 0 commits. Instruction H's output register, p6 is one input of instruction Q, which has not yet executed. If H re-renames before Q executes, re-allocating p6 to H will allow Q to execute properly instead of slicing out. Slicing Q out in such a situation not only poses an inefficiency—Q spuriously re-executes—it also introduces a difficulty in ensuring Q slices in properly, as the slice buffer pass may have already passed it.

One approach to identifying when an instruction must be re-allocated its original output register is to keep a table which maps physical register numbers to the most recent instruction to write that register. When an instruction is renamed—and a physical register is allocated to it—it updates this table with a unique sequence number identifying the instruction. If the instruction subsequently defers and re-executes, then it compares the sequence number in the table for its originally allocated destination physical register to its own. If the sequence numbers match, then the instruction is re-assigned its original destination physical register. Otherwise, a new physical register is allocated from the free list. This approach—which is essentially what CFP uses—is correct, but it can be made more efficient in BOLT.

BOLT can refine this approach because of its explicit distinction between tail and deferred instructions. Tail instructions use ROB-based conventional register management, and do not actually need re-renaming—they already have valid register mappings. Consequently, BOLT only needs to perform the live-out test in the deferred region. BOLT exploits this fact to make the algorithm more efficient. Instead of updating the sequence number table at rename for *all* instructions, BOLT updates the sequence number table at speculative retirement, and only for instructions which are actually deferring. Because only deferred instructions require sequence numbers, BOLT uses an instructions slice buffer position as its sequence number. BOLT also uses a bit-vector to track which entries in this table are valid. Bits in this valid bit-vector are cleared as instructions are renamed, and set as instructions defer. When a deferred instruction re-renames, it is re-assigned its original output register if the sequence number in the table matches its slice buffer position *and* the corresponding valid bit is set.

The right side of Figure 3.5 shows the refined sequence number table corresponding to the instructions in the left side. When A is re-renamed, the sequence number corresponding to A's original output register (p3) is 4, which is different from A's slice buffer position (0). This mismatch indicates that p3 has been subsequently re-allocated to another deferred instruction (*e.g.* E). For D, the sequence number for p4 is 1, which matches D's slice buffer position, however, the valid bit is clear. The invalid entry indicates that p4 has subsequently been re-allocated to an instruction which has not deferred (*e.g.* Q). A and D are allocated new output registers during re-renaming. F, H, and M find that the entries corresponding to their original registers (p5, p6, and p3 respectively) are valid and match the corresponding slice buffer positions. These three instructions are re-assigned their original output registers.

BOLT's refined algorithm is more efficient than the original algorithm in three ways. First, the sequence number table and bit-vector are only manipulated when speculative retirement is active. Second, when latency tolerance is active, the sequence number table is updated only by those instructions which actually defer. Instructions which rename while latency tolerance is active must clear the valid bit-vector, but this is a smaller structure. Third and finally, the sequence numbers themselves are narrower. When sequence numbers are assigned at rename, a sufficiently large number of bits to prevent roll-over within the window is required (*e.g.* 12 bits). BOLT's refined scheme uses slice buffer position, which is both narrower (*e.g.* 7 bits) *and* does not need to be represented explicitly in the slice buffer entry.

3.3 Store-to-load Forwarding

BOLT explicitly divides the window into tail and deferred regions. Like the instructions in a conventional ROB processor, instructions in the tail region are speculative in several ways. All tail instructions are speculative relative to older un-resolved branches. Tail loads are speculative relative to older stores from both the same thread and other threads in a shared memory program. Instructions in the deferred region are less speculative. Specifically, deferred loads are non-speculative relative to older stores from the same thread. This allows BOLT to avoid re-verifying these loads relative to same thread stores after speculative retirement (see Section 3.4 for details of BOLT's load verification) and reduces squashing and lost work. However, it also requires a new scalable non-speculative forwarding mechanism.

This new store to load forwarding mechanism must meet several requirements. First, forwarding in the deferred region must be non-speculative. This requirement also implies that the forwarding mechanism must be able to avoid write-after-read hazards that may occur when speculatively retired loads re-execute in the presence of younger stores to the

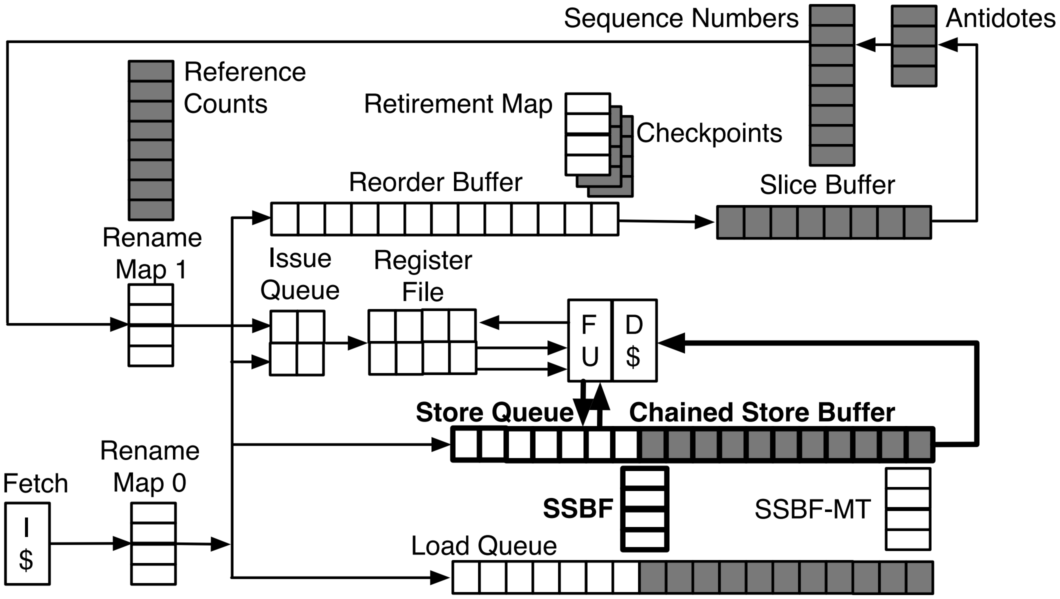


Figure 3.6: Structural diagram of BOLT with store management related structures highlighted.

same address. Second, it should provide the ability to propagate poison through store-to-load dependences. Third, common case accesses should be fast—short enough to overlap with data cache hits—so that they do not impose latency penalties on most loads. Finally, the mechanism used for forwarding within the deferred region should mesh well with the tail region's forwarding mechanism.

BOLT uses SQIP [74] to meet three of the four requirements—all except non-speculative forwarding for re-executing deferred loads. To address this requirement, BOLT combines SQIP with a new technique, the chained store buffer (CSB). Both SQIP and CSB require only indexed accesses to the store queue/buffer, and can be combined into a single physical structure. Tail loads forward speculatively using SQIP—from both tail and deferred region stores. Re-executing deferred loads use CSB to forward non-speculatively. Figure 3.6 highlights BOLT's structures used for store management.

Note that while a transactional data cache may seem like an appealing possibility, it is ill-suited to the re-execution requirements of load latency tolerance. Specifically, it is unable to forward values from stores other than the youngest, posing write-after-read hazards which require squashes for forward progress. The problems with using a transactional data cache with BOLT are evaluated in Section 5.6.

3.3.1 Chained Store Buffer Mechanics

A chained store buffer (CSB) is a traditional FIFO store buffer that replaces associative search with iterative indexed access. Loads "search" the CSB by starting at the youngest store and reading one entry per cycle until a match is found or until it is certain that no match exists. To make this process fast, a CSB overlays an address indexed hash table on top of the traditional FIFO store buffer. Structurally, this hash table requires two additions. First, a *root table* maps low-order address bits to the SSN of the most recent store to write to a partially matching address—*i.e.* to the youngest store in that hash table "bucket." Recall from Section 2.7.1 that an SSN uniquely identifies an in-flight store and the SSN's low-order bits specify the store's store buffer position. Second, each store buffer entry is expanded to include a *link field*—an SSN which specifies the immediately prior store to an address with the same low order address bits—*i.e.* to the next store in the same bucket. BOLT's CSB buffer organizes these buckets at a granularity of eight bytes—the size of the largest store data size.

Creating the hash table. BOLT exploits the in-order nature of (speculative) retirement to incrementally construct the hash table in the store buffer. When a store (speculatively) retires, it reads the root table entry corresponding to its address, and uses the SSN it finds there to fill in its store buffer entry's link field. The store then updates that root table entry with its own SSN. If a store's address is unknown because it depends on a long latency miss via its address input, speculative retirement stalls until the miss returns and the stores address is calculated. Empirically, this occurs relatively rarely, causing seven stall cycles per thousand cycles on average for SPEC 2006.

Figure 3.7 shows an example of (speculative) retirement and chained store buffer insertion in two time steps. Fields in white (address, value, size) exist in a conventional store buffer. Fields in grey (poison) are required for load latency tolerance. Fields in black (root table and link field) are specific to the chained store buffer. The left side of the figure

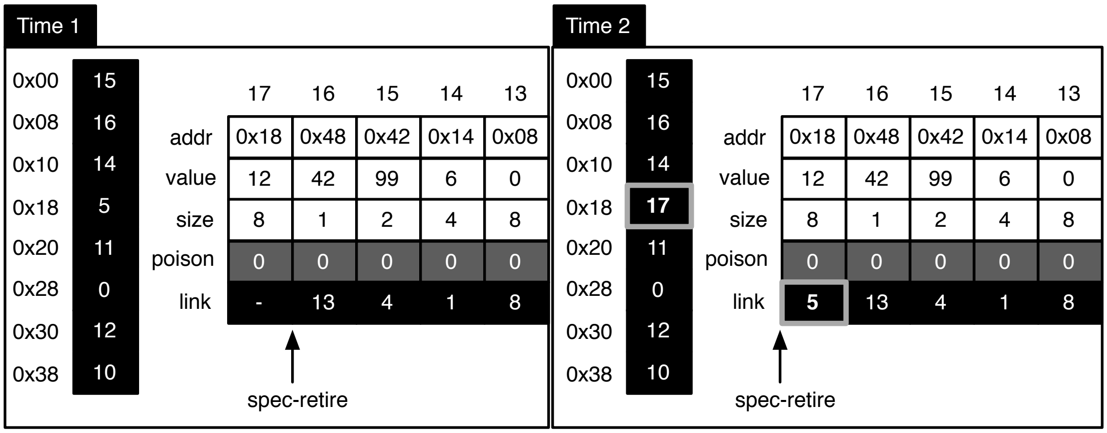


Figure 3.7: Chained Store Buffer. Left: before the (speculative) retirement of store 17. Right: after the (speculative) retirement of store 17. Black fields are CSB specific. Grey fields are required for load latency tolerance in general. White fields exist in conventional store queues/buffers.

shows the state of the store buffer before the store with SSN 17 (speculatively) retires. At (speculative) retirement, this store reads the root table entry corresponding to its address (bucket 3 for address 0x18) and finds that the immediately preceeding store to this bucket was at SSN 5. It updates its link field to 5, then writes 17 into the root table entry. The state of the chained store buffer after store 17 retires is shown on the right side of the figure.

Searching the CSB. A load "searches" the CSB by first reading the root table entry corresponding to its address. If the SSN that the load reads is less than or equal to $\text{SSN}_{\mathsf{store-complete}}$—indicating that the most recent potentially matching store is already in the data cache—then the load does not need to examine the store buffer further. If the SSN from the root table entry is greater than $\text{SSN}_{\mathsf{store-complete}}$, then the load reads the corresponding store buffer entry in the next cycle. On an address match, the load forwards from the store buffer entry. If the address does not match, the load repeats the process using the SSN from the store buffer entry's link field—$\text{SSN}_{\mathsf{link}}$. This process continues until either a match is found or $\text{SSN}_{\mathsf{link}} \leq \text{SSN}_{\mathsf{store-complete}}$.

A deferred load may re-execute in the presence of younger stores to the same address. To avoid write-after-read hazards with these stores, a load simply ignores younger stores even if their addresses match. A load notes the SSN of the immediately older store when it first enters the pipeline. During a CSB traversal, the load ignores any store with an SSN

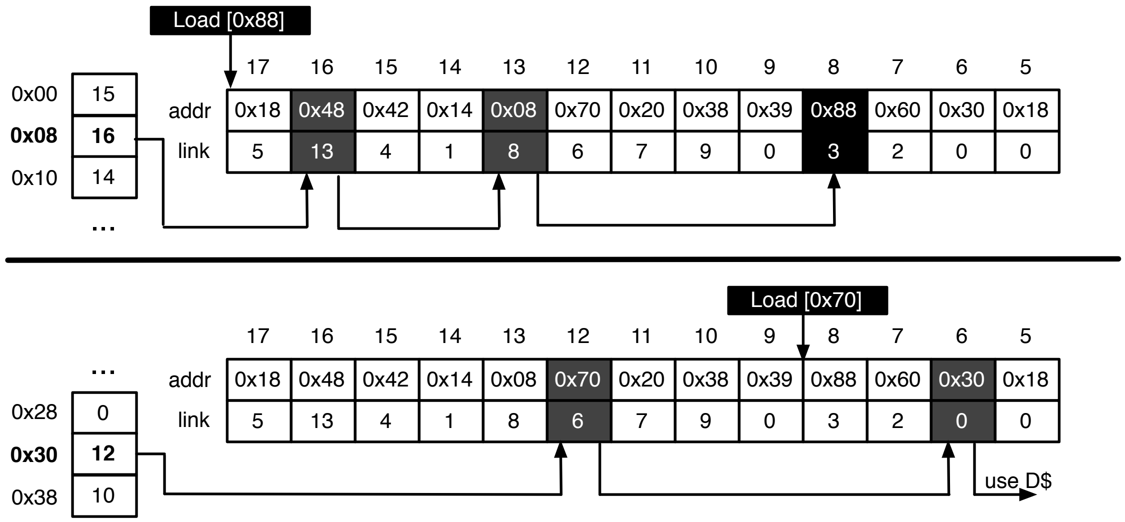

Load [0x88]
17 16 15 14 13 12 11 10 9 8 7 6 5
0x00 15
0x08 16
0x10 14
...
addr 0x18 0x48 0x42 0x14 0x08 0x70 0x20 0x38 0x39 0x88 0x60 0x30 0x18
link 5 13 4 1 8 6 7 9 0 3 2 0 0
Load [0x70]
17 16 15 14 13 12 11 10 9 8 7 6 5
...
0x28 0
0x30 12
0x38 10
addr 0x18 0x48 0x42 0x14 0x08 0x70 0x20 0x38 0x39 0x88 0x60 0x30 0x18
link 5 13 4 1 8 6 7 9 0 3 2 0 0
use D$

Figure 3.8: CSB Search Traversal. Examples of two loads searching the CSB. A load's age relative to the stores is depicted by its position in the diagram. Grey entries are examined but do not constitute matches. Black entries indicate forwardings.

greater than the one noted.

Figure 3.8 shows the actions of two loads as they search the CSB. Only the address and link fields are shown as the other fields are irrelevant. Grey entries are examined but not forwarded from. Black entries indicate forwardings. In the top example, a load to address 0x88 searches the CSB. This load is younger than all entries in the CSB—its immediately older store is 17. The load reads the root table and finds that store 16 is the youngest store in bucket 1 (corresponding to address 0x88). In the next cycle, the load reads store 16 and finds that the store does not match—its address is 0x48. In the next cycle, the load uses store 16's SSN_{link} (13) to read the store buffer again. Another mis-match results in the load using store 13's SSN_{link} to subsequently check store 8. Here, a match is found so the load forwards.

The bottom example in Figure 3.8 shows a re-executing deferred load to address 0x70 which is logically ordered between stores 8 and 9. Here, the load reads the root table and finds store 12 to be the youngest potentially matching store. The address match with store 12 is ignored because store 12 is younger than the load. The load uses store 12's SSN_{link} to check store 6. Store 6 is older than the load, but does not match. Store 6's SSN_{link} is 0, which is older than any store in the store buffer, so the load simply uses the value from the data cache.

A CSB presents the possibility of adding many cycles to load verification or re-execution latency. However, in practice almost no latency penalties are incurred. Many loads simply read the root table and find that there is no need to consult the store buffer. Of loads that *do* read the store buffer, most find the forwarding store on the first hop. These two cases are significant as they can be overlapped with data cache access and do not impose a latency penalty. If extra hops are required, the load is replayed—as if it had missed the data cache. In this case, the remaining CSB traversal is performed by a small finite state machine, and the load is re-issued when the traversal is complete. Empirically, reasonable sized (*e.g.*, 256-entry) root tables result in five extra hops per 100 loads. Section 5.7.2 provides more detailed results on CSB performance.

Interaction of SQIP and CSB. The CSB covers only those stores which have reached (speculative) retirement. For tail loads, only searching the CSB may miss stores which have not retired and been inserted into the hash table yet, meaning tail loads must forward speculatively. BOLT uses SQIP for tail loads, then verifies them at (speculative) retirement using SVW (see Section 3.4). In a different design, BOLT could use a conventional associative store queue for pre-retirement stores and CSB for retired stores. This design would involve two physical structures, and require copying stores from the store queue to the store buffer at (speculative) retirement. Alternatively, BOLT could use *both* SQIP *and* CSB for tail loads—searching the CSB if the entry predicted by SQIP does not match. This design would eliminate some SQIP forwarding mis-predictions from (speculatively) retired stores. However, SQIP already forwards from deferred stores with high accuracy, so the added complexity of doing both SQIP and CSBF for tail loads is not needed.

3.3.2 Implementation Issues

Partial forwardings. Partial forwarding is the situation in which a wide load forwards from a narrow store. Here, the load's value must be constructed by combining values from the narrow store, and some combination of other older stores and/or the data cache. Conventional processors typically wait to execute loads which require partial forwarding until the partially matching store completes to the data cache [37]. In a load latency tolerant design, stalling such loads is *particularly* undesirable. Stalling a load means that it and its dependents block the issue queue and speculative retirement. In addition to the direct performance impact—younger instructions cannot enter and execute—this may cause a resource deadlock. Specifically, older miss-dependent instructions may need to acquire issue queue entries and physical registers whose reclamation is blocked by the stalled load. The store for which the load is stalling may be unable to complete until these instructions re-execute. Breaking this deadlock requires squashing tail instructions until forward progress can be made.

A chained store buffer makes it easy to execute a load even if it partially forwards. As the load traverses the CSB, it constructs the appropriate value piece-by-piece. This construction requires a register to hold the value constructed so far, and a bit mask to indicate which bytes have been filled in. Traversal stops when either the entire value has been constructed, or an SSN less than or equal to $\mathrm{SSN}_{\mathsf{store-complete}}$ is encountered. In the later case, the value read from the data cache is used to fill in the remaining pieces of the output value of the load.

Partial forwardings may occur for either re-executing deferred loads which search the CSB as they re-execute, or for executing tail loads which do not. When a tail load attempts to forward using SQIP but cannot due to a partial forwarding, it waits until the partially matching predicted store (speculatively) retires—which is not the same as waiting for that store to complete. The load then executes by searching the CSB.

Un-aligned loads and stores. Un-aligned loads and stores may span two CSB buckets. These require special handling as they do not fit cleanly into the chaining algorithm. An un-aligned load that spans two CSB buckets traverses the chained store buffer twice—once for each bucket—and the output value is constructed piece-by-piece similarly to the partial forwarding case.

Stores which span two root table buckets introduce more complexity. Not only do such stores require writing to two root table buckets, they also logically have two links. One option is to provision the store buffer with two link fields to accommodate these stores. This option is simplest, but over-provisions link field capacity and root table bandwidth for the uncommon case. In BOLT, an un-aligned store that spans two buckets reads only the root table bucket for its starting address, but writes its SSN in both buckets. The store then only links properly in the first bucket. As a load traverses the CSB, it checks whether the store's link field corresponds to its own bucket by checking the store's starting address. If the store's link field does *not* correspond to the load's bucket, then the load reverts to linearly scanning the store buffer until it finds a store that belongs to its bucket again. At this point, it resumes chaining. Fortunately bucket-spanning un-aligned loads and stores are rare.

Optimization for sub-word writes. The word granularity of the CSB poses two performance problems in the presence of consecutive narrow (sub-word) writes. Both problems arise from the fact that consecutive sub-word writes all map to the same hash table bucket effectively forming a high number of collisions. The first problem occurs when a load maps to a bucket that contains a chain of sub-word writes but does not match their address. Here, the load must traverse each of the narrow stores serially. In the worst case, eight sequential single-byte writes would add eight cycles to the search latency of the load. The other problem arises when a narrow load which matches an older store to a word must traverse the younger stores to the same word before finding its match. An example, is byte-stores to 0x10, 0x11, 0x12,...0x17, followed by a byte-load to 0x10. Here, the load must traverse 0x17–0x11 before finding 0x10.

BOLT solves the second problem first by adding a single "run" bit per store buffer entry. This bit is set if the preceeding store buffer entries form a contiguous run of same-size sub-word stores to consecutive addresses which start at a bucket boundary. The CSB-insertion logic tracks the address and data size of the most recently inserted store to determine whether to set or clear the "run" bit in each entry. The most recent address and data

Description	Condition	Run bit	lastAddr	lastDsize
Continuation	st.addr == lastAddr + lastDsize && st.dsize == lastDsize && SameBucket (st.addr, lastAddr)	Set	st.addr	(same)
Start of run	StartOfBucket (st.addr)	Clear	st.addr	st.dsize
Not a run	Otherwise	Clear	Invalid	Invalid

Table 3.1: Rules for tracking the "run" bit in the CSB. Rules are matched in order

size registers are set to "invalid" values when no valid run is in progress. The rules for manipulating these registers and the "run" bit are shown in Table 3.1. When the "run" bit in a store buffer entry is set, a load may skip to the correct store within a word by directly calculating the matching store's index, rather than examining each store cycle-by-cycle. The "run" bit for the store which starts the contiguous run is clear because there are no younger stores whose index may be calculated from it.

With same-word stores in contiguous runs being searchable by SSN arithmetic, their $\mathrm{SSN}_{\mathsf{link}}$ fields no longer need to reference each other. BOLT solves the first problem by using the oldest store in the sequence's $\mathrm{SSN}_{\mathsf{link}}$ for the remaining stores in the sequence. Specifically, in the case of 8 byte-size stores to 0x10, 0x11, 0x12, ... 0x17, the store to 0x10 will read the root table as usual, but the stores to 0x11, 0x12, ... 0x17 will not. Instead, as CSB-insertion determines they are part of a run, they will receive the same $\mathrm{SSN}_{\mathsf{link}}$ as the store to 0x10 did. A colliding load to a non-matching address will simply skip over all eight stores at once.

3.4 Load Verification

BOLT is an out-of-order processor that executes loads speculatively relative to stores from the same thread. It also executes loads speculatively relative to stores from other threads in a shared memory program. BOLT must verify both forms of speculation. BOLT uses SVW-style verification for both of these tasks, but divides them into two separate pipeline stages, each with their own structures. Figure 3.9 highlights BOLT's structures used for

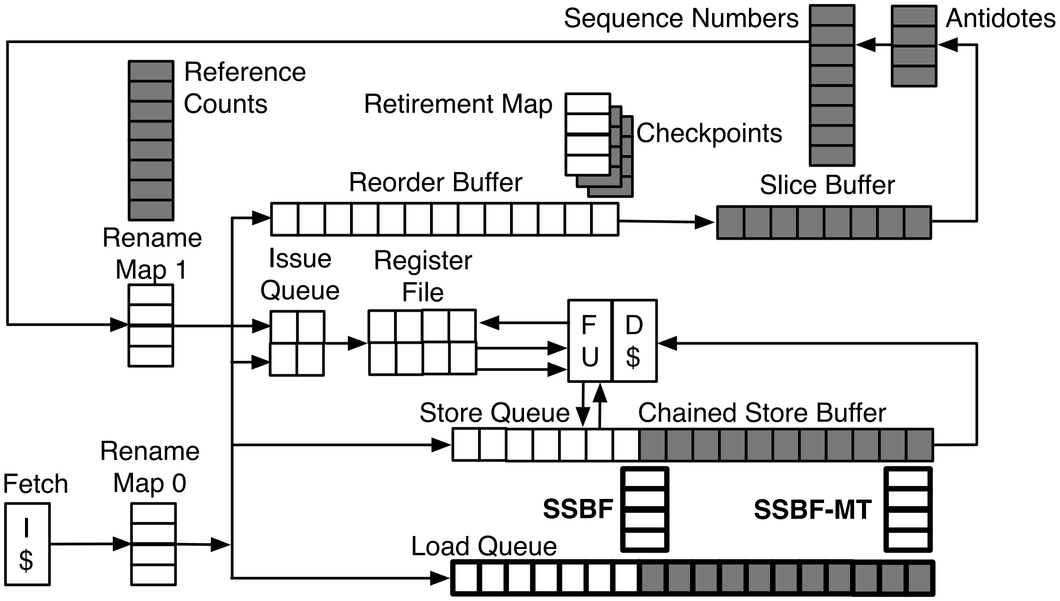


Figure 3.9: Structural diagram of BOLT with load verification related structures highlighted.

load verification.

3.4.1 Verification Relative to Same-thread Stores

BOLT's store-to-load forwarding is speculative with respect to same-thread stores only before speculative retirement. Loads re-executing after speculative retirement forward non-speculatively from the chained store buffer. This distinction makes speculative retirement the natural place for BOLT to apply SVW to verify loads with respect to older stores from the same thread.

BOLT modifies SVW slightly from its original form for its same-thread verification. The root table used by the chain store buffer is identical to the direct-mapped SSBF with which SVW was originally proposed [67]. BOLT combines the chained store buffer's root table and SVW's SSBF into a single physical structure.

Many subsequent proposals [68, 74, 75, 78] have replaced the direct-mapped SSBF with a tagged "set-associative" SSBF in which each set is managed like a FIFO. SSBF associativity has the same advantage as cache associativity—fewer conflict misses. In an SSBF, a conflict miss—relative to some load—is caused by a store to a different but aliasing address that is younger than the load's forwarding store. Such aliasing forces the load to re-execute and, potentially, to wait for some older stores to complete before re-executing. Stalling speculative retirement for store completions is not desirable for load latency tolerant designs.

Fortunately, the searchable nature of the chained store buffer covers exactly those cases in which SSBF associativity is useful. In ROB/SQIP designs, SVW must stall on set-conflicts when the SSBF does not contain information about the load's address, but presents the possibility of a matching store in the store buffer. The non-searchable nature of a SQIP store buffer requires SVW to stall until enough stores complete that the SSBF's information guarantees re-execution is safe. SVW *can* search a chained store buffer. BOLT can keep the root table direct mapped and un-tagged, reducing per-access energy.

One difference between SVW using a set-associative SSBF and SVW searching arises when SVW must train the memory dependence predictor. A set-associative SSBF can contain stores which have already completed to the data cache. SVW can still train from such stores because their SSNs are available, and the appropriate distance may be computed. When SVW uses the chained store buffer instead, stores which were in the store buffer during load execution may have since completed. If SVW is unable to train the memory dependence predictor for these pairs, significant squashes will impede performance. While completed store are no longer logically in the store buffer, they may still physically remain—their entries may not have been reclaimed yet. With a large enough store buffer, the chances of the store having been overwritten becomes low. SVW uses this observation to address the training issue—SVW traverses the chained store buffer past $\mathsf{SSN}_{\mathsf{store-complete}}$. To know when it must stop a traversal, SVW tracks a new SSN—$\mathsf{SSN}_{\mathsf{overwritten}}$ the SSN of the youngest store whose store buffer entry has been overwritten. $\mathsf{SSN}_{\mathsf{overwritten}}$ is updated at dispatch according to the rule $\mathsf{SSN}_{\mathsf{overwritten}}$ = MAX($\mathsf{SSN}_{\mathsf{overwritten}}$, $\mathsf{SSN}_{\mathsf{dispatch}}$- storebuffer.size).

3.4.2 Verification Relative to Stores from Other Threads

Until a load retires non-speculatively, it is vulnerable to stores from other threads. BOLT also detects these violations using SVW-style filtered load re-execution, but does so in a separate (later) pipeline stage from same-thread verification, called SVW-MT. While SVW verification is performed at speculative retirement, SVW-MT verification is performed at non-speculative retirement. Non-speculative retirement is the logical place for SVW-MT to take place because the memory ordering constraints imposed on loads by the memory model are phrased in terms of non-speculative retirement, when loads become "observable".

At a high level, SVW-MT operates similarly to SVW—load re-executions are filtered by checking an SSBF-like table for potentially conflicting stores within the loads' window of vulnerability. The exact details of SVW-MT, however, are slightly different. SVW-MT's SSBF is called an SSBF-MT to distinguish it from SVW's structures. Whereas the SSBF is organized at a word granularity, the SSBF-MT is organized at data cache line granularity. The SSBF-MT uses this granularity because coherence messages—which serve as proxies for completing stores on other cores—operate at cache line granularity.

The rules for manipulating the SSBF-MT differ as well. The SSBF-MT does not track same-thread stores at all—they are now irrelevant. Whenever a line is invalidated from the data cache, the SSBF-MT entry corresponding to that line is updated with $\text{SSN}_{\text{store}-\text{complete}}$ +1. This SSN is significant because it will signal a potential violation on all non-forwarding loads which have not yet read the data cache. These invalidations may occur either because another processor requested write permissions for the cache line, or because of a local replacement. In the former case, re-execution is required as the other core's write may cause a violation. In the latter case, re-execution is required because there exists the possibility that another core writes the line *after* eviction. With the line no longer in the cache, SVW-MT will not be able to track any future activity on that cache line.

SVW-MT processes loads in the same manner as SVW—loads read the SSBF-MT and compare SSNs to test for vulnerability. If the load is vulnerable to a store from another thread, then the load re-executes, and signals a violation if the new value differs from the old value. This verification must be done in a way that respects the constraints of the memory consistency model. Consequently, SVW-MT must stall at any instruction—fence, load, or atomic operation—until its memory consistency constraints are satisfied.

Optimizations. SVW-MT uses two optimizations to improve its performance. The first optimization targets the pathological case in which a program executes for a prolonged period with no stores occurring in the dynamic instruction stream. Without stores, $\text{SSN}_{\text{store}-\text{complete}}$ never changes, so SVW-MT must re-execute many loads despite the fact they are safe. Specifically, an eviction may occur well before a load even enters the window, and record $\text{SSN}_{\text{store}-\text{complete}}$ +1 in the SSBF-MT. Much later, a load dispatches and executes, noting $\text{SSN}_{\text{store}-\text{complete}}$. If no stores have occurred between those two times, the load appears vulnerable, as $\text{SSN}_{\text{store}-\text{complete}}$ has not changed. Effectively, “time” stands still for SVW-MT in the absence of stores. Without a technique to address this pathology, SVW-MT re-executes 6% of loads on *soplex*, causing a 20% slowdown.

SVW-MT addresses this pathology by ensuring that the SSNs advance even in the absence of stores. If dispatch encounters a certain number (*i.e.* 100) of consecutive non-store instructions, and no stores are present in the store queue/store buffer, it increments $\text{SSN}_{\text{dispatch}}$ by one immediately before dispatching the next load that it encounters. Whenever a load instruction non-speculatively retires, its SSN—which was the value of $\text{SSN}_{\text{dispatch}}$ when the load dispatched—is greater than $\text{SSN}_{\text{commit}}$, then $\text{SSN}_{\text{commit}}$ and $\text{SSN}_{\text{store}-\text{complete}}$ are advanced by one. This effectively puts a “fake” store in the store queue to mark the passage of time, preventing this pathology.

The other optimization targets low re-execution rates while using a small SSBF-MT by filtering SSBF-MT writes. Here, a write to the SSBF-MT is eliminated if there is no possibility of an in-flight load having read the block being evicted. To implement this filtering, the data cache tags are extended to include an SSN ($\text{SSN}_{\text{max}-\text{read}}$). Whenever a load executes, it writes $\text{SSN}_{\text{dispatch}}$ to the $\text{SSN}_{\text{max}-\text{read}}$ for the block it reads. Whenever a block is evicted, $\text{SSN}_{\mathsf{max-read}}$ is compared to $\text{SSN}_{\mathsf{store-complete}}$. If $\text{SSN}_{\mathsf{max-read}}$ is less than $\text{SSN}_{\mathsf{store-complete}}$, then the write may safely be eliminated. This test guarantees that all loads which have read the block have retired or been squashed. Specifically, the fact that at least one store which had not yet entered the window when the block was last read, has completed means that every instruction in the window when the block was last read must have either retired or been squashed. Whenever SSNs wrap around, the $\text{SSN}_{\mathsf{max-read}}$ is flash cleared in all cache blocks at the same time that the SSBF and SSBF-MT are cleared.

The performance impacts and costs of these optimizations are evaluated in Appendix A. These optimizations can be made more space efficient by storing and comparing only the more significant bits of the SSNs and ignoring the least significant bits. Here, some precision is lost in tracking when an eviction may safely be ignored, but such tracking is conservative. Storing only the upper 8 bits of a 16-bit SSN has negligible impact on performance.

Handling SMT. In an SMT processor, steps must be taken to ensure that consistency constraints are enforced not only between different cores, but also between the threads running on the same core. To accomplish this naïvely, SVW-MT could update the SSBF-MT for all other threads sharing the core whenever one thread completes a store. While correct, this strategy would need a huge SSBF-MT to avoid high re-execution rates due to conflicts from all of the SSBF-MT updates. The write filter optimization above already provides a test for whether or not a block may have in-flight loads. In an SMT processor, the $\text{SSN}_{\mathsf{max-read}}$ in each block's tag must be tracked per-thread because SSNs are meaningless across threads. This information can be used to then filter SSBF-MT writes done by stores from other threads on the local core—whenever one thread stores, $\text{SSN}_{\mathsf{max-read}}$ is checked against $\text{SSN}_{\mathsf{store-complete}}$ for each thread other than the one storing, and SSBF-MTs are updated as needed. Effectively, the write filtering optimization doubles as an enhanced version of SMT-Directory [36].

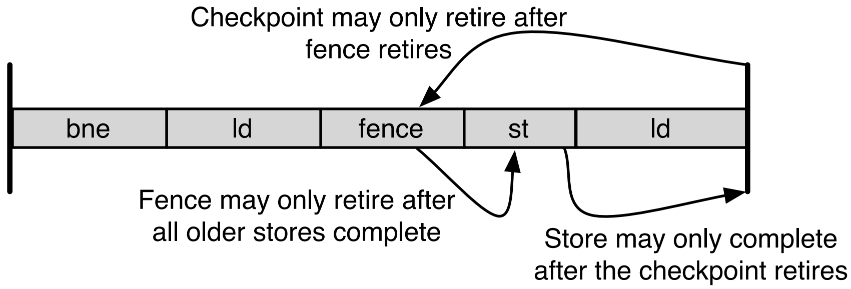

Checkpoint may only retire after fence retires
bne
ld
fence
st
ld
Fence may only retire after all older stores complete
Store may only complete after the checkpoint retires

Figure 3.10: Circular Dependence

3.5 Instruction Granularity Retirement

In BOLT—and in every speculative retirement architecture–there is a tension between the checkpoint-based nature of speculative retirement and the instruction granularity of memory consistency constraints. Specifically, many constraints require all older stores to complete before an instruction (*e.g.* a fence) is retired. If the fence is in the same speculative retirement checkpoint as an older store, then a circular dependence is created: the store cannot complete until it retires, the store cannot retire until its enclosing checkpoint can retire, the checkpoint is cannot retire until all instructions in it—specifically the fence—can retire, and the fence cannot retire until the older store completes. Figure 3.10 illustrates this circularity.

The problem here is that memory consistency models specify constraints on load and store *observable* ordering at the granularity of an instruction. Speculative retirement raises the granularity of observable load ordering—recall, loads are observed when they retire—to the granularity of a checkpoint, but otherwise leaves the granularity of observable store ordering—stores are observed when they completed—at an instruction granularity. This is not allowed.

One possibility is to raise the granularity of store completion (observable store ordering) to that of a checkpoint as well using either a transactional data cache—which can make a checkpoint's worth of speculative stores non-speculative at once—or using a two-phase store completion protocol [15, 82].

Another possibility is operate speculative retirement at a checkpoint granularity that is allowed by the memory consistency model. This does not limit the size of the speculative retirement region but rather divides it into multiple checkpoints. Under a relaxed model, such designs would create a checkpoint before every fence instructions and before and after every read-modify-write operation. Under sequential consistency (SC), such a design would create a checkpoint before every load. This is inefficient and essentially defeats speculative retirement whose strength otherwise would be its efficient management of register state.

An alternative approach is to somehow marry checkpoint-granularity register management with instruction granularity retirement. This would allow BOLT to use a small number of checkpoints, space them widely for efficient register management, and still implement any ordering requirement in a straightforward way.

3.5.1 SDR (Silent Deterministic Replay) and DSC (Decoupled Store Completion)

BOLT implements this marriage and decouples the granularity of retirement from the granularity of checkpoints using SDR (Silent Deterministic Replay) [34]. SDR uses a value capturing load queue to capture the values of all speculatively retired loads. These values are maintained as speculatively retired loads retire non-speculatively at instruction granularity. If a mis-speculation is detected within a checkpoint, the processor aborts to the checkpoint, which at this point is older than the non-speculative retirement point. However, it then executes back to the non-speculative retirement point by replaying loads from the load queue. This execution is silent—externally un-observable because loads get their values from the load queue rather than from the cache where they can trigger coherence actions. It is also deterministic—replayed loads (and by extension all instructions) will always produce the value they retired with.

SDR enables DSC (Decoupled Store Completion). With non-speculative retirement proceeding at instruction granularity, non-speculative store completion can proceed at instruction granularity as well. A mis-speculation in the speculative retirement region can

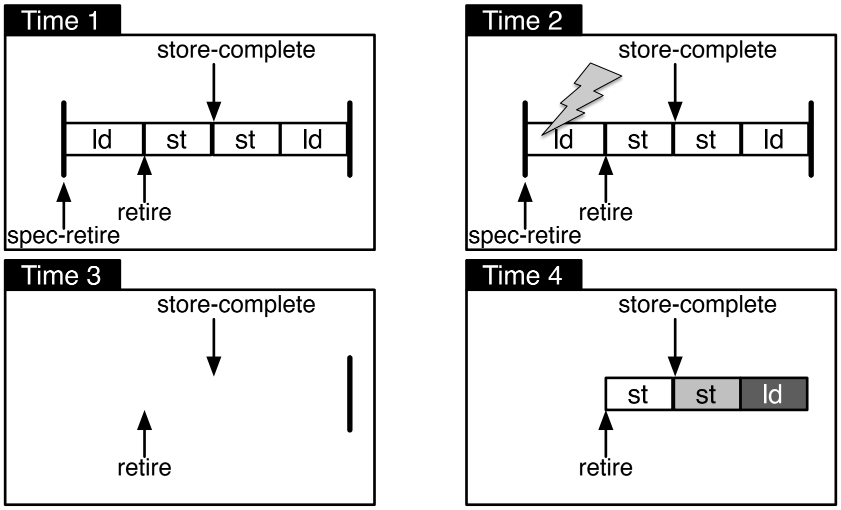


Figure 3.11: DSC/SDR

squash and re-execute past non-speculatively completed stores. SDR ensures that loads get the correct value even in the presence of younger completed stores that would otherwise cause write-after-read hazards. In BOLT, latency tolerance does not activate during SDR re-executions—all loads' outputs are known, so none are long latency. This fact simplifies forward progress guarantees, as conventional instruction granularity behavior is ensured until all retired instructions are re-executed.

Example. Figure 3.11 illustrates DSC and SDR. In Time 1, the retirement pointer and store completion pointer have advanced into the checkpoint. Note that without DSC and SDR, the retirement pointer would have to advance in one step from the checkpoint to the speculative retirement pointer. In Time 2, a violation is detected on a speculatively retired load. In Time 3, recovery is performed to the checkpoint. This recovery squashes retired instructions. In Time 4, these instructions are re-executed using SDR. The retired load replays its value from the load queue entry (dark grey). The completed store (light grey) is discarded at decode. Its effects are already globally visible and redoing it would not only be unnecessary, but also incorrect.

Chapter 4

Dynamic Efficiency

Even with an efficient substrate, load latency tolerance may introduce too many instruction re-executions to be energy efficient. Each re-execution consumes energy as the instruction re-accesses pipeline structures. Energy efficient load latency tolerant designs must be aware of this pitfall and apply *pruning techniques* to limit re-executions, keeping energy overheads under control. The goal of such pruning techniques is to have as few re-executions as possible, without harming performance.

The key aspect of any pruning algorithm is properly identifying which instructions must re-execute during a given slice buffer pass and which are superfluous. Proper identification is crucial, as failure to execute an instruction in the proper pass will result in the appropriate register mappings—contained in the slice map table—to be lost, causing a functionally incorrect execution, which is unacceptable.

4.1 Miss Re-execution Pruning

In BOLT, re-execution passes are triggered by miss returns. The most naïve approach to re-execution is to re-inject all instructions from the slice buffer into the pipeline. Whenever any miss returns, many re-injected loads may still have their misses outstanding. Leaving these still outstanding loads in the issue queue would clog the window, so they must be sliced back out to maximize performance. Unfortunately, doing so leads to excessive re-executions and wasted energy. A more efficient approach is to apply *miss pruning* and avoid re-injecting instruction that do not depend on a returning miss. WIB introduced this optimization [44], but CFP did not include it [76].

In WIB, miss pruning is implemented by explicitly tracking which instructions depend on every outstanding load using a bit-vector per Miss Status Holding Register (MSHR). Each bit in the vector corresponds to a position in the WIB, and indicates if the corresponding instruction depends on the miss in question or not. When a miss returns, WIB uses the corresponding bit-vector to determine which instructions it should re-inject and re-execute.

A WIB-based must use per MSHR bit-vectors to perform this tracking, as the WIB is large (2K-entries) and sparse—examining each instruction at a reasonable bandwidth would take many hundreds of cycles. For miss-pruning, this approach is sufficient, however, a different approach is required for the more sophisticated pruning algorithms employed by BOLT.

4.1.1 Implementation: Antidotes

Instead of computing persistent dependence of instruction on outstanding loads during initial execution, BOLT recomputes dependences during each re-execution pass over the slice buffer. BOLT can take this approach—which examines each deferred instruction at rename bandwidth—because it uses a dense slice buffer. In exchange, this approach uses less hardware and supports some additional pruning optimizations. To identify load dependence instructions during a slice buffer pass, BOLT propagates antidotes—the opposite of poison—through register and store-to-load data dependences.

BOLT tracks one bit per-logical register indicating if an antidote is available for that register during the current slice buffer pass. At the start of a pass, the bit-vector is set to all 0s, indicating that no antidotes have been encountered. As BOLT examines each instruction in the slice buffer, it determines whether to re-inject the instruction and updates the

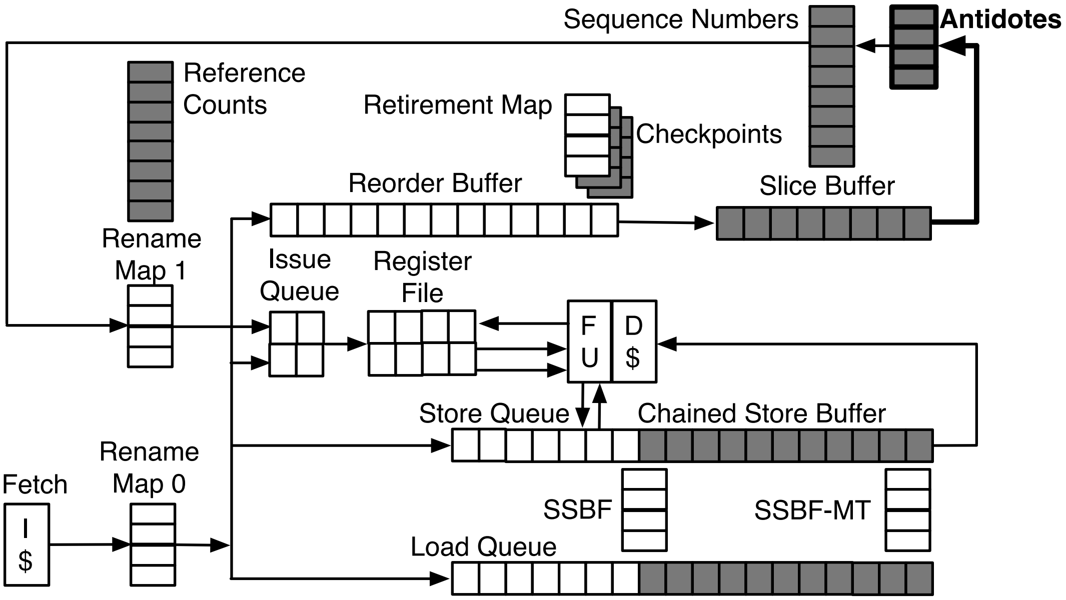


Figure 4.1: Structural diagram of BOLT with the antidote bit-vector highlighted.

Instruction type	Re-inject?	Destination register antidote
Returning load	Yes	Set
No poison inputs	No	Clear
At least one poison input has an antidote	Yes	Set
No inputs have antidotes	No	Clear

Table 4.1: Antidote bit-vector manipulation rules to implement miss-pruning. The rules are listed in order of precedence.

antidote bit-vector to reflect whether or not the examined instruction provides an antidote. The rules for manipulating antidote bits and determining re-injections for miss pruning are presented in Table 4.1. Note that the first instruction examined in any pass is a load which has just returned, providing an antidote, and setting a bit in the antidote bit-vector. The antidote bit-vector is highlighted on the structural diagram of BOLT in Figure 4.1.

As re-execution passes occur, the slice buffer may develop "holes," BOLT does not attempt to compact the slice buffer, instead allowing it to become sparser with each pass. However, even if the slice buffer becomes relatively sparse, it still remains dense enough for BOLT's pruning mechanisms to be efficient—only miss-dependent instructions ever enter the slice buffer, so re-execution passes still need to examine many fewer entries than if all instructions were tracked in the slice buffer.

4.1.2 Store-to-load Dependences

Dependences between instructions exist not only through registers, but also through memory. This form of communication means that a poisoned store may propagate poison to a younger load which reads the address written by the store. Correspondingly, the store must be able to propagate an antidote to the load during slice re-execution so that the load can determine whether it should re-execute. While store-to-load dependences are typically problematic to track, slice re-execution has a significant advantage—the store-to-load dependence has already been identified during the initial execution, and can simply be remembered for slice re-execution. Put a different way, if a load was poisoned by a store, it knows the identity of the store that poisoned it. Effectively, the store-to-load dependence simply acts as if it were another input when applying the antidote bit-vector rules to a poisoned forwarding load.

This observation means that BOLT can perform a direct analog of its register dependence mechanism, by simply checking whether the store buffer entry is still poison or not. If the forwarding store is no longer poisoned, it is considered to be providing an antidote to the load in question. Effectively, the same antidote algorithm is used, but there is no need to represent antidote bits explicitly since all in-flight stores are present in the store buffer. The one important difference is that the forwarding store may receive an antidote, execute, commit, and then complete to the data cache before the slice buffer pass reaches the forwarding load. In this situation, the forwarding load must receive an antidote and re-execute. Fortunately, this situation is easy to detect by simply comparing $\text{SSN}_{\text{forward}}$ to $\text{SSN}_{\text{store-complete}}$—any stores completed to the data cache must no longer be poisoned, and should be treated as providing antidotes to younger loads.

4.1.3 Restarting A Re-execution Pass: Optimizing Loads that Return Out of Order

A side benefit of antidote bit-vectors is that they can be used to tell when all slice-produced register values have been clobbered by younger instructions. This information is useful if an older load returns in the middle of a slice buffer pass. In this situation, it is typically desirable to re-initiate slice processing at the older load. However, doing so would destroy the current slice map-table. Destroying the slice map-table is only safe if no younger instructions need values produced by the slice. While determining the consumption behavior of younger instructions is difficult in general, it is simple in the special case when all slice-produced register values are known to have been clobbered. This observation means that an empty (i.e. all 0) antidote bit-vector serves as a simple test for whether or not restarting slice processing at an older load is safe.

Note that while restarting a slice buffer pass at and older load is desirable, simply stopping a pass before reaching the end of the slice buffer is not—there may be younger returned loads, which must eventually be processed.

Store-to-load dependences and their corresponding antidotes provided via the store buffer do not affect this optimization. If the slice buffer pass is restarted in the middle while a store to load dependence is live, the dependence will be detected and handled properly in the new pass. The important difference here between store-to-load dependences and register dependences is that register dependences must be reconstructed by the slice map table, whereas store-to-load dependences are persistent in the store buffer.

4.1.4 Need For More Sophisticated Pruning

Miss pruning by itself is not sufficient for efficiency. Even with miss pruning, instructions which depend on multiple loads misses may re-execute once per load that they depend on, as each miss returns. The worst case observed in experiments with Spec2006 is *mcf* which experiences almost a factor of 4 increase in executions for a mere 10% speedup. Such a

C Code:

```
for (int i = 0; i < N; i++) {
   total += array[i];
}
```

Assembly Code:

```
loop:
  load [r1] -> r2
  add r2 + r3 -> r3
  add r1 + 4 -> r1
  bnz r1, r5 loop
```

Data-flow Graph:

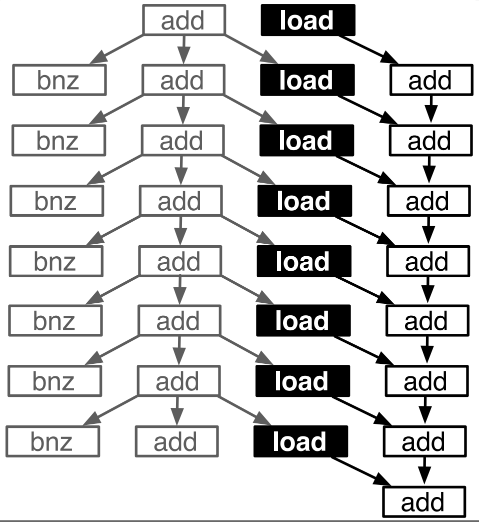


Figure 4.2: Top Left: C code to sum the elements of an array. Bottom Left: The corresponding assembly code. Right: The code pictured in a data-flow graph representation.

trade off is *definitely not* energy efficient.

To improve energy efficiency, BOLT includes more sophisticated pruning techniques than simply miss pruning. Theses pruning techniques are based on the observation that multiple re-executions require dependence on multiple load misses. Fundamentally, dependence on multiple load misses may come in one of two forms: parallel (*i.e.* independent) load misses, or serial (*i.e.* dependent) load misses. These two cases have very different behaviors, and must be dealt with in different ways.

4.2 Join Re-execution Pruning: Optimizing Parallel Load Misses

The superfluous re-executions which miss pruning does not handle are those that arise when a given instruction depends on multiple load misses. The first way in which this can occur is when computations combine the outputs of independent, parallel loads. For example, consider the C code on the top-left side (and corresponding assembly code on the

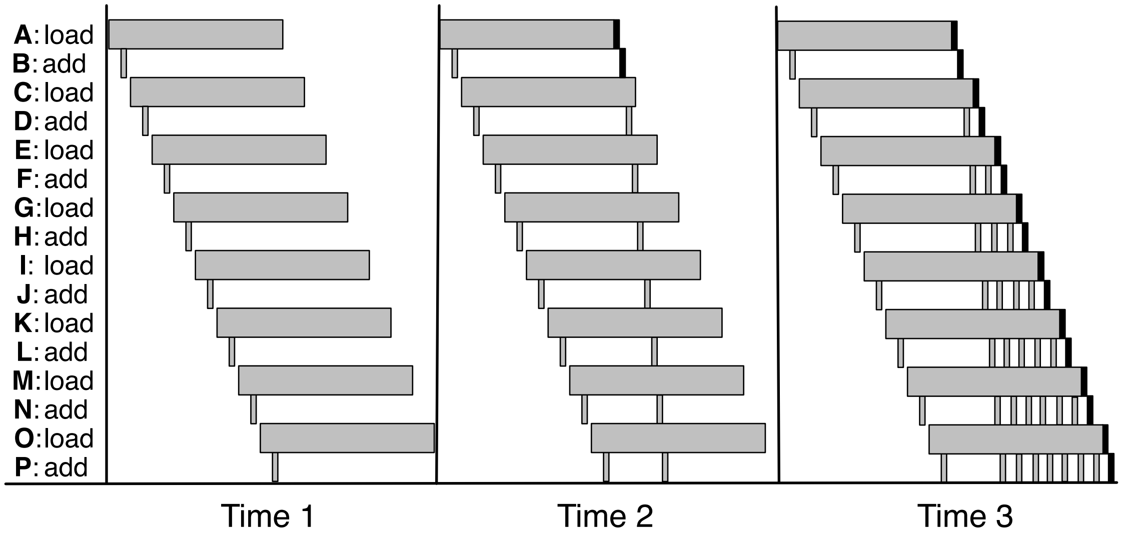


Figure 4.3: Example execution with parallel loads. Grey block represent pseudo-executions. Black blocks represent true executions. Time 1: loads and dependents slice out. Time 2: A's miss returns. A and B re-execute, D captures a value, F,H,etc needlessly re-cycle thru the pipeline. Time 3: After all misses return, many instructions have executed several times.

bottom-left side) of Figure 4.2 which sums the elements in an array[1]. The `add` instruction which accumulates the sum depends on the loads in all previous iterations. This fact, as well as the independence of the loads can be seen more clearly when multiple loop iterations are pictured as a data-flow graph, as in the right side of Figure 4.2. The loop control instructions are un-interesting. They are shown in lighter color, and will be elided in the remainder of the example.

Figure 4.3 shows the behavior of this code under load latency tolerant execution in three time steps. In the first time step (left side), the loads miss the cache, and slice out along with their dependent instructions. This behavior improves performance, as the loads are all independent, generating MLP.

In the second time step (middle), the first load's (A's) miss returns, and all instructions which depend upon it re-execute. This set of instructions includes not only B, which executes correctly, and D, which must capture B's output, but also F, H, J, L, N, and P

[1] Note that while actual hardware would prefetch this address pattern easily, a simple example is used here in the interest of space and clarity.

Instruction type	**Re-inject?**	**Destination register antidote**
Returning load	Yes	Set
No poison inputs	No	Clear
All poison inputs have antidotes	**Yes**	**Set**
One poison input has an antidote	**Yes**	**Clear**
No inputs have antidotes	No	Clear

Table 4.2: BOLT's rules for manipulating antidote bit-vectors. The rules are listed in order of precedence. Rules in bold-face differ from miss-pruning.

which simply cycle into and back out of the pipeline.

The third time step (right) shows what has happened after the last load's miss returns. All instructions have correctly executed, but the younger instructions which depend on many loads have executed a significant number of times (*e.g.* 8 executions for N, 9 executions for P). These re-executions consume energy needlessly, as many of the instructions do not even need to capture an input. Additionally, they may reduce performance gains, as pipeline bandwidth is consumed by cycling these instructions through, rather than doing useful work that can be speculatively retired.

4.2.1 Implementation: Refining Antidote Propagation

To solve this problem, BOLT refines the rules of antidote propagation to ensure that instructions only receive an antidote and re-execute if a new input is *actually* available. While many of these rules are similar to those in Table 4.1, the case where at least one poisoned input receives an antidote has been sub-divided. This refinement—which distinguishes the case where an instruction simply captures an input but is still poisoned, from the case where an instruction executes and produces its correct value—is the key to *join pruning*. Join pruning eliminates superfluous re-executions where slices join together, and instructions depend on both returning and outstanding loads. Join pruning extends and naturally subsumes miss pruning.

The left side of Figure 4.4 shows the propagation of the antidote produced by A, under the refined rules of join pruning. A's return provides an antidote to B. B's only poison

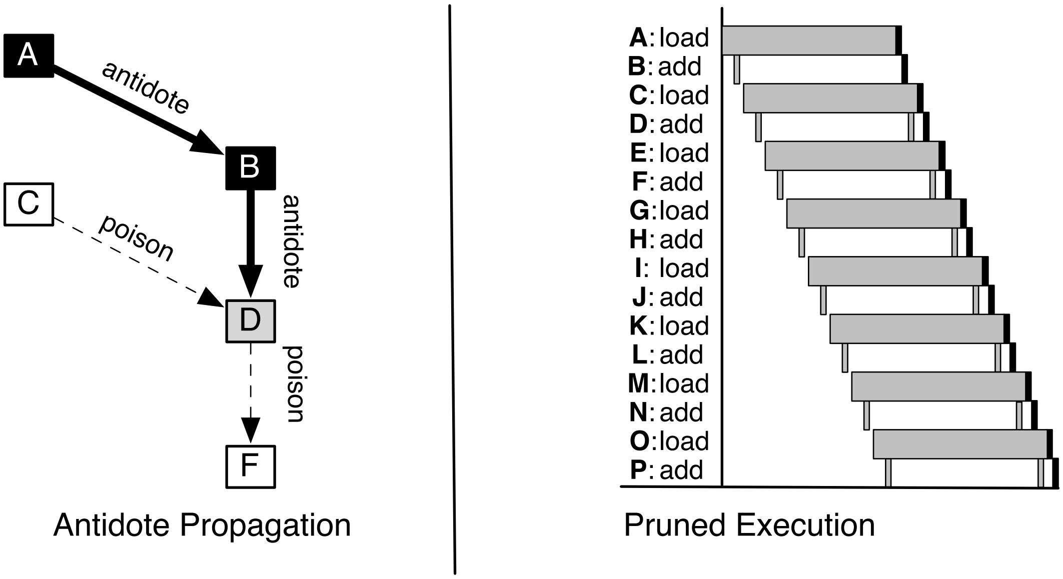
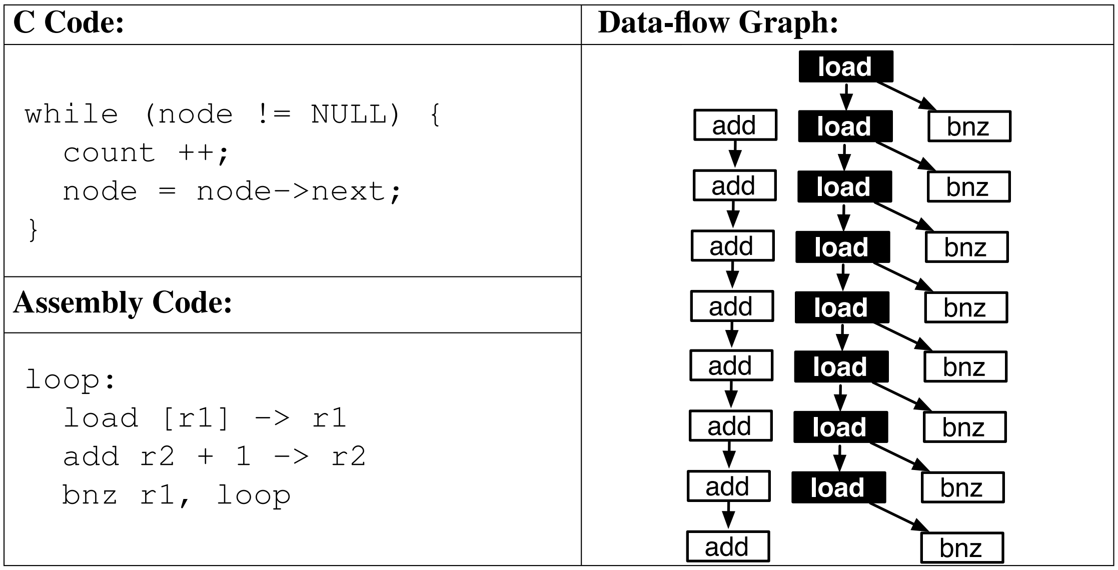


Figure 4.4: Left: A's return produces an antidote. B propagates this antidote to D. D must pseudo-execute to capture B's output, however, D's output is still poison. No instruction provides an antidote to F, so F should not re-execute. Right: The pruned execution of the previous example. Instructions for which no antidote is available are not re-injected into the pipeline.

input is receiving an antidote, so it can execute correctly and produce an output value. As B's output is no longer be poisoned, B provides an antidote to its dependents, specifically D. D receives an antidote from B, however, its other input is still poisoned by outstanding load C. Here is where join pruning improves on miss pruning. The fact that D is receiving *an* antidote means that it must re-execute to capture an input value (from B). However, the fact that it still has a poisoned input means that its output is still poison—it cannot provide an antidote to its dependents. The fact that F, (and transitively, H, J, etc.) do not receive any antidotes indicates that there is no reason to re-execute them during the current pass—they should wait for more loads to return.

The right side of Figure 4.4 shows the execution of the entire example when re-executions are join pruned—an instruction is not re-injected into the pipeline if no antidote is available for that instruction. Contrasting the right side of Figure 4.4 with the third time step of Figure 4.3 shows the significant number of superfluous re-executions that can be eliminated.



C Code:

```
while (node != NULL) {
  count ++;
  node = node->next;
}
```

Assembly Code:

```
loop:
  load [r1] -> r1
  add r2 + 1 -> r2
  bnz r1, loop
```

Data-flow Graph:

Figure 4.5: Top Left: C code to count the elements of a linked list. Bottom Left: The corresponding assembly code. Right: The code pictured in a data-flow graph representation.

4.3 Pointer-chasing Deferral Pruning: Optimizing Dependent Load Misses

The second way in which an instruction can depend on multiple loads is if it depends on a chain of loads in series. For example, the C code on the top-left side (and correspondingly the assembly on the lower-left side) of Figure 4.5 counts the elements in a linked list. The right side of Figure 4.5 depicts this code as a data-flow graph, making it easy to see the chain of loads upon which younger instructions depend.

Attempting to apply load latency tolerance to this linked list traversal (*i.e.* "pointer chasing") results in significantly more re-executions with *no* corresponding increase in performance. This behavior is shown in Figure 4.6. Initially, load A and all of its dependents slice out. The only independent instructions are the `add` instructions, which may execute earlier than normal. However, these `adds` are non-critical and do not affect performance. As no independent loads are encountered, no MLP is generated. These two facts together mean that load latency tolerance is providing no performance benefit in this

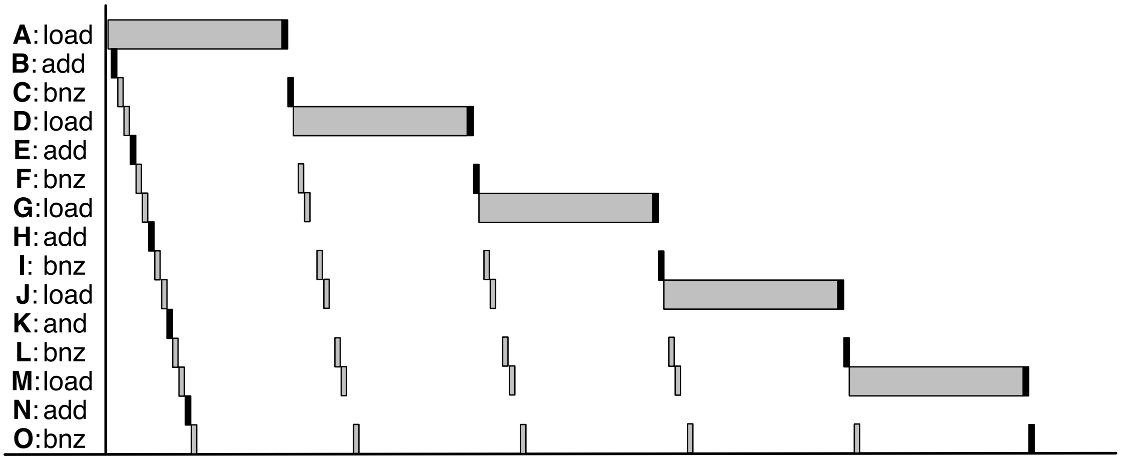


Figure 4.6: Example execution with pointer chasing loads. Load latency tolerance is unable to improve performance as no MLP exists and the only independent instructions are the non-critical adds. Energy consumption increases significantly as instructions are re-executed many times.

situation. When A's miss returns, A and C can execute correctly, but now D's miss is detected. D and its dependents slice back out. This process repeats with a `load` and a `bnz` completing when a miss returns, a new miss being detected, and *all* dependent instructions slicing back out. All of these re-executions consume significant energy—a pointless waste considering performance does not improve.

With no performance benefit, the natural approach to this problem is to simply shut down latency tolerance and revert to conventional execution—doing no harm, the architecture version of the Hippocratic Oath. The challenging part of this approach is detecting exactly those scenarios under which latency tolerance is harmful. A naïve approach is to simply detect when loads receive poison along their address inputs and prevent them from slicing out. The problem with this approach is that it is too broad—it covers not only the harmful case of pointer chasing, but also benign cases of load misses in series.

One benign example of load misses in series is shown Figure 4.7. Here, an object pointer is loaded from an array[2], and the result of that load is used as the address from which a field in the object is loaded. Contrasting the data-flow graph on the right side of Figure 4.7 with the one from Figure 4.5 shows the drastically different characteristics

[2] Again, the example is intended for simplicity—real hardware would prefetch the array accesses.

C Code:

```
for (int i = 0; i < N; i++) {
  x += objects[i]->val;
}
```

Assembly Code:

```
loop:
  load [r1] -> r2
  load [r2] -> r3
  add r4 + r3 -> r4
  add r1 + 4 -> r1
  bne r1, r5 loop
```

Data-flow Graph:

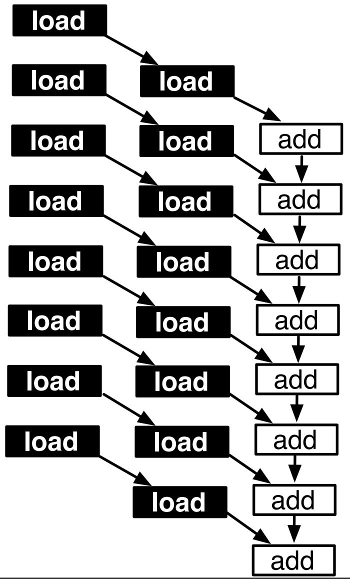


Figure 4.7: Loads in series which result from a simple level of indirection, not pointer chasing. The chains of loads are short, and many independent chains exist, providing MLP.

of this simple indirection versus pointer chasing. The most significant difference is that independent loads exist, providing the opportunity to expose MLP and increase performance. Additionally, the length of the serial load chains is much smaller—two instead of the length of the list—meaning that the number of re-executions is also much smaller. These differences mean that while load latency tolerance is strictly harmful in the case of recurrent indirection, it is beneficial in the case of non-recurrent indirection.

4.3.1 Implementation: Extended Antidote Bit-vectors

The key distinction between these two examples is that in the case of pointer chasing, poison is produced and consumed by different dynamic instances of the *same* static load. This behavior does not exist in the case of non-recurrent indirection, in which the poison is produced and consumed by dynamic instances of *different* static loads. This distinction is significant as it gives a simple test for pointer chasing—if a load consumes poison which was originally produced by and older dynamic instance of the same static instruction, the load exhibits pointer chasing.

Once pointer chasing is recognized, conventional execution can be re-established by preventing pointer chasing loads from originating poison. More specifically, the processor tracks a small table of the PCs loads which are known to participate in pointer chasing. When a load executes and misses the cache, its PC is compared against those in the table. If the PC hits in the table, the load is not permitted to poison its output and slice out. Instead the load and its dependents wait in the issue queue for the miss to return as they would in a conventional processor.

There are two possible ways for the processor to implement the test for pointer chasing and populate this table of known pointer chasing loads. The first is to simply extend the poison information with the low-order bits of the PC to identify which static instruction originated the poison. Under this approach, when a load consumes poison on its address input, the PC of the load which originated the poison is compared against the PC of the executing load. On a match, the executing load's PC is entered into the pointer chasing table, and stalls rather than slicing out. Whereas this approach has the advantage that it identifies and shuts down pointer chasing immediately, its disadvantage is that it adds complexity to the latency critical execution core—specifically, it increases the number of bits of poison information in the register file and bypass networks.

The second approach, which BOLT takes, places the complexity on the non-critical slice-in path. This approach is based on the observation that an analogous pattern exists during antidote propagation. Specifically, the test for pointer chasing can be rephrased as a static load initiating an antidote and then providing that antidote to a younger dynamic instance of itself. Under this approach, the entries in the antidote bit-vector are extended to hold the low-order bits of the PC. When a returning load sets the antidote bit-vector entry, it also fills in the extended entry with its PC. An address poisoned load receiving the antidote compares its own PC to the extended antidote information. If the PCs match, then the PC is entered into a small table of PCs which are forbidden from originating poison in the future (including the current re-execution of the load being sliced in). The disadvantage of this approach is that the first time a piece of pointer chasing code is encountered, latency tolerance is applied until the first miss returns, wasting energy. This disadvantage is small as the offending PC is learned quickly, and conventional execution is applied in all subsequent encounters.

Unrolled loops. Loop unrolling does not typically interfere with pointer-chasing pruning. If BOLT encounters a pointer chasing loop which has been unrolled N times, it will still learn the loads involved, but will take N times as many iterations to learn. In the first iteration, the return of the first load's miss will initiate an antidote, which will propogate around the loop—passing through the dependent loads introduced by loop unrolling—and reaching another dynamic instance of itself. The load of the second (unrolled) iteration will then undergo the same process, and BOLT will learn its pointer chasing behavior. BOLT will repeat this process until it learns the behavior of each pointer chasing load, then it will execute conventionally. Loop unrolling can interfere with the pointer chasing detection mechanism if the loop is unrolled so many times that either the there are more pointer chasing loads in the unrolled loop than BOLT's table of pointer chasing loads can hold, or that the loop body becomes so large that BOLT does not encounter another instance of the original static load before its miss returns.

4.3.2 Design Choices

One non-issue with both of these approaches is how to handle the extended poison/antidote information when two slices join together. Specifically, the possibility exists for an instruction to receive an antidote (or poison) on one input originated by one PC, and an antidote on the other input originated by another PC. In such a situation, it does not matter which input's extended antidote information is propagated to the output—the existence of multiple slices joining together from different origins indicates that MLP is being extracted, and pointer chasing is highly unlikely to be present.

Another situation which is likely to be a non-issue is the risk that there might be some situation in which pointer chasing is present, but load latency tolerant execution is beneficial to performance. This scenario could occur if a piece of code had pointer chasing interleaved at a fine granularity with other independent load misses. In this hypothetical scenario, the fact that BOLT disables latency tolerance for the pointer chasing code prevents it from reaping any benefits of latency tolerance from the independent misses—the pointer chasing code will fill the issue queue and prevent the extraction of MLP from the independent loads. The possibility of such scenarios may suggest that BOLT's pointer chasing pruning risks missing a significant performance opportunity.

However, such performance opportunities are likely quite rare—speeding up the independent loads is likely to have little impact on overall performance, as the serial, pointer chasing code will not speed up. Even if performance improvements were obtainable in such a case, they would need to be very large to outweigh the energy increase incurred by the significant re-execution overhead posed by applying latency tolerance to pointer chasing. The risk that BOLT might miss some performance opportunity in a corner case is far outweighed by the need to prevent pathological behavior in the common case.

A third hypothetical issue is whether, when pointer chasing is detected, to stall on the first load in the chain, or the second one—which consumes an address poisoned input. The fact BOLT stalls the first load in the chain before it originates poison is advantageous in that the pointer chasing code fully reverts to conventional execution. If the second load in the chain stalls when it consumes an address-poison input, the first load and its direct dependents still have latency tolerance pointlessly applied to them. The only point to stalling on the second would be to hope to uncover independent instructions. However, as just discussed, such cases are unlikely to yield performance gains.

Chapter 5

Performance Analysis

This dissertation uses cycle-level simulation to compare the performance and energy-efficiency of BOLT to those of three other architectures—a conventional ROB baseline, Runahead Execution, and Continual Flow Pipelines. The simulator executes the user-level portions of statically linked 64-bit x86 programs. It decodes x86 macro instructions and cracks them into a RISC-style μop ISA. The μop cracking used in Core i7 is proprietary, so there is no way to tell how similar the μop ISAs are. Table 5.1 describes the configurations used for each micro-architecture. The baseline ROB processor is designed to be similar to Intel's Core i7 processor. While the lack of proprietary knowledge prevents modeling all details of Core i7, the memory hierarchy (large L3 cache split into banks on a ring interconnect with private L2/L1), and structure sizes (ROB, LQ, SQ, IQ, etc.) are modeled using publicly available information.

The simulated micro-architecture deviates from Core i7 in two major ways. First, instead of using a 2-level branch predictor, it uses a three table PPM (prediction by partial matching) [49] predictor. The second difference is that the simulated baseline uses SVW [67] and SQIP [74] load and store queues. The use of SQIP and SVW load/store queues has little impact on performance and allows a more direct energy comparison between the baseline and BOLT.

		Baseline	Runahead	CFP	BOLT
	Clock	3.2 GHz			
Front-end	Bpred	3-table PPM: 256x2, 128x4, 128x4. 8-bit tags, 2-bit counters			
	Fetch	16 bytes/cycle. 3 cycle latency			
	Decode	Max 4 macro ops/6 μops per cycle. 4 cycle latency			
	Rename	Max 4μops per cycle. 2 cycle latency			
	Dispatch	Max 4μops per cycle. 1 cycle latency			
Window	IQ	36 entries			
	Registers	224 physical registers(96 rename + 2 * 64 arch.)			
	ROB	128	128	—	128
Execute	Issue	4-wide. Speculative wakeup.			
	RegRead	2 cycles.			
	Int FUs	4 ALU. 1 branch. 1 load. 1 store. 2 mul/div			
	FP FUs	2 ALU/convert. 1 mul. 1 mul/div/sqrt.			
Memory Hierarchy	L1 I$ Prefetcher	32KB. 128-set. 4-way. 64-byte blocks. 3 cycles 2-streams, 4 blocks each			
	L1 D$ Prefetcher	32KB. 64-set. 8-way. 64-byte blocks. 8-entry VB. 3 cycles 4-streams, 4 blocks each			
	L1 ↔ L2 bus	32-bytes/cycle. 1 cycle.			
	L2$ Prefetcher	Private 256KB. 512-set. 8-way. 64-byte blocks. 16-entry VB. 10 cycles 8 streams. 16 blocks each. Prefetches into L3			
	L2 ↔ L3 bus	8-stop bi-directional ring. 8-bytes/cycle/hop. 2.0GHz clock			
	L3$ Each bank	Shared 8MB. 4-banks at different ring-stops. 2048-set. 16-way. 64-byte blocks. 32-entry VB. 25 cycles (2.0GHz).			
	Mem. Bus	800MHz. DDR. 8-bytes wide. Dual channel. 16ns latency			
	Memory	tCAS: 10ns. tRAS: 10ns. tRP: 10ns			
LT	Slice buffer	—	—	384	256
	Checkpoints	—	1	8	2
Ld/St Queues	SQ type	SQIP	SQIP	Assoc + SRL	SQIP/CSB
	SQ size	32	32	32/128	128
	LQ type	SVW	SVW	Assoc + 8-way set-assoc	SVW
	LQ size	48	48	48 / 256	256
	Fwd $	—	Ideal	Ideal	—

Table 5.1: Simulated processor configurations

A few other details in Table 5.1 merit explanation. First, CFP is provided with a 384-entry slice buffer, while BOLT uses a 256-entry slice buffer. This difference arises from the fact that BOLT has a ROB, which also acts as (sparse) extention to the slice buffer. To prevent an unfair advantage, CFP's slice buffer is increased by the size of BOLT's ROB. Second, CFP has eight checkpoints while BOLT only has two. This difference is because CFP needs checkpoints for its entire window, while BOLT only needs them for the deferred region. With only two checkpoints, CFP's performance degrades dramatically [30].

Additionally, the implementation of CFP's load and store queues are idealized in several ways. First, CFP is given an ideal forwarding cache—one in which conflicts never cause entries to be evicted. This idealization not only gives the benefit that forwardings from the cache are always correct, it also removes the need to verify forwardings from the SRL later (*i.e.*, search the load queue on each store completion). Second, SRL's Loose Check Filter (LCF) is assumed to be perfect. Third, the migration of loads from the fully associative load queue to the set-associative load queue is "free" in terms of both energy and performance. No extra energy costs are modeled for the reads and writes required to copy the data, beyond the addition of an indexed read port which does not otherwise exist on the fully associative queue. No performance penalty is modeled for possible write bandwidth contention between load execution and loads migrating from one queue to the other.

Benchmarks. BOLT targets single-thread performance and energy-efficiency, so the evaluation uses the SPEC2006 benchmarks[1]. The benchmarks were compiled with gcc-4.1 at optimization level -O3. The benchmarks are executed on either their test or their training inputs depending on the simulation time required for the training input. To determine which input to use, benchmarks were simulated in functional mode on their training inputs. If the simulation time required for the training input was less than two hours, the training input was used in all experiments. Otherwise, the test input was selected. The benchmarks are executed to completion using 2% periodic sampling. The first 50 million instructions

[1] *perlbmk* and *bwaves* are omitted because we cannot run them at all—natively, nor in simulation.

are skipped, then each sampling period proceeds with ten million instructions warming up all predictors and caches, ten million instructions of sampled timing simulation, and 480 million instructions of functional simulation. Table 5.2 lists the SPEC2006 benchmarks as well as which input was used and the benchmarks' characteristics—IPC, speedup from using DVFS to increase clock frequency to 4.0 GHz, L2 and L3 Misses Per Thousand Instructions, and L2 and L3 MLP—on the baseline ROB processor.

The baseline's simulated performance was compared against an actual Core i7 of the same clock frequency. The point of these experiments is not to try to establish that the baseline precisely models Core i7, as it does not—without detailed knowledge of the micro-architecture, a faithful implementation is difficult to impossible. Instead, the purpose is to establish that the baseline models a reasonable microarchitecture by contemporary standards. These correlations show an average (of absolute values) of error of 15.2% with associative load and store queues and 15.7% with SVW/SQIP. These errors include sampling as well as simulation errors. Note that a simulation error in the baseline does not translate into a number which may be added or subtracted from a speedup derived in an experiment to determine the "real" speedup. The source of the baseline error may be orthogonal to the performance impacts causing the speedup in the experiment. A more important factor is the fidelity with which the micro-architectural (*i.e.*, Runahead, CFP, and BOLT) changes are modeled. Unfortunately, those errors are impossible to measure without an actual chip for that micro-architecture to compare against.

As load latency tolerance is intended to help memory-bound programs, our evaluation focuses on them. These programs (shown in bold in Table 5.2) are those for which increasing clock frequency (*i.e.*, DVFS) has a relatively small impact on performance. Specifically, we define the set of memory-bound programs as those programs for which increasing the clock frequency by 25% (from 3.2 GHz to 4.0 GHz) improves performance by less than 15%. We report averages over these memory-bound programs, as well as over all of SPEC2006.

Energy and ED^2 methodology. In this dissertation, the energy-efficiency metric is

Benchmark	Input	IPC	TB Spdup	L2 MPKI	L2 MLP	L3 MPKI	L3 MLP
cactusADM	test	1.1	13.5%	4.7	4.1	2.7	4.4
calculix	train	1.5	21.9%	2.2	1.9	0.5	3.7
dealII	test	1.6	21.7%	2.7	1.8	0.3	3.4
gamess	test	1.8	22.0%	1.6	1.4	0.1	2.5
GemsFDTD	test	1.1	12.3%	10.1	2.5	3.0	4.1
lbm	test	0.6	7.4%	38.8	5.0	18.1	4.4
leslie3d	test	1.8	20.3%	9.4	2.4	0.0	3.1
milc	train	0.5	4.3%	24.5	2.5	14.7	2.8
namd	train	1.6	24.7%	0.1	1.7	0.0	2.9
povray	train	1.4	24.9%	0.1	1.5	0.0	2.8
soplex	train	0.6	6.6%	24.3	2.2	8.2	2.9
sphinx3	train	1.3	13.1%	14.2	2.1	1.2	3.5
tonto	test	1.6	24.7%	0.2	1.4	0.0	2.1
wrf	test	1.4	20.2%	5.8	2.2	0.5	4.3
zeusmp	test	1.0	13.6%	6.8	2.0	2.7	3.6
astar	test	0.8	22.0%	5.3	2.0	0.1	2.2
bzip2	train	1.4	22.3%	3.3	2.3	0.1	2.5
gcc	train	1.2	19.2%	3.4	1.4	0.5	1.9
gobmk	train	1.0	22.6%	2.0	1.5	0.2	1.6
h264ref	test	1.9	23.3%	1.8	1.5	0.0	1.7
hmmer	test	1.4	25.0%	0.0	1.3	0.0	2.3
libquantum	train	2.0	7.6%	22.0	3.0	0.0	3.9
mcf	train	0.2	4.4%	112.4	2.5	17.6	2.3
omnetpp	test	1.2	24.1%	0.2	1.2	0.1	1.5
sjeng	test	0.9	18.9%	7.7	3.0	1.4	7.7
xalancbmk	test	1.2	23.2%	2.6	1.6	0.1	3.0

Table 5.2: Benchmark characterization

ED^2 which is invariant under ideal conditions with DVFS [47]. We consider a technique to be energy-efficient if it reduces ED^2. We note that a performance technique that increases ED^2 still has value in an energy conscious environment but only in a few scenarios, *e.g.*, in the serial portion of a parallel application [29].

Because we are interested only in *relative* ED^2, we only need to measure *relative* energy consumption. This is fortunate because measuring absolute energy consumption using bottom-up techniques is difficult. We approximate relative energy consumption using relative area, relative execution time, and relative instruction execution counts. We justify this simple approximation by observing that the four architectures we are comparing are similar—all execute instruction in the same out-of-order core—and that the differences between them manifest as a few additional structures, most of which are RAMs. As we will show, the static and dynamic costs of the new structures account for only a few percent of the energy effects of BOLT (and load latency tolerance in general). BOLT's most significant effect on dynamic energy consumption is an increase in the number of instructions executed, not accesses to the new structures. Its most significant effect on static energy consumption is a reduction in execution time rather than leakage by new structures.

We assume that static energy accounts for 20% of total energy, and dynamic energy accounts for the remaining 80%. For architecture α,

$$\frac{E_{\alpha}}{E_{\text{base}}} = 0.2 \times \frac{E_{\alpha\text{-static}}}{E_{\text{base-static}}} + 0.8 \times \frac{E_{\alpha\text{-dynamic}}}{E_{\text{base-dynamic}}}$$

Appendix B evaluates the impact of this 20/80 assumption on the energy model.

Static energy. The first component of this equation, $E_{\alpha\text{-static}}$, represents static energy due to leakage currents. To approximate this component, we assume that static energy is directly proportional to area and execution time. The first part of this assumption—proportionality to area—is based on the premise that leakage is proportional to area. The second part of the assumption—proportionality to execution time—is based on the premise that supply voltage to the core is gated once execution is completed, preventing further leakage. If architecture α has marginal (*i.e.*, additional) area MA_{α} over base area

A_{base} and executes a program in time T_α, then we compute its static energy consumption as:

$$\frac{E_{\alpha\text{-static}}}{E_{\text{base-static}}} = (1 + \sigma \times \frac{MA_\alpha}{A_{\text{base}}}) \times \frac{T_\alpha}{T_{\text{base}}}$$

The factor σ parameterizes the model in terms of how much the new latency tolerance structures leak relative to the average structure on the core. In the model used in the rest of this chapter, we assume $\sigma = 1$. This assumption is based on the *conservative* premise that new structures added for load latency tolerance leak at the same rate as the average transistors in the core. In practice, load latency tolerance structures are likely to leak significantly less because most are SRAMs which are off the latency-critical execution path and can be implemented with low-leakage, high-threshold circuits. Appendix B shows that the energy model is insensitive to σ within a reasonable range because the dominant factor in static energy changes is the reduction in execution time—the structural overheads are quite small.

Dynamic energy. $E_{\alpha\text{-dynamic}}$, represents dynamic energy due to transistor switching. To approximate relative dynamic energy consumption, we consider two factors—the overheads of executing additional instructions on existing hardware structures, and the overheads of accessing new hardware structures. We use the following equation, where I_α is the number of instructions executed on architecture α as reported by the timing simulator:

$$\frac{E_{\alpha\text{-dynamic}}}{E_{\text{base-dynamic}}} = \frac{I_\alpha}{I_{\text{base}}} + \delta \times (\frac{MA_\alpha}{A_{\text{base}}} \times \frac{I_\alpha}{I_{\text{base}}} + \frac{MA_{\alpha\text{-mem}}}{A_{\text{base}}} \times \frac{I_{\alpha\text{-mem}}}{I_{\text{base}}} + \frac{MA_{\alpha\text{-lt}}}{A_{\text{base}}} \times (\frac{I_\alpha}{I_{\text{base}}} - 1))$$

The first term corresponds to the execution of additional instructions—primarily the re-execution of deferred miss-dependent instructions—on existing structures. The second, more complex, term correspond to accesses to new structures. This is scaled by a parameter δ. As with the static energy parameter σ, this allows the energy model to weight the new structures from load latency tolerance differently than the average structures on the chip. For the model used in the remainder of this chapter, we assume $\delta = 2$. This value of δ represents our *conservative* assumption that access to a new structure draw twice as much dynamic energy as access to an existing structure. Appendix B shows that varying δ within a reasonable range has little impact on the energy model, as most of the dynamic energy overhead comes from re-execution—the structural overheads are quite small.

The three sub-terms of the second term correspond to new structures accessed by all instructions, new structures accessed by loads and stores only, and new structures accessed by re-executing deferred instructions only. For BOLT, the only new structure accessed by all instructions is the reference count matrix. The larger load and store queues are accessed by load and store instructions. The new structures accessed by re-executing deferred instructions are the slice buffer, extensions to the ROB to capture values, the checkpoints, and the register sequence number tables.

Area models. We use CACTI-4.1 [79] to estimate structure areas and compute area overheads relative to the area of a Core i7 processor (roughly[2] 32 mm^2 in 45nm). Table 5.3 shows the area overheads (in mm^2) incurred by each design. Each entry shows the area of a structure if it is completely new, or the marginal increase for a structure which exists in the conventional baseline but is expanded or replaced. The table also shows the total area overhead, as well as that total as a percentage of the area of a single core in a Core i7.

Many of the entries in Table 5.3 are self explanatory, but a few require explanation. Runahead has much smaller physical register reference counting machinery than CFP or BOLT because it does not need to reference count register held by instructions in the issue queue, and only uses one checkpoint. BOLT's reference counting structures are smaller than CFP's because BOLT uses fewer checkpoints. BOLT's physical register sequence number table is smaller than CFP's due to the optimizations described in Section 3.2.3. While all designs (including the baseline) include branch confidence prediction [38], CFP requires a more sophisticated predictor than the other schemes, because its performance is sensitive to the efficient placement of checkpoints on low-confidence branches—for

[2] An entire Core i7 is 263mm^2. Approximately half of this area is consumed by the L3 cache. Each core is slightly less than one quarter of the remaining area, resulting in an approximation of 32mm^2 per core.

Structure	BOLT	CFP	Runahead
Checkpoints	0.006	0.006	0.004
Reference Counts	0.134	0.223	0.018
Forwarding Cache	—	0.085	0.085
ROB + Slicebuffer	0.168	0.168	—
Load Queue	0.069	0.228	—
Store Queue	0.049	0.180	—
Physical Register Sequence Numbers	0.058	0.082	—
Physical-to-physical Map Table	—	0.231	—
Confidence Predictor	—	0.074	—
Total	0.484	1.277	0.107
% of Core i7	1.5%	4.0%	0.33%

Table 5.3: Area Overheads (mm^2)

CFP, the PPM branch prediction entries are expanded to each hold a confidence counter. Additionally, while CFP and Runahead are given the benefit of perfect forwarding caches in the performance model, they are treated as 256 entry, 8 bytes per entry, 4-way set-associative for the purposes of modeling area.

5.1 BOLT Configuration: Load Latency Tolerance for L2 and L3 Misses

With a three-level cache hierarchy, there exists a choice between applying latency tolerance to L2 misses which hit in the L3 cache, and applying latency tolerance only to L3 misses. Before comparing BOLT to other designs, we fix a configuration for BOLT by evaluating this design choice.

Figure 5.1 shows BOLT's performance (top), re-execution overhead (middle), and relative ED^2 (bottom) when applied only to L3 misses (left, lighter bar) or to both L2 and L3 misses (right, darker bar). Note that in the performance graphs, higher is better, but in the re-execution and ED^2 graphs, lower is better. BOLT uses pruning in both cases

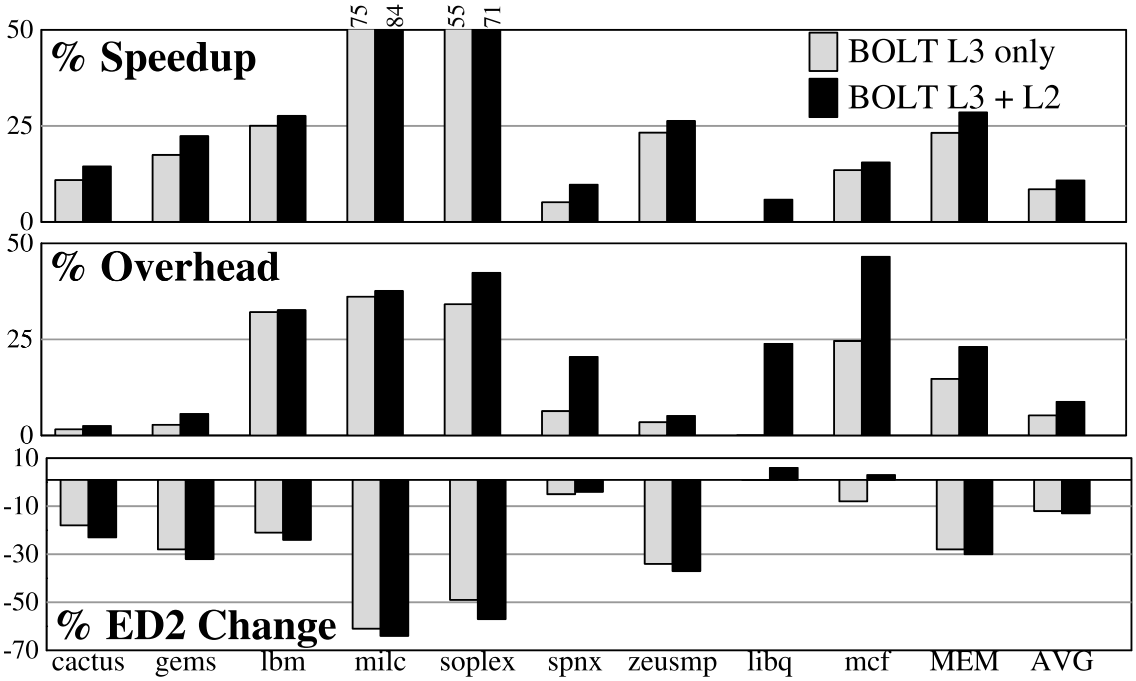


Figure 5.1: Applying BOLT to L3 misses only or to L2 and L3 misses

(Section 5.4.3 isolates the effects of pruning). Both performance and re-execution overheads are always higher when applying load latency tolerance to both L2 and L3 misses, compared to L3 misses only.

In BOLT, applying load latency tolerance to L2 misses improves *both* performance *and* ED^2. This sets BOLT apart from CFP (see Section 5.4)—where applying latency tolerance to L2 misses helps performance but hurts ED^2—and Runahead (see Section 5.3)—where applying latency tolerance to L2 misses helps ED^2, but only by pruning more and hurting performance. This difference arises from a combination of BOLT's efficient slice processing and pruning algorithms. With the moderate latency of an L3 hit, there is little margin for inefficiency to obtain performance.

BOLT has two benchmarks—*libquantum* and *mcf*—where applying latency tolerance to L2 and L3 misses has higher ED^2 than applying it to only L3 misses. Unlike most memory-bound benchmarks, *libquantum*—which almost exclusively experiences L3 hits— has a high baseline IPC—2.0. This means that, unlike many memory-bound programs, the contention for dispatch bandwidth between tail instructions and re-executing slice instructions poses a significant performance overhead. When applied to L2 misses on *libquantum*, BOLT reduces dispatch stall cycles due to a full issue queue from 38% to 8%. However, 14% of cycles are consumed re-dispatching miss-dependent instructions and another 8% are spent stalling to avoid resource inversions during re-dispatch. In *mcf*, a significant portion of the re-execution overhead comes from squashing more instructions when a mis-predicted branch resolves. This occurs because load latency tolerance allows more wrong-path instructions to enter the window and execute before the mis-prediction is detected, resulting in more instructions squashed and re-executed per branch mis-prediction.

5.2 BOLT vs. DVFS

BOLT is intended to complement DVFS *i.e.*, to improve the performance of memory-bound programs, for which increasing clock frequency provides little benefit. Figure 5.2 shows the performance improvement (top) and relative ED^2 (bottom) of DVFSing by 25% (from 3.2 GHz to 4.0 GHz), BOLT, and BOLT combined with DVFS (*i.e.*, BOLT at 4.0 GHz). Re-execution overhead is not shown because DVFS does not introduce re-execution overhead. The energy for DVFS is modeled as $1.56\times$ (*i.e.*, 1.25×1.25) the energy of the underlying micro-architecture.

This graph shows that BOLT is, in fact, a good complement for DVFS. First, BOLT can achieve significant performance improvements in cases where DVFS cannot. The most significant examples are *milc* and *soplex*, where BOLT produces 85% and 71% speedups respectively, while DVFS only obtains 4% and 7%. Even in cases where BOLT and DVFS obtain similar performance improvements (*e.g.*, *cactus*, *sphinx*, and *libquantum*), BOLT obtains that performance in a more energy-efficient way—BOLT's ED^2 is significantly lower than DVFS's.

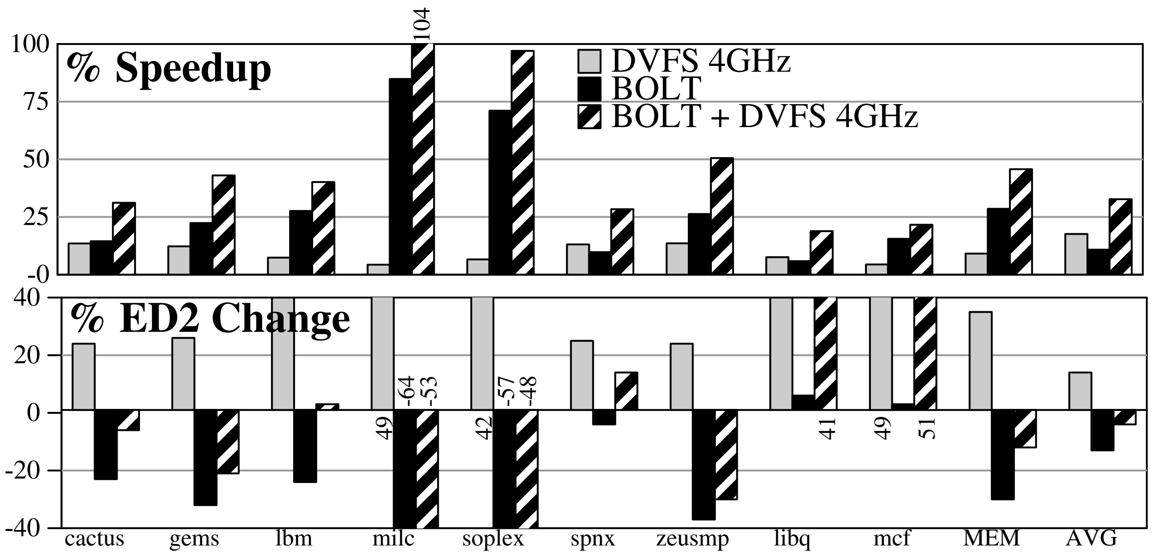


Figure 5.2: BOLT vs. DVFS

Second, DVFS and BOLT show an interesting synergy. Specifically, the speedups obtained by applying both techniques are *larger* than the product of the speedups obtained by applying each technique individually. This super-cumulative behavior indicates that not only do the techniques address different aspects of performance—DVFS accelerates computation-bound regions while BOLT accelerates memory-bound regions—but that they exhibit a further synergy. BOLT's ability to tolerate cache misses makes the program "un-memory-bound," exposing more opportunities for DVFS to improve performance.

Other Clock Frequencies. To further explore the relationship between BOLT and clock frequency, Figure 5.3 examines the behavior of BOLT and the conventional baseline at various frequencies ranging from 2.0 to 4.8 GHz. The left two show averages over the memory-bound programs, while the right two show averages over all of SPEC2006. The top pair of graphs show performance improvement relative to the 3.2GHz baseline, and the bottom two show relative ED^2. As with other results, the general shape is the same, but the trends are more pronounced when only memory-bound programs are considered. BOLT outperforms the baseline and has better ED^2 at all points, but the most important trend is the divergence of the curves as clock frequency increases. As the clock frequency

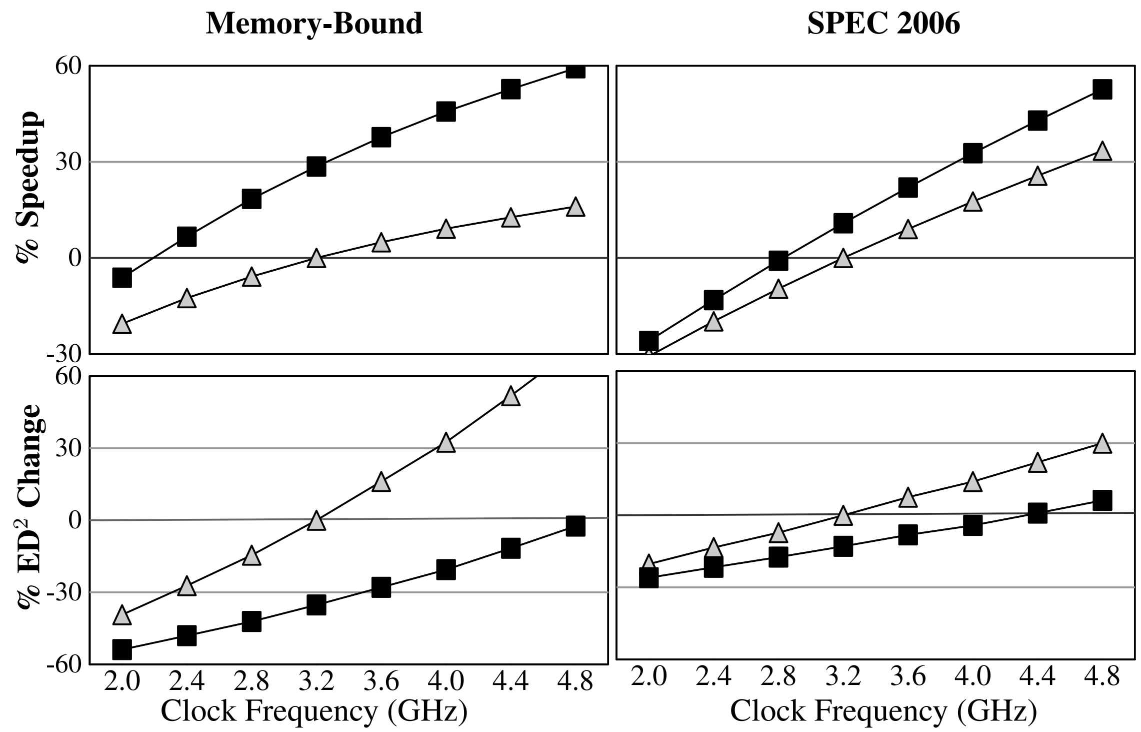


Figure 5.3: Performance of BOLT and Baseline at various clock frequencies

increases—and correspondingly memory and L3 hit latencies increase—BOLT's performance and energy-efficiency gains also increase. Even though processor designers are not aggressively increasing clock frequencies at the time, this result is significant for two reasons. First, future systems might see increases in memory latency without changing clock frequency. For example, larger, more complex interconnection networks to support large numbers of cores may add to memory latency. Second, although other impediments remain—most significantly dynamic power—this change in performance/frequency tradeoffs is one step towards making higher clock frequencies more attractive.

5.3 BOLT vs. Runahead

Figure 5.4 shows the performance impact (top), re-execution overheads (middle), and ED^2 (bottom) of applying Runahead to L3 misses only or to L2 and L3 misses, both with and without pruning, and compares these configurations to BOLT (right/dark bar). BOLT is

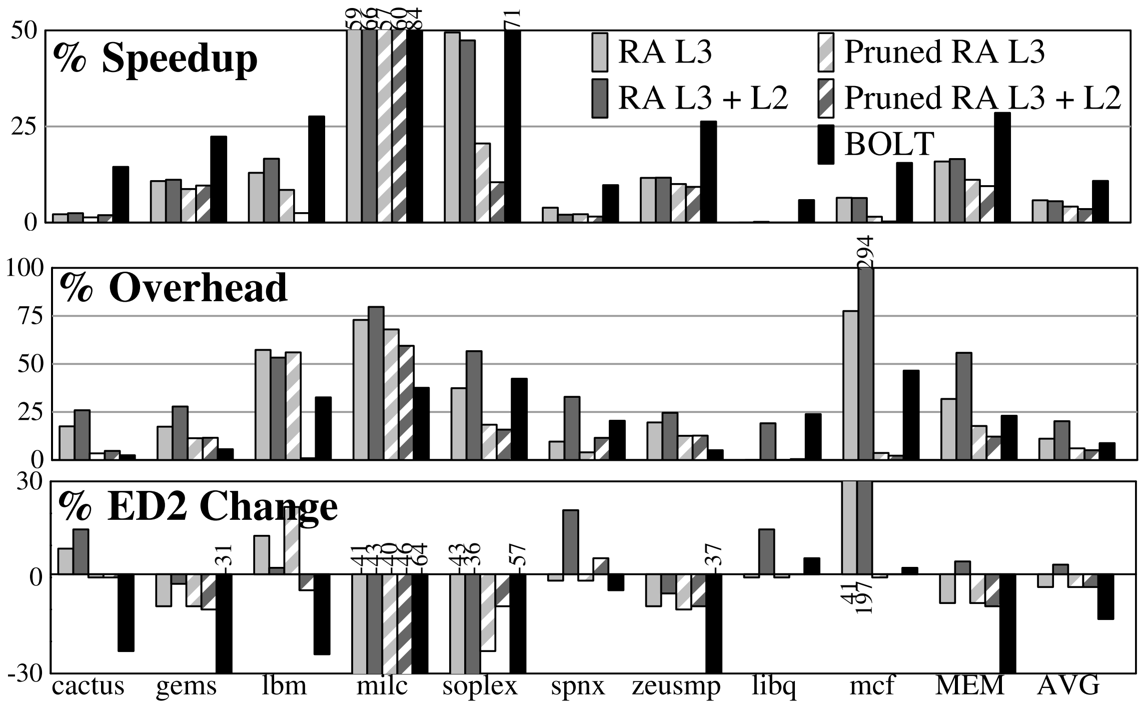


Figure 5.4: Applying Runahead to L3 misses only or to L2 and L3 misses with and without pruning.

applied to both L2 and L3 misses.

Performance. Without pruning, Runahead typically sees slightly higher performance when applied to both L2 and L3 misses than when applied to only L3 misses. There are exceptions (*e.g.*, *soplex* and *sphinx*) which occur when the performance penalty from additional squashes outweighs the extra MLP uncovered.

Pruning has little impact on the performance most benchmarks, as it mainly suppresses pathologies. It significantly hurts *soplex*, which has many miss-dependent (but non-recursive) loads which themselves are long latency misses. Here, suppressing overlapping periods of Runahead mode prevents it from exposing the MLP present in these second and third "layers" of loads.

With pruning, the difference between activating Runahead on L2 and L3 misses becomes more varied. Some programs (*e.g.*, *milc* and *gems*) see some performance benefits from applying Runahead to L2 misses, as expected. However, for other programs (*e.g.*, *soplex* and *lbm*), applying Runahead to L2 misses actually degrades performance. In these cases, the shorter L2 misses do not expose MLP on certain loads, which are then learned as “useless.” If these loads later miss the L3, pruning suppresses Runahead mode, missing opportunities where MLP may be exposed under the longer latency.

BOLT outperforms all four Runahead configurations across the board. BOLT’s performance advantage comes from a combination of two factors. First, BOLT can exploit ILP under long latency misses as well as MLP. Second, BOLT does not need to balance its performance gains against the cost of flushing the entire pipeline—a benefit which is relatively more important for the moderate latencies of L2 misses.

Overhead. Runahead’s overheads follow similar trends to its performance. Without pruning, Runahead sees more overhead when applied to L2 misses than when applied to L3 misses only (especially on *mcf* which does significant pointer chasing). With pruning, the differences vary, but typically L2+L3 Runahead sees lower overheads because more pruning occurs. With one exception, BOLT’s re-execution overheads fall into two categories relative to Runahead’s overheads. For *cactus*, *gems*, *milc*, and *zeusmp* BOLT’s re-execution overheads are uniformly lower than all of Runahead’s configurations. This is due to BOLT’s ability to re-execute *only* miss-dependent instructions while Runahead must re-execute both miss-independent and miss-dependent instructions.

The second category is those benchmarks where BOLT’s re-execution overhead is lower than un-pruned L2+L3 Runahead, but higher than pruned L2+L3 Runahead. This category is comprised of *lbm*, *soplex*, *sphinx*, and *mcf*. Here, BOLT enjoys its standard overhead advantages compared to un-pruned Runahead, but Runahead’s pruning rules out enough missing loads that its overhead drops below BOLT’s. This behavior typically goes along with Runahead’s largest performance losses from pruning, while BOLT maintains performance in these cases.

The exception is *libquantum*. Here, BOLT actually has *higher* re-execution overhead than all Runahead configurations. For the two L3-only configurations, Runahead does nothing because *libquantum* almost exclusively experiences L3 hits. For pruned L2+L3, Runahead reverts to conventional execution in many cases because it learns that MLP is frequently not present under these moderate latency L2 misses. The difference between the remaining configuration—un-pruned L2+L3 Runahead—which has 19% overhead and BOLT which has 24% overhead comes from the fact that *libquantum* has a significant number of L2 miss dependent branch mis-predictions. In this situation, BOLT begins re-executing miss-dependent instructions before discovering the mis-prediction and squashing to a checkpoint. Runahead simply squashes to the checkpoint immediately when the miss returns.

ED^2. Overall, pruned Runahead applied to L2 and L3 misses has the best (lowest) ED^2. However, this is because it has the lowest re-execution overheads—it also has the lowest performance. BOLT has better ED^2 than all of Runahead's configurations on all benchmarks except *libquantum* and *mcf*. BOLT's ED^2 advantages are typically quite significant, while its disadvantages on these two benchmarks are small. The net result is that BOLT improves ED^2 by 30% on the memory-bound subset of SPEC while the best Runahead configuration only improves ED^2 by 9%. On the two benchmarks where BOLT's ED^2 is higher than Runahead's, BOLT has higher performance, but this is outweighed by higher re-execution overhead.

5.4 BOLT vs. CFP

Figure 5.5 shows the behavior of CFP when applied to L3 misses only, or all L2 misses, and compares both configurations to BOLT. CFP does not use pruning—although BOLT's pruning mechanisms could be applied to it.

Performance. CFP's performance is typically higher when it applies load latency tolerance to all L2 misses rather than only to L3 misses. The exception is *mcf*, where more useless attempts to apply latency tolerance to pointer chasing result in more re-execution and resource deadlock squashes with no performance benefit.

Compared to CFP, BOLT always obtains higher performance. These differences arise

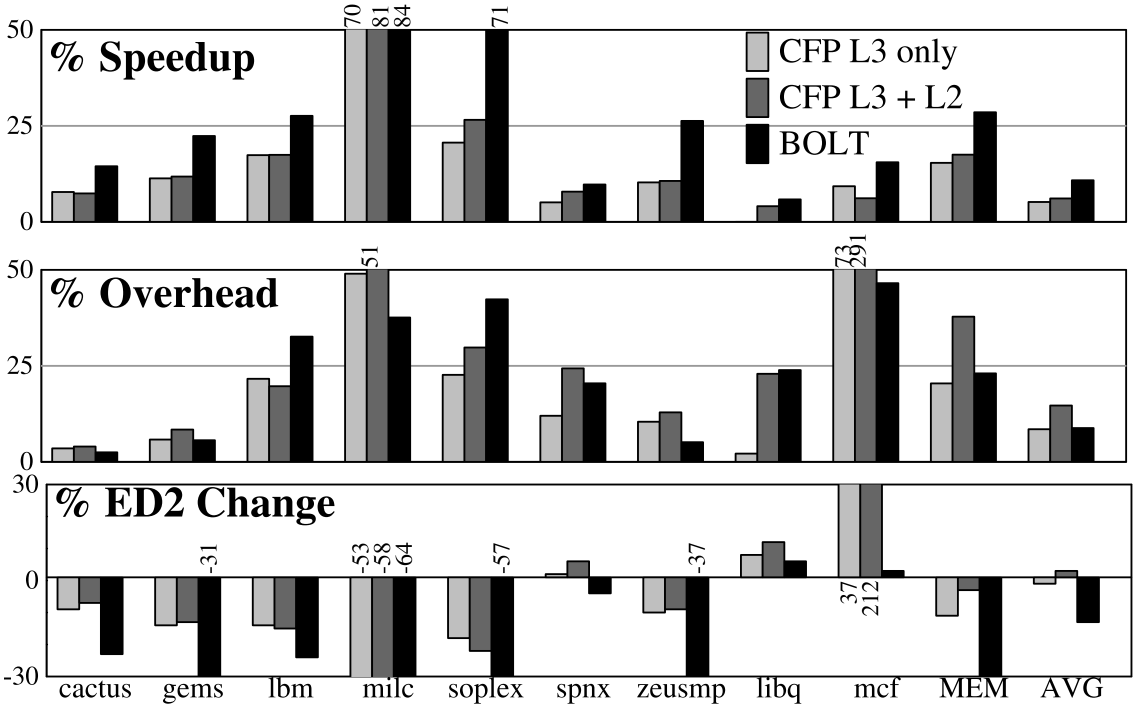


Figure 5.5: Applying CFP's load latency tolerance to L3 misses only or to L2 and L3 misses

from a combination of factors—BOLT's register substrate results in fewer resource deadlocks and better recovery, BOLT's load/store queues are more efficient, and BOLT's join pruning removes needless re-executions allowing the pipeline to execute other instructions. Experiments that separate the effects of these differences are shown below.

Overhead. CFP's re-execution overheads are typically higher for L2+L3 compared to L2 only CFP. Compared to CFP, BOLT's overheads are sometimes higher and sometimes lower. When BOLT has higher re-execution overhead, it goes along with higher performance—BOLT is able to perform more latency tolerance than CFP. BOLT may also have much lower overhead (*i.e.*, *milc*) due to fewer resources deadlock-induced squashes. Both behaviors are typically related to the register substrate and are discussed in more detail in Section 5.4.1.

ED^2. CFP obtains the best overall ED^2 when applied only to L3 misses, as the overheads incurred by applying it to L2 misses outweigh the small performance benefits. BOLT—which applies latency tolerance to L2 and L3 misses—always obtains better

ED^2 than both of CFP's configurations. BOLT's higher performance is typically a large factor in its lower ED^2, but is not always the most significant aspect. In *mcf* and *sphinx*, BOLT's lower re-execution overhead is the dominant factor. For *libquantum*, where BOLT and L2+L3 CFP both have very similar performance *and* re-execution overheads, BOLT's lower structural overheads give it the ED^2 advantage. Overall on the memory-bound subset of SPEC, CFP's best configuration improves ED^2 by 11%—slightly better than Runahead's 9%, but significantly less than BOLT's 30%.

BOLT differs from CFP in three areas: register substrate, load and store queue design, and pruning. We isolate the effects of these three differences starting with the register substrate.

5.4.1 Effect Isolation: Register Substrate

Figure 5.6 isolates the effects of BOLT and CFP's register substrate. Here, both BOLT and CFP use fully associative load and store queues, BOLT's pruning mechanisms, and are applied to both L2 and L3 misses. Fully associative load and store queues are used for this experiment rather than BOLT's because BOLT's memory system cannot be applied to CFP exactly as is. While the general techniques—SQIP, SVW, DSC/SDR, and CSB—could all be retrofitted onto CFP, CFP does not have the same notion of speculative retirement as BOLT. This difference makes it simpler to isolate the register effects by using fully associative load and store queues on both designs.

The top graph in Figure 5.6 shows CFP's and BOLT's speedups, while the bottom graph shows their re-execution overhead. BOLT typically outperforms CFP. The 40% performance difference on *soplex* primarily arises from the fact that CFP checkpoints at rename and must use checkpoints for all forms of recovery, while BOLT checkpoints at speculative retirement and only uses checkpoints for mis-speculations in the deferred region. CFP holds more registers live in checkpoints, making fewer available for renaming and re-renaming. On *soplex*, CFP spends 22% of dispatch cycles stalled because no physical registers are free, compared to only 7% in BOLT.

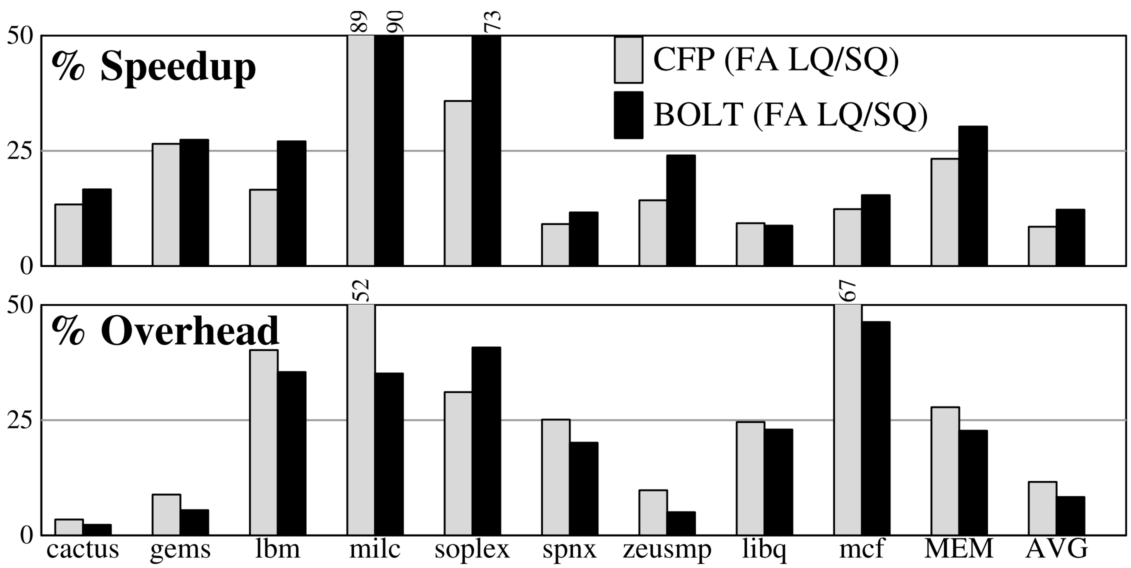


Figure 5.6: Comparison of CFP and BOLT register substrate

CFP outperforms BOLT on only one benchmark—*libquantum*—by 0.5%. Here, there are enough L2 miss-dependent branch mis-predictions to make CFP's aggressive use of eight checkpoints—instead of two in BOLT—give it a tiny advantage. Although BOLT stalls the speculative retirement of poison branches in an attempt to reduce checkpoint overhead, it only stalls speculative retirement as long as physical registers and ROB space are available. For *libquantum*, this policy allows many branches poisoned by L2 misses to speculatively retire.

The bottom graph in Figure 5.6 shows that BOLT typically has lower re-execution overhead than CFP as well. The exception here is *soplex*, where BOLT stalls less and applies latency tolerance more frequently. However, BOLT's 40% performance advantage justifies the 10% re-execution overhead difference.

CFP's physical-to-physical re-renaming and checkpoint-only recovery both contribute to its higher re-execution overhead. Physical-to-physical re-renaming is prone to register deadlocks—all registers are held live in checkpoints or the physical-to-physical map table, making forward progress impossible without a squash. This problem is exacerbated by the fact that squashing is done at a checkpoint granularity. The re-execution overheads for *milc* demonstrate this combination most clearly. With CFP's register management, *milc* must squash 316 μops for physical register deadlocks per 1000 μops committed. Of these, 316 μops, 22 are younger than the point of the squash, and 294 are older—meaning they were squashed to reach the immediately older checkpoint. By contrast, BOLT squashes 4 μops per 1000 committed μops for register deadlocks on *milc*. Not only is BOLT's logical-to-physical re-renaming more register efficient, but BOLT's speculative retirement also makes new physical registers come available regularly. For BOLT to experience a register deadlock during slice in, speculative retirement must stall under conditions that cannot be resolved until slice processing completes. This occurs only when the instruction at the head of the ROB (1) is a store poisoned on its address input, (2) produces an exception, (3) is poisoned and slice buffer space is exhausted, or (4) depends on an instruction which has already been re-renamed but not yet re-dispatched.

On *mcf*, CFP's higher re-execution overheads come primarily from checkpoint-only recovery. Here, branch mis-predictions, store-load ordering violations, and resources deadlocks all come into play for both BOLT and CFP, but CFP must squash more instructions to reach a checkpoint, while BOLT can typically recover in the tail region.

Relative to CFP, BOLT's register substrate also improves performance and lowers re-execution overhead for branch mis-prediction bound programs. While these are not pictured in Figure 5.6, the impact of the register substrate on these programs can be seen in Figure 5.10, as the pruning algorithms and different load/store queues are not significant.

5.4.2 Effect Isolation: Load and Store Queue Designs

Figure 5.7 shows the performance of three different load/store queue designs. Each load/store queue design is coupled with BOLT's register substrate and pruning mechanisms. Load latency tolerance is applied to L2 misses as well as L3 misses for all three designs.

The first bar corresponds to idealized fully associative load and store queues. While such a design is not practical, it serves roughly as an upper bound on performance against which to compare the other two designs.

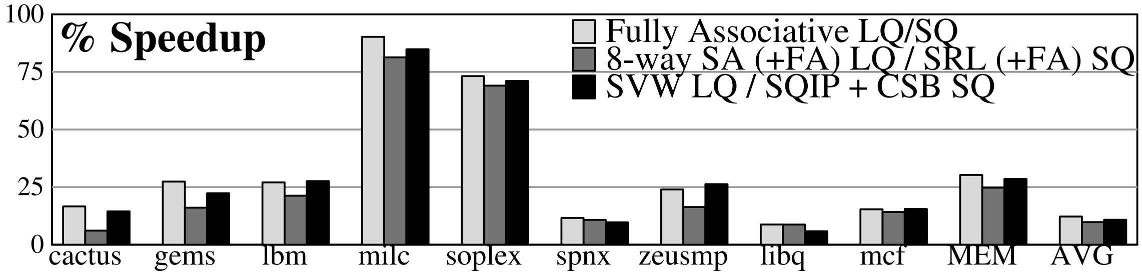


Figure 5.7: BOLT with three different load and store queue designs.

The middle bar corresponds to the hierarchical load/store queue used by CFP—a conventional fully associative 48-entry load queue backed by a 256-entry 8-way set-associative load queue, and conventional 32-entry fully-associative store queue backed by a 128-entry SRL. In this design set-conflicts in the load queue hinder performance. *cactus* is the best example of this behavior, where this design under-performs the other two by roughly 10%. Note, BOLT mitigates this disadvantage somewhat relative to CFP—when a squash is required to resolve a set-conflict, BOLT can typically squash many fewer instructions than CFP.

The rightmost bar is the SVW load queue and SQIP + CSB store queue used by BOLT. This design typically comes very close to the idealized fully-associative queues in terms of performance. In the case of *zeusmp*, BOLT's design slightly outperforms the fully-associative design. Here, the chained store buffer's ability to avoid stalls due to partial forwardings gives it a slight advantage over the fully-associative store queue which does not avoid such stalls. On benchmarks where the fully-associative queues outperform BOLT's design, the differences arise from two factors. In some cases, SQIP suffers from more store-to-load violations. This is the case on *milc*, where SQIP suffers 20 load squashes per 100,000 μops, while the associative store queue suffers 1 load squash per 100,000 μops. The other factor is that the associative load queue can detect a load mis-speculation more quickly than SVW can. With an associative load queue, mis-speculations are detected when the store which causes the violation executes. With SVW, the mis-speculation is detected when the load which causes the violation speculatively retires. This is the case in *gems*, where SQIP/SVW actually has slightly *fewer* squash events than the associative

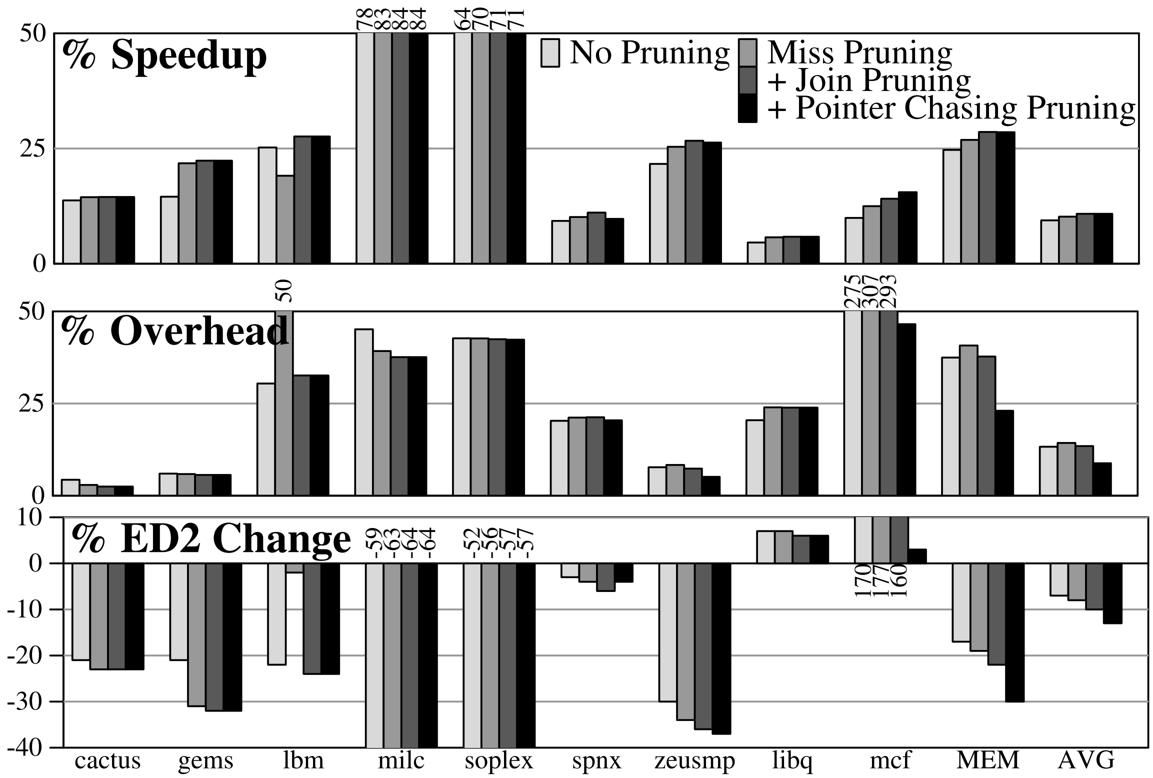


Figure 5.8: Impact of BOLT's pruning algorithms

queues, but squashes more instructions total.

5.4.3 Effect Isolation: Pruning Mechanisms

Figure 5.8 isolates the impact of BOLT's pruning mechanisms on performance (top), re-execution overhead (middle) and ED^2 (bottom). The left-most bar shows the behavior when no pruning mechanisms are used.

Miss pruning. The second bar shows "miss" pruning. This form of pruning, which was proposed in WIB [44], only eliminates the re-injection of those instructions which do not depend on *any* returning misses. While this technique generally helps performance and overhead, surprisingly, it hurts both on *lbm*. This effect comes from the prevalence of multiple slices joining together. With only miss pruning (*i.e.*, without join pruning) a load whose miss is still-outstanding will not re-execute, but, its dependents *will* re-execute if they depend on other misses which have returned. This has the effect that a slice buffer pass will not re-inject still-missing loads (because miss pruning is employed), but will re-inject the other instructions because they depend on one returned miss (because join pruning is not employed). If some of the passed-over loads return in the middle of the slice buffer pass, their dependents will re-execute and slice back out, but re-injection of the loads must wait until the slice buffer pass completes and a new pass can begin. Without miss-pruning, the loads are re-injected as well—providing the opportunity for those whose misses return before they re-execute to properly execute immediately.

Join re-execution pruning. The third bar adds join re-execution pruning—elimination of the re-injection of those instructions for which no input will become available during the current slice buffer pass. Join re-execution pruning has two beneficial effects—it both lowers re-execution overhead and increases performance. Performance improves because fewer useless instructions re-execute making more pipeline bandwidth available for useful instructions. While join re-execution pruning does not fix pathological energy overheads in the benchmarks shown here, results on SPEC2000 benchmarks show that without join pruning, *art*'s ED^2 becomes 40% higher (worse) than a baseline ROB processor but remains unchanged with join pruning [35].

Pointer-chasing deferral pruning. The fourth bar adds pointer chasing deferral pruning—detection of pointer chasing loads and reversion to conventional execution for those loads. Here performance is largely unchanged—*sphinx* and *zeusmp* see very small decreases, and *mcf* sees a slight increase, but overall performance remains the same. The lack of change here is important as it means that the pointer chasing detection is not accidentally suppressing latency tolerance on benign instances of indirection. The important effect of pointer chasing pruning is the reduction of re-execution overheads—and correspondingly, ED^2—on *mcf*. Here, re-execution overhead drops from 293% to 47%, changing the ED^2 lose from 160% to 3%. This decrease in energy consumption is important, because it brings *mcf*—and by extension, pointer chasing programs in general—from a pathological blow up in energy consumption from use of load latency

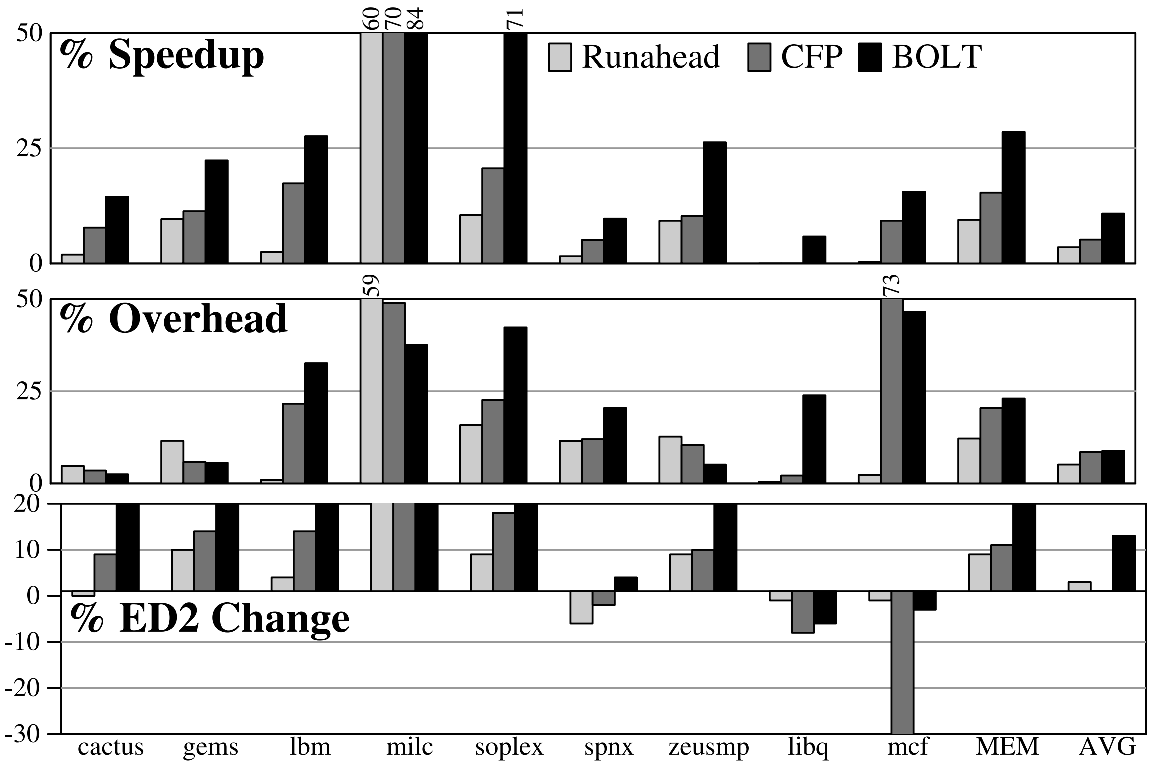


Figure 5.9: Runahead, CFP, and BOLT

tolerance to an acceptable increase. In SPEC2000, *ammp* and *parser* also see significant reductions in re-execution overheads (and improvements in ED^2) from pruning pointer chasing [35].

5.5 Summary Comparison

The previous sections provided detailed comparisons of Runahead to BOLT and CFP to BOLT. Figure 5.9 shows the best configuration for each design—pruned L2+L3 Runahead, L3-only CFP, and L2+L3 BOLT—all together.

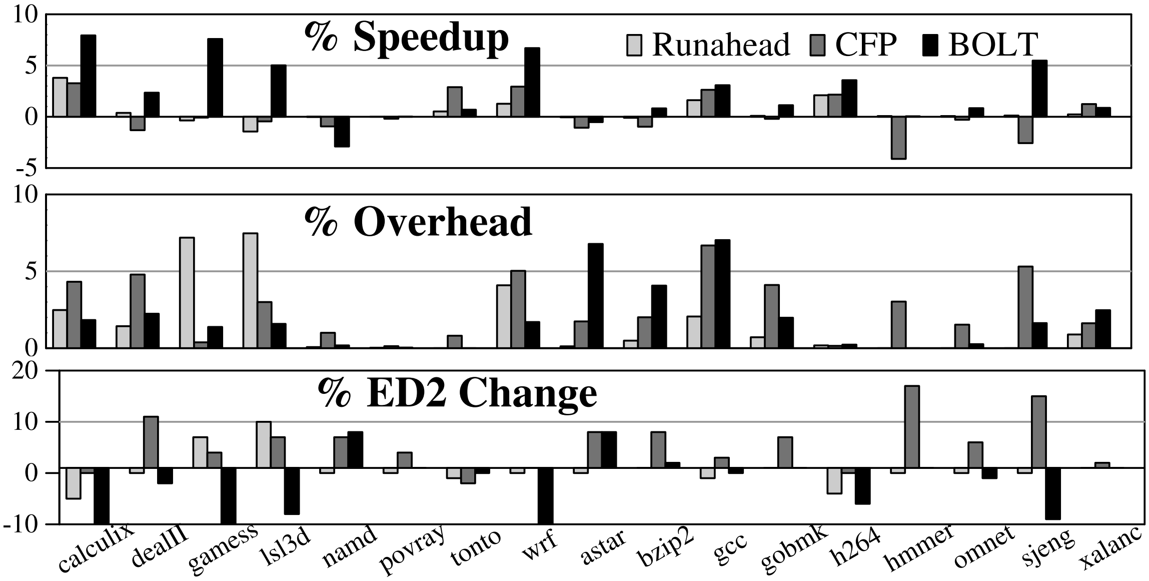


Figure 5.10: Performance on Non-memory-Bound Benchmarks

5.5.1 Behavior on Non-Memory Bound Benchmarks

While this evaluation primarily focuses on memory-bound benchmarks, the behavior of Runahead, CFP, and BOLT on the non-memory-bound benchmarks is also important. Figure 5.10 shows the performance, overhead, and ED^2 of Runahead, CFP, and BOLT on the remaining benchmarks in SPEC 2006. Note that only individual benchmarks are shown—the average for all of SPEC appears in Figure 5.9 and is not repeated here.

While these benchmarks do not have as many long latency misses as the memory-bound ones, there are still opportunities for performance gains. One benchmark of particular interest is *gamess*, on which BOLT obtains an 8% speedup, while CFP and Runahead do not. *gamess* primarily experiences L2 misses which hit the L3. CFP does nothing because it only applies latency tolerance to L3 misses. These misses are also far enough apart that they do not overlap in Runahead execution—Runahead experiences a slight slowdown while it learns to suppress Runahead. BOLT obtains a speedup in this case by exploiting ILP under the moderate latency L2 misses.

More important than the moderate speedups on some benchmarks is the avoidance of slowdowns. CFP suffers slight slowdowns on about half of these benchmarks, including 4% on *hmmer*. By contrast, BOLT suffers slowdowns on only two benchmarks—3% on *namd* and 0.5% on *astar*. For these two benchmarks—on which CFP also sees slowdowns—the cause is cache pollution from the application of latency tolerance down the wrong path. Here, as the program executes down the wrong path due to a branch mis-prediction, cache misses for un-needed addresses occur. While conventional execution would result in these misses and their dependents clogging the issue queue, BOLT and CFP apply latency tolerance and expose more wrong-path MLP. This MLP hurts performance by replacing useful data with useless data in the caches. It may be possible in either CFP or BOLT to detect the wrong path execution using previously proposed techniques [5] and shut down latency tolerance to prevent this problem. Runahead is not affected because it only begins latency tolerance when the long latency miss reaches the head of the ROB—which will not happen because the older incorrectly predicted branch must resolve first.

CFP's remaining slowdowns are due to its checkpoint overhead. When mis-speculations occur between checkpoints, older correct-path instructions must be squashed, decreasing performance and increasing re-execution overhead. This particularly hurts *hmmer* and *sjeng*, where the combination results in ED^2 increases of 17% and 15% respectively.

CFP outperforms BOLT (and Runahead) on one benchmark—*tonto*. Here, difficult forwarding patterns pose problems for BOLT's speculatively indexed store queue. CFP uses a combination of associative and indexed store queues, and is able to forward more accurately in these cases. Runahead never outperforms BOLT.

Overall, BOLT typically improves ED^2 relative to the baseline. When it hurts ED^2, it does so in a small way. The only two benchmarks on which BOLT increases ED^2 by more than 2% are *namd* and *astar*—the two on which it obtains slowdowns due to wrong-path MLP. Runahead also has only two benchmarks with ED^2 increases of more than 2%—*gamess* and *leslie3d*. For these, Runahead incurs overheads and performance losses while

	Runahead	WIB	D-KIP	CFP	BOLT
Register management					
Checkpoints	1	0	0	8	2
Register files	1	1	9	1	1
Re-renaming	None	None	None	Physical	Logical
Load and store management					
Store forwarding	Assoc Store Queue + Fwd$	Assoc Store Queue	Assoc Store Queue	Assoc + Indexed Store Queues	Indexed Store Queue
Load verification	Assoc Load Queue	Assoc Load Queue	Assoc Load Queue	Assoc + Set Assoc Load Queues	Indexed Load Queue
Re-execution					
Insns buffered	None	All	Miss-dep	Miss-dep	Miss-dep
Insns re-executed	All	Miss-dep	Miss-dep	Miss-dep	Miss-dep
Non-blocking	N/A	Yes	No	Yes	Yes
Start at load	Oldest	Any	Oldest	Any	Any
Visible to tail	N/A	Immediately	On Squash	Immediately	Immediately
Pruning	Useless/ Overlapping	Miss	None	None	Miss/Join/ Pointer
Overall Effects					
Performance	Low	High	Moderate	Moderate	High
Static cost	Low	High	High	High	Moderate
Dynamic cost	High	Moderate	High	Moderate	Low

Table 5.4: Comparison and contrast of out-of-order load latency tolerant designs.

it learns that there is no MLP. By contrast, CFP has twelve benchmarks with ED^2 increases greater than 2%, including three over 10%.

5.5.2 Qualitative Comparison of All Five Designs

Table 5.4 summarizes the differences between all five out-of-order load latency tolerant designs. The comparison of overall effects at the bottom—performance, static area and energy overhead, and dynamic energy overhead—come from the data shown above for Runahead, CFP, and BOLT. WIB and D-KIP are not simulated, but their entries can be inferred from their described behaviors and individual evaluations.

WIB. WIB's performance should be comparable to BOLT's with only miss-pruning. The problem with WIB is not performance, but rather static area and energy costs. WIB buffers all instructions and more importantly does not scale the register file. WIB does not itself scale the load/store queues, but could be coupled with SQIP/SVW.

D-KIP. D-KIP has problems in both performance and cost. For performance, D-KIP re-executes instructions on an in-order pipeline, which makes its re-execution blocking—if it encounters a long latency miss during re-execution, re-execution must stall for that miss to return. D-KIP does not improve performance in the presences of dependent (non-pointer chasing) misses. For static cost, D-KIP requires an entire in-order processor as well as a large number of register checkpoints which must support incremental updates—essentially, D-KIP maintains nine register files, one physical and eight logical. For dynamic overhead, D-KIP does not propagate slice register values from the in-order re-execution processor to the out-of-order tail execution processor except in the case of a tail squash. This forces D-KIP to defer and re-execute instruction un-necessarily. D-KIP must also make multiple copies of every register value.

5.6 Why Not a Transactional Data Cache?

The mechanisms to handle memory operations in the deferred region— CSB, SVW-MT, and DSC/SDR—are an important novel aspect of BOLT. This section justified the need for this new design by measuring the performance and overhead of an alternative design based on a transactional data cache. A transactional data cache is a popular mechanism to speculatively buffer a large number of writes [4,8,9,13,14,51,64]. Therefore, it seems like a natural choice for load latency tolerance. However, the situations in which transactional data caches are used do not have the possibility of younger stores being written to the cache before the re-execution of older miss-dependent loads or stores. This section explores a hypothetical BOLT design (BOLT+TD$) which uses a transactional data cache *instead of* CSB, SVW-MT, and DSC/SDR for speculatively retired memory operations.

5.6.1 Design

In BOLT+TD$, a speculatively retiring store speculatively writes the data cache. If the block is non-speculatively dirty, then it is first written back to the L2. If the block is speculatively dirty with writes from an older checkpoint, speculative retirement stalls. Loads also access the transactional data cache as they speculatively retire and set the corresponding speculatively read bit. Note that this access occurs at a stage when loads do not typically access the cache, so either the cache tags must be provisioned for extra bandwidth, or the design must accept any performance loss from contention.

Re-execution of deferred loads. BOLT+TD$ must avoid write-after-read hazards posed by re-executing deferred loads in the presence of younger stores which have speculatively completed. To accomplish this, BOLT+TD$ tracks age information (*i.e.*, an SSN) in the data cache. When a deferred load re-executes, it compares the SSN of the immediately older store (recorded at dispatch) to the SSN in the cache. If the SSN in the cache is greater than the load's SSN, a write-after-read violation is detected. BOLT+TD$ squashes, aborting all speculative writes, and re-executes from the immediately older checkpoint. Note it is not possible to simply obtain the value form the L2 in this case because there may be an older speculative store to the same address whose value was overwritten by the younger store.

In the data cache, age can be tracked as one SSN per cache block, representing the youngest store to (speculatively) write that block. This representation adds only 16 bits of overhead per cache block, but provides (conservatively) imprecise information. More precise information—one SSN per word or even per byte—could be employed to avoid false violations, but incurs higher overheads.

Re-execution of deferred stores. When a deferred store re-executes, it must check its own age against the age recorded in the data cache. If the data cache indicates that a younger store has written the block, the re-executing store must ensure that it does not accidently overwrite the younger value. If the cache tracks age precisely (*i.e.*, one SSN per byte), then the re-executing store can determine with certainty which (if any) of the addresses it writes have been overwritten. However, if the information is imprecise, and *any* younger write is present then it is impossible to tell if the younger store overwrote some, all, or none of the region written by the re-executing store. In this case, BOLT+TD$ must squash to avoid write-after-write hazards.

Poison propagation. BOLT+TD$'s store-load forwarding mechanism must be able to propagate not only values, but also poison. In BOLT+TD$, this means that the data cache also stores poison information. As with age, poison may be tracked with one bit per cache block or at a finer granularity. The former poses the potential for unnecessary deferrals and re-executions, while the latter increases storage overhead.

Tail region load and store queues. BOLT's load and store queues are designed around the synergy between SQIP, SVW, CSB, DSC/SDR, and SVW-MT. BOLT+TD$ changes this synergy in three ways. BOLT+TD$ discards CSB, performing store-to-load forwarding through the data cache instead. It also discards SVW-MT for multi-threaded load verification—it can use the speculative read bits in the data cache. Finally, BOLT+TD$ does not need DSC/SDR's ability to non-speculatively complete stores between checkpoints as the transactional data cache may complete them speculatively.

With three pieces of the five-part synergy removed, a BOLT+TD$ design will likely use conventional, small associative load and store queues for the tail region. Without CSB, an SVW-based design must rely on a set-associative SSBF to maintain a low re-execution rate. At small load queue sizes, the energy overheads of such an SSBF remove much of the appeal of SVW. SQIP requires SVW.

5.6.2 Evaluation

Relative to BOLT, the potential drawbacks of BOLT+TD$ are squashes due to WAR and WAW hazards. The potential benefits are fewer capacity stalls in the load and store queues, and potentially better use of data cache bandwidth. This section isolates the first effect—and in the process counts the number of WAR and WAW scenarios in BOLT—by simulating an idealized version of BOLT+TD$. BOLT+TD$ is modeled by simulating BOLT with

Age	Poison	Latency (cycles)	Area (mm^2)	Read Power (pJ/read)	Write Power (pJ/write)
Per-Block	Per-Block	2	0.078	0.350	0.058
	Per-Word	2	0.078	0.379	0.077
	Per-Byte	2	0.112	1.098	0.147
Per-Word	Per-Block	2	0.336	1.158	0.091
	Per-Word	2	0.336	1.158	0.091
	Per-Byte	2	0.616	1.586	0.146
Per-Byte	Per-Block	2	1.438	14.605	1.084
	Per-Word	2	1.439	14.605	1.084
	Per-Byte	2	1.512	15.849	1.129

Table 5.5: Area and energy costs for age and poison tracking in BOLT+TD$

large (*i.e.*, 512 entries each) fully associative load and store queues, with added checking for RAW and WAR hazards. BOLT+TD$ does not model an actual transactional cache—aborts do not invalidate blocks so there is no performance penalty to re-fetch them after a squash, blocks are not cleaned to the L2 so there is no bandwidth contention or latency penalty, and speculative retirement does not stall on stores which write blocks already written by an older checkpoint.

There are nine implementations of BOLT+TD$—the cross-product of per-byte, per-word, or per-block poison information with per-byte, per-word, or per-block age information. The area and energy costs of these choice are shown in Table 5.5. As with the extra data used by SVW-MT, these overheads are modeled as a separate direct mapped structure accessed in series with the data cache tags. In this case, it is probably more efficient to create a separate structure, as this information only needs to be manipulated by speculatively retiring stores and by re-executing deferred loads and stores. These models assume one read port and one write port, but the performance model does not account for the port contention that may occur on this structure (*i.e.*, if a deferred load and store re-execute in the same cycle and both need to read the table).

Of these nine configurations, the three with per-byte poison information are simulated and compared to BOLT. Simulations show that per-byte and per-word age information are

Structure	BOLT+TD$ Per-Word	BOLT+TD$ Per-Block	BOLT
Unchanged Overheads	0.366	0.366	0.366
Load Queue	0.018	0.018	0.069
Store Queue	0.069	0.069	0.049
Age + Poison	0.336	0.078	—
Total	0.789	0.531	0.484
% of Core i7	2.5%	1.7%	1.5%

Table 5.6: Area Overheads (mm^2)

almost identical—the only noticeable difference is 1.5% on *sphinx*. Because of this similarity, we give BOLT+TD$ the benefit of the per-word poison information area overheads, assuming that it will also not affect performance, and show the results for the per-word and per-block age information BOLT+TD$ configurations.

Table 5.6 shows the marginal area overheads for these two BOLT+TD$ configurations, and for BOLT. The overheads which do not change (*i.e.*, checkpoints, reference counts, rob, slice buffer, and sequence number table) are compressed into their total on the first line. The two BOLT+TD$ configurations use conventionally sized fully associative queues for their tail regions. Being associative, these queues are slightly larger than the SVW/SQIP baseline queues[3]. The area for the age/poison table is included for BOLT+TD$, but the transactional read and write bits are treated as "free."

Figure 5.11 compares the performance, re-execution overheads, and ED^2 of the two BOLT+TD$ configurations to BOLT. BOLT+TD$'s inability to forward from any store other than the youngest lowers its performance, and increases its re-execution overhead and ED^2.

Performance. In general, BOLT outperforms both BOLT+TD$ configurations, with the BOLT+TD$ per-word configuration out-performing the per-block configuration. There are two exceptions. On *sphinx*, BOLT+TD$'s per-byte (not pictured) configuration slightly (*i.e.*, 1%) outperforms BOLT. *sphinx* has very few squashes, and is exploiting the

[3] Note that the area for CSB's root table (which is also used by SVW) is included in the baseline store queue.

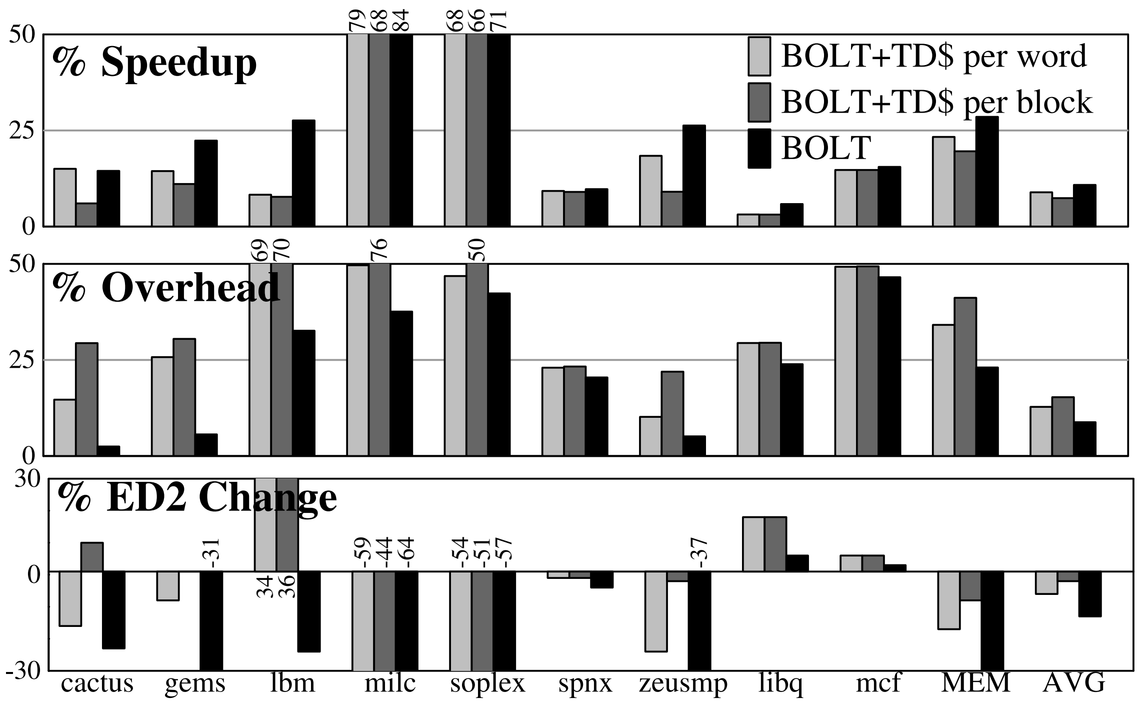


Figure 5.11: BOLT+TD$ vs. BOLT

fully associative load and store queues (*i.e.*, the behavior is the same as in Figure 5.7). Second, on *cactus*, BOLT+TD$'s per-word configuration outperforms BOLT by 0.5%. Here, the difference is store buffer capacity—the transactional data cache can hold more speculative store than the CSB. As shown later in Figure 5.13, *cactus* benefits significantly (9%) from a larger store buffer. The benefits here are much smaller because they are offset by a large number of squashes.

Re-execution Overhead. BOLT+TD$ must squash when it encounters a WAR or WAW hazard due to its use of a transactional data cache. These squashes increase re-execution overheads relative to BOLT, in some cases quite significantly (*e.g.*, *cactus*, *gems*, *lbm*, *milc*, and *zeusmp*).

ED^2. BOLT has the best ED^2 in all cases except *sphinx* in the per-byte configuration which is not shown (where BOLT+TD$ is basically BOLT with fully associative queues). Overall, BOLT+TD$ obtains only 17% (per-word) and 8% (per-block) ED^2 improvements, compared to BOLT's 30%.

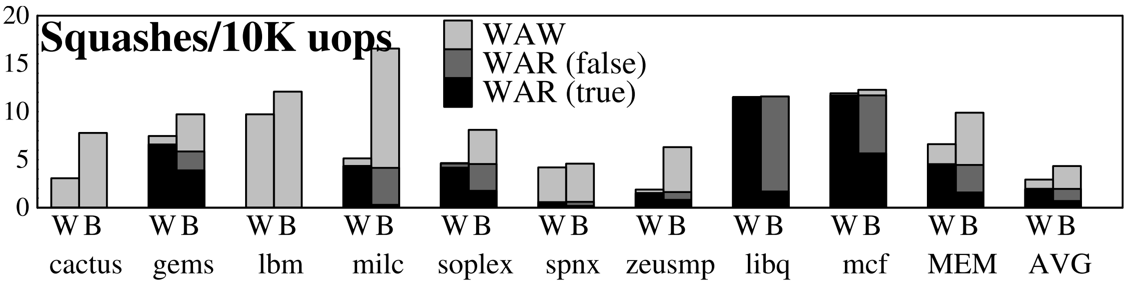


Figure 5.12: BOLT+TD$ squashes per 10,000 μops

Squashes. Figure 5.12 breaks down the squashes in each BOLT+TD$ configuration. The left bar in each pair ("W" underneath each bar) shows a breakdown of the transactional data cache related squash es for the per-word configuration of BOLT+TD$. The right bar in each pair ("B" underneath each bar) shows this breakdown for the per-block configuration. Each bar shows squash events per per 10,000 μops and is broken down by the cause of the squash—true WAR hazards, false WAR hazards, and WAW hazards. The true WAR hazards are smaller for per-block than for per-word for two reasons. First, a false WAR hazard or a WAW hazard may squash a checkpoint before the true WAR hazard manifests. Second, the distinction between the two types of WAR hazards is done by the youngest store to write the block. Specifically, if a true WAR hazard (actual address match) and a false WAR hazard (same block, but disjoint addresses) are both present for a given load, the squash is attributed to the category matching the younger store. This may cause a false hazard to "shadow" a true hazards. On average over the memory-bound subset of SPEC, the per-word configuration experiences six squashes per 10,000 μops while the per-block configuration experiences ten.

Combination of a transactional cache and CSB. The previous experiments showed that BOLT cannot profitably exploit a transactional cache in isolation. However, it may be able to exploit *both* a transactional cache *and* a chained store buffer. In BOLT+TD$+CSB, the CSB contains only miss-dependent stores and stores that write the same addresses as older poisoned loads or stores—to resolve WAR and WAW hazards respectively. The transactional data cache accepts all miss-independent stores. The transactional data cache also does not track age or poison as that information is tracked by the CSB for the stores where it is relevant. To do this, BOLT+TD$+CSB tracks which addresses are read or written by miss-dependent loads and stores. When a store speculatively retires, BOLT+TD$+CSB checks if it is to an address needed for re-execution. If so, BOLT+TD$+CSB inserts the store into the CSB and speculatively completes it. Having the relevant stores in the CSB allows non-speculative re-execution of deferred loads to proceed even when the cache has younger writes. Loads search the CSB to find the proper value, reading the clean value from the L2 if needed—note that in this design it *is* possible to non-speculatively read the L2 because the CSB contains all relevant store ages. Stores search the CSB to find what, if any, part of their address range has been overwritten. A squash to avoid a WAR hazard may still be needed if a load's address was unknown at speculative retirement (*i.e.*, its address input was miss-dependent)—BOLT+TD$+CSB cannot know which younger stores to track in this case. When squashes are limited to WAR hazards on loads which do not know their address at speculative retirement, the squash rate drops significantly—averaging 0.1 squashes per 10,000 μops across the memory-bound subset of SPEC.

5.7 Sensitivity Analysis

This section investigates BOLT's sensitivity to changes in the size of BOLT's key window structures. While changing structure sizes changes energy consumption as well as performance, only performance results are shown here. For the variations of some structures (*i.e.*, doubling the load/store queue or slice buffer size), the area change is quite small compared to a Core i7 so it does not make a significant impact in the area-based energy approximations. Other variations (*i.e.*, extremely large window structures) pose large enough changes that area-based energy approximation may break down.

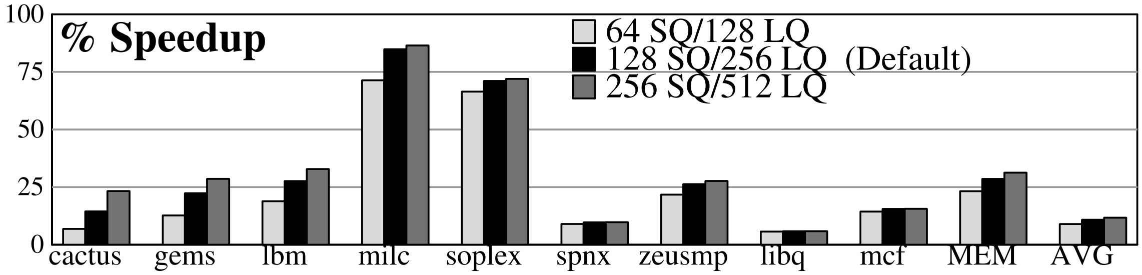


Figure 5.13: Sensitivity to load and store queue sizes. All BOLT configurations are normalized to the 32/48 ROB baseline.

5.7.1 Load and Store Queue Capacity

BOLT's large window requires larger load and store queues than a conventional processor. Figure 5.13 shows the performance of BOLT with three different size combinations of load and store queues. BOLT's default configuration—128-entry store queue and 256-entry load queue—is shown in the middle/darkest bar.

The left/lightest bar shows a 64-entry store queue and a 128-entry store queue—still larger than the baseline 32/48 sizes, but smaller than BOLT's default. This configuration can significantly under-perform the default, most notably on *milc* (14% difference), *gems* (10%), *lbm* (9%), and *cactus* (8%). The load queue is the primary factor for *gems* and *milc*, with load queue capacity stall cycles accounting for 18% and 11% of all dispatch cycles respectively. The store queue is the primary factor for *lbm* with 15% of dispatch bandwidth wasted due to store queue capacity stalls. Both queues are a factor in *cactus*, with 14% of dispatch bandwidth lost due to the store queue being full, and 13% lost due to the load queue being full. One benchmark not shown here, *sjeng*—which falls just barely outside the criteria for "memory-bound"—has significant store queue full cycles, accounting for 12% of its dispatch cycles.

The default 128-entry store queue and 256-entry load queue (middle/darkest bar) eliminates most load/store queue capacity stalls. Here, only a few benchmarks see significant (*i.e.*, more than 2% of dispatch bandwidth) load queue (9% on both *gems* and *cactus*) or store queue (11% on both *sjeng* and *cactus*, and 9% on *lbm*) capacity stall cycles.

The rightmost bar shows a 256-entry store queue and a 512-entry load queue. This

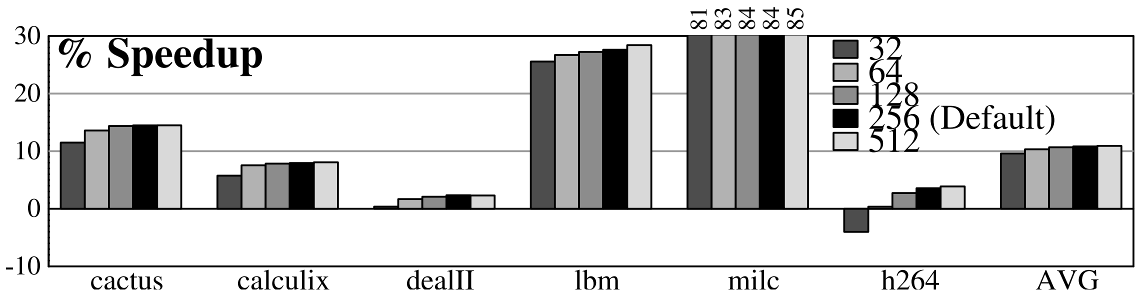


Figure 5.14: Sensitivity to CSB's root table size.

configuration shows noticeable performance gains in only a few benchmarks. The most significant gain comes in *cactus*, where performance increases by 9%. Even in this configuration, *sjeng* and *lbm* spend 10% and 5% respectively of their dispatch bandwidth stalling due to a full store queue. The fact that *sjeng*'s stalls decrease less than expected suggests that code regions which are comprised of a high percentage of stores are limited by store completion bandwidth.

Although not explicitly shown, the performance loss from the smaller load and store queues outweighs the energy savings of the smaller structures, resulting in slightly higher (worse) ED^2—2% overall and 6% across the memory-bound benchmarks. The slightly larger load/store queue combination has slightly lower (better) ED^2 on the benchmarks that benefit from it—having an average 2% improvement for the memory-bound subset of SPEC over the 128/256 combination. However, programs which do not benefit from the larger queues still consume more energy because of it. This difference is small—resulting in a 1–2% increase in ED^2 on most other programs. The overall effect is that the average ED^2 is the same for the larger load/store queues and for the 128/256 default.

5.7.2 CSB Root Table Capacity

Figure 5.14 evaluates BOLT's sensitivity to the size of the chained store buffer's root table. This graph shows a different subset of benchmarks from the other graphs—the root table is used in SVW, and can impact performance independently of load latency tolerance. Here, the benchmarks selected are those with the largest sensitivity to the size of the root

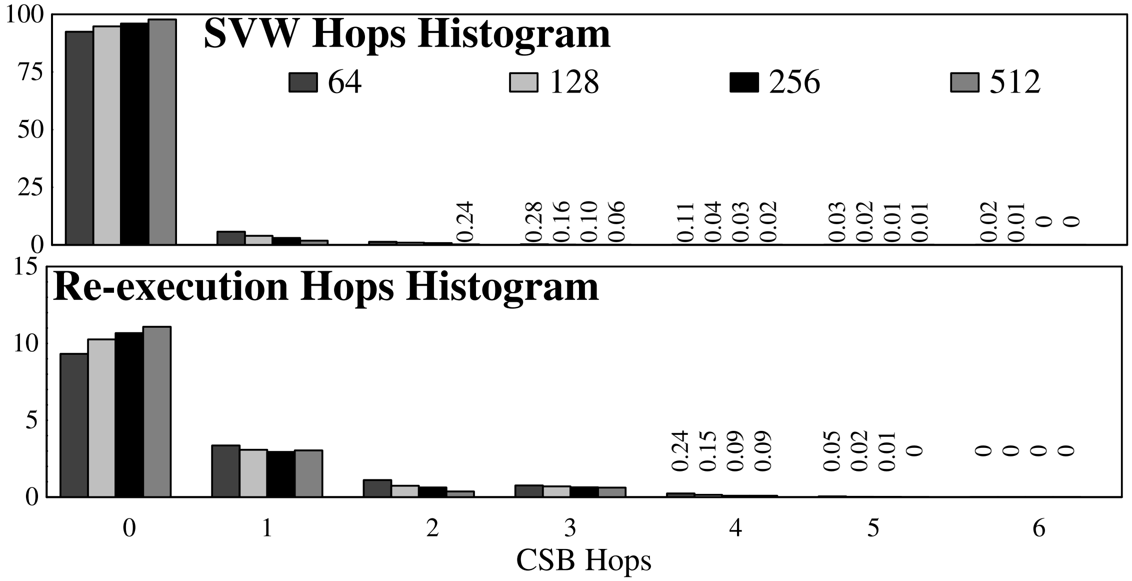


Figure 5.15: Histograms of the number of CSB hops required for loads with different sized root tables. Top: average over SPEC2006 for CSB hops done by SVW. Bottom: average over memory bound benchmarks for CSB hops done by re-executing loads (as the percent of all loads).

table—at least a 1.5% change in performance at the smallest size table shown.

Between 128 and 512 entries, BOLT's performance is relatively insensitive to the root table size. Average extra chain hops increase from three per 100 loads at 512 entries to five (entries) and seven per 100 loads (128 entries). At 64 entries, the average chain hops increases to 11 per 100 loads, and *h264ref* suffers a 3% performance loss. While *h264ref* does not have the largest number of extra hops, they have the largest performance impact because *h264ref* has a relatively high IPC, and the additional latency of SVW becomes critical. With only a 32-entry root table, the average extra hops increases to 17 per 100 loads, and performance begins to degrade more significantly.

To further examine the behavior of CSB with different sized root tables, the top graph in Figure 5.15 shows a histogram of the number of CSB hops required for SVW to verify a load. Each group of bars represents a number of CSB hops, ranging from zero to six, with one bar per root table size. Bars with un-readably small values have their value printed above them. For all four root table sizes, no hops (*i.e.*, verification by simply reading the SSBF) dominates—ranging from 92% in the smallest root table to 98% in the largest.

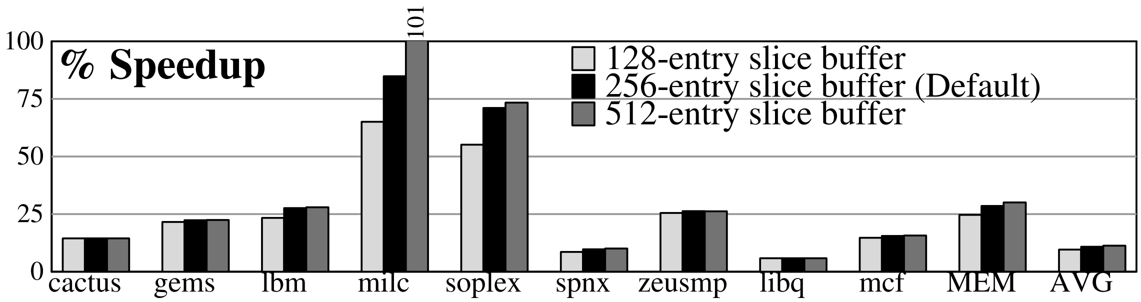


Figure 5.16: Sensitivity to slice buffer size

More loads fall into this category for larger root tables because fewer conflicts exist. The probabilities of encountering larger numbers of hops diminish quickly.

The CSB is not only used by SVW, but also by re-executing deferred loads. The histogram for the number of hops required to re-execute deferred loads is shown in the bottom graph in Figure 5.15. The probabilities in this graph are normalized to the total number of loads committed, so they add to the percentage of loads which are deferred rather than to 100% (about 15%). This graph shows only the average across the memory bound subset of SPEC2006. Here, zero hops—no potential forwarding is encountered—is still the largest, however, one hop is also significant. Here, one hop is common because it typically represents forwarding on the first attempt. These two cases—zero and one hops—are the ones incur no latency penalty. Together, they account for 14% of all loads, or slightly over 90% of re-executing deferred loads in the default 256-entry root table configuration. The remaining 10% of loads require two or more hops and incur latency penalties. Larger numbers of hops become increasingly rare, with less than 0.005% of loads requiring six or more hops in any configuration. A few loads in *zeusmp* and *sphinx* require more than 20 hops, however, there are only several hundred such loads out of hundreds of millions of loads committed.

5.7.3 Slice Buffer Capacity

Figure 5.16 shows BOLT's sensitivity to slice buffer capacity. Three sizes are shown—128 (left/lightest), 256 (middle/darkest), and 512 (right/medium color) entries. Here, the

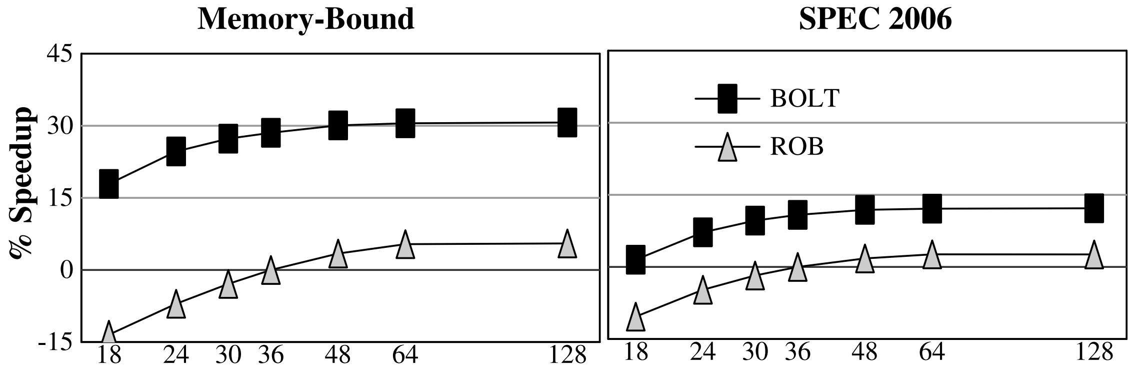


Figure 5.17: Sensitivity to issue queue size

programs most affected are those which benefit from latency tolerance the most—*milc* and *soplex*. For *milc*, the 128-entry slice buffer represents a performance loss of 20% compared to the default, while the 512-entry slice buffer provides an additional 16% performance. *soplex* loses 16% performance with the smaller slice buffer, but only sees a 2% gain from the larger slice buffer. The other memory-bound benchmarks lose 1–5% performance with the smaller slice buffer and see virtually no gain from the larger.

The performance degradation from the smaller slice buffer mitigates ED^2 reductions by 4% on the memory-bound programs and by 1% overall. For the larger slice buffer, all benchmarks except *milc* see at most a 1% change in either direction—either from the small speedup, or from the additional energy overhead with no corresponding performance benefit. The large speedup on *milc* results in approximately 5% lower (better) ED^2.

5.7.4 Issue Queue Capacity

Both the baseline's and BOLT's performance are affected by the size of the out-of-order window. Figure 5.17 shows the performance effects of varying the size of the issue queue, while holding all other structures (including the physical register file and ROB) constant. The left graph shows the average across the memory-bound subset of SPEC, while the right graph shows the average across all of SPEC.

The general trends of both graphs—performance monotonically increases with issue queue size; marginal performance gains decrease as the issue queue becomes larger; BOLT out-performs the baseline in all configurations—are as expected. There are, however, two interesting results. First, on average, BOLT with an 18-entry issue queue outperforms a vanilla ROB processor with a 128-entry issue queue, by 1.5%. Here, computation bound (*i.e.*, not memory-bound) programs suffer slowdowns of 3–15% on both processors, but the large performance gains of the memory-bound programs outweigh these losses. While this result is interesting, it is not practical—the large slowdowns are more significant than the small average gain. However, BOLT with a 30-entry issue queue *is practical*. Here, four benchmarks suffer slowdowns of 1–4% compared to the 36-entry issue queue baseline, with a 10% average speedup. By contrast, in a 30-entry issue queue ROB processor, almost every benchmark suffers a 1–7% slowdown over the baseline.

The second interesting result comes from analyzing the slopes of the curves. Between 24 and 64 issue queue entries, BOLT's curve has a significantly smaller slope than the baseline's—for the memory-bound programs, BOLT's slope is less than two-thirds of the baseline's. This trend corresponds to the fact that BOLT has a lower marginal benefit from additional issue queue entries. However, below 24 entries the trend reverses and BOLT's slope becomes *greater than* the baseline's. This cross-over occurs when the out-of-order window is too small to tolerate the latencies of L2 hits and floating point instructions. BOLT suffers relatively more from these problems because it suffers less from L2 and L3 misses. At the other extreme—beyond 64 entries—performance is unchanged for both the baseline and BOLT. BOLT has no need for a larger issue queue, while the baseline is constrained by the other aspects of the window—the ROB, physical register file, load queue, and store queue.

5.7.5 Issue Queue and Physical Register File Capacity

Figure 5.18 varies the ROB, physical register file, and issue queue sizes together. For each configuration, the ROB size is scaled, and the issue queue and physical register file are increased proportionally. Specifically, the physical register file size maintains the relationship #regs = 3/4 * ROBsize + 128, while the size of issue queue is kept slightly larger than

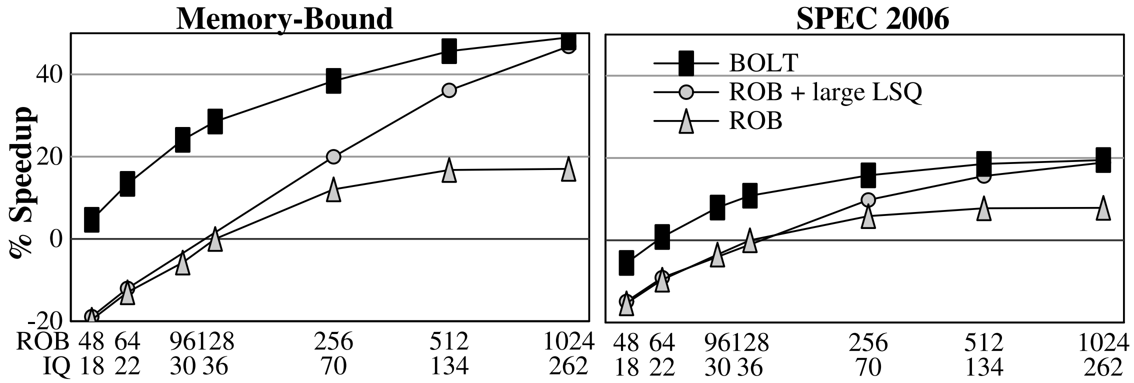


Figure 5.18: Sensitivity to ROB/IQ size

one quarter of the size of the ROB. The formula for the number of registers assumes that roughly three-quarters of in-flight instructions require output registers (the other quarter being branches or stores), and adds the 128 architectural registers required for a 2-way SMT processor. Two baseline configurations are shown—one with the conventionally sized (32/64) store and load queues, and one with BOLT's larger (128/256) store and load queues.

With conventionally sized load and store queues, ROB performance quickly becomes constrained—even though the ROB and issue queue can support a large window, the load or store queue fills up and the processor stalls. With the larger load and store queues, ROB performance continues to increase and catches up with BOLT when it has a 1024-entry ROB and a 262-entry issue queue.

Somewhat surprisingly, BOLT's performance continues to improve with larger ROB and issue queue sizes. Two factors contribute to this increase. First, the larger ROB effectively increases slice buffer size. This increase is not one-for-one as the ROB is sparse while the slice buffer is dense, but for very large ROB sizes, it can be significant. *milc* benefits significantly from this increase, reaching a speedup of 165%, which is 64% higher than the speedup obtained from the 512-entry slice buffer in Figure 5.16. The second factor is that many benchmarks are simply able to extract more ILP from the large window. This behavior is apparent in some programs which are not memory-bound, such as *wrf* (29%), and *gamess* (15%). Here, the larger window extracts more ILP by reordering more deeply around the moderate latencies of floating point instructions. This effect also manifests in some memory-bound programs which do not benefit from the larger slice buffer in Figure 5.16, such as *lbm* (63%), *zeusmp* (47%), *sphinx* (30%), and *gems* (29%). Here, BOLT has solved the memory-bound issue, but the larger window still provides ILP benefits with respect to other latencies *i.e.*, floating point.

5.7.6 Baseline Micro-architecture Sensitivity

In addition to measuring sensitivity to the sensitivity to the structures most relevant to BOLT, we also measured BOLT's sensitivity to several parameters of the underling micro-architecture. These experiments are briefly summarized here.

Superscalar issue width. BOLT can benefit more from a wider pipeline than a conventional ROB processor can. Without BOLT, memory-bound programs benefit little from wider issue pipeline—an 8-wide pipeline averages a 3% speedup over the 4-wide baseline for the memory-bound subset of SPEC. By contrast, 8-wide BOLT improves performance by an additional 9% compared to 4-wide BOLT.

Likewise, BOLT suffers more from a narrower pipeline. A 2-wide ROB processor yields an 18% slowdown on the memory-bound programs compared to the 4-wide baseline. 2-wide BOLT suffers a 33% slowdown compared to 4-wide BOLT, making it 5% slower than the original 4-wide baseline.

L3 cache capacity. Varying L3 cache capacity from 1MB to 32MB produces the expected behavior—BOLT's advantages increase for smaller cache sizes due to more misses—for most benchmarks. The exception is *mcf*, whose behavior is dominated by pointer chasing. With the default 8MB L3 cache, BOLT obtains a 16% speedup on *mcf*. BOLT with a 1MB L3 cache suffers a 53% slowdown compared to the original baseline while a conventional ROB processor with a 1MB cache suffers a 56% slowdown. This means that BOLT's relative advantage with the *smaller* L3 cache is only 3%, compared to 16% for the *larger* L3 cache. This effect arises from the fact that BOLT is unable to increase the performance of pointer chasing. As cache size decreases, more pointer chasing loads miss, causing the pointer chasing regions to be a larger portion of the execution time, which dilutes the gains BOLT can provide in other regions.

L2 cache capacity. Varying L2 cache capacity from 128KB to 1024KB (1MB) has a relatively small (*i.e.* less than 5%) performance impact on most benchmarks, even in the conventional ROB processor. The three benchmarks which it impacts by more than 5% are *libquantum*, *astar*, and *gcc*. *libquantum* speeds up by 55% when L2 cache size is increased to 512KB. Its performance does not change further for 1024KB. It experiences no change in performance decreasing the cache size to 128KB. *libquantum*'s working-set fits in a 512KB cache but not in a 256KB cache. BOLT loses all performance advantages on *libquantum* at the larger cache sizes because its latency tolerance never activates.

astar and *gcc* both lose 6% performance with the smaller (128KB) L2 cache size, and gain 6% (*gcc*) or 7% (*astar*) at the largest (1024KB). BOLT does not obtain significant performance advantages on either of these benchmarks at any of the cache sizes because their misses typically feed mis-predicted branches. BOLT is unable to exploit ILP, because the processor is executing down the wrong path. In *astar*, some MLP is exposed down the wrong path, but it simply pollutes the caches and harms performance slightly. Coupling BOLT with Control Independence [3, 33, 65, 66]—a technique to avoid squashing correct path instructions after a mis-predicted branch's control re-convergence point—might provide more performance opportunities in such situations.

For the remaining benchmarks, BOLT follows the expected trend—more relative benefits at smaller cache sizes—however, the relative differences are small because baseline ROB performance does not change much.

Chapter 6

iCFP: Load Latency Tolerance for In-order Processors

Load latency tolerance is beneficial to in-order cores as well as out-of-order cores. In-order cores can even benefit from applying load latency tolerance to L1 misses that hit the L2—unlike out-of-order cores which naturally tolerate such latencies, in-order cores cannot re-order instructions around them. This chapter describes iCFP (in-order Continual Flow Pipeline), an in-order load latency tolerant design that is analogous to BOLT. It also qualitatively compares iCFP to other in-order load latency tolerant designs. It omits a quantitative performance and energy analysis primarily because the top-down area-based relative-energy approximation we use (in Chapter 5 for BOLT) is inappropriate when the marginal area is more than a few percent of the baseline area.

The mechanisms used in iCFP (described below) are similar to those used in BOLT. By design, BOLT's mechanisms are largely core agnostic—they can be fitted onto any kind of core. The core agnostic design derives from the use of program-order interfaces. Miss-dependent instructions are deferred to the slice buffer in program order. Program order slices are re-injected into the execution core when misses return. This in-order interface can be attached to an in-order core—at in-order register read and completion—just as easily as it can be attached to an out-of-order core—at in-order rename/dispatch and (speculative) retirement.

6.1 iCFP: In-order Continual Flow Pipeline

iCFP (In-order Continual Flow Pipeline) [31, 32] is an in-order load latency tolerant design that has many structural and "algorithmic" similarities to BOLT. Figure 6.1 shows BOLT (top) and iCFP (bottom). Although the underlying pipeline is different, the key latency tolerance structures (shaded grey)—the slice buffer and chained store buffer—are the same.

Like BOLT, iCFP uses checkpoint-backed speculative retirement and explicitly distinguishes between tail and deferred instruction. Both designs slice out miss-dependent instructions, captures their miss-independent inputs, and defers them to a program-order slice buffer. Deferring instructions release their resources—here, the pipeline latches themselves—and free them for younger instructions. iCFP also makes multiple passes over the slice buffer—initiating passes as misses return—re-injecting miss-dependent instructions for re-execution. As with BOLT, re-execution can be multi-threaded with tail execution and filtered by antidote bit-vector based pruning mechanisms[1].

6.1.1 Register Management

iCFP's register management scheme can be thought of as the in-order analog of BOLT's. Both designs treat a re-executing deferred slice as a second self-contained thread. Although an out-of-order processor has per-thread map tables, and in-order processor (which does not have register renaming) has per-thread register files. iCFP "borrows" a second register file to re-execute a slice. This "scratch" register file—which starts out empty at the beginning of every slice buffer pass—is used for register communication within the slice. This is analogous to BOLT staring every slice buffer pass with an empty "scratch"

[1] iCFP was originally proposed with simpler multi-poison bit pruning mechanisms. As BOLT's more sophisticated pruning can be used without modification, it provides a more appealing design choice.

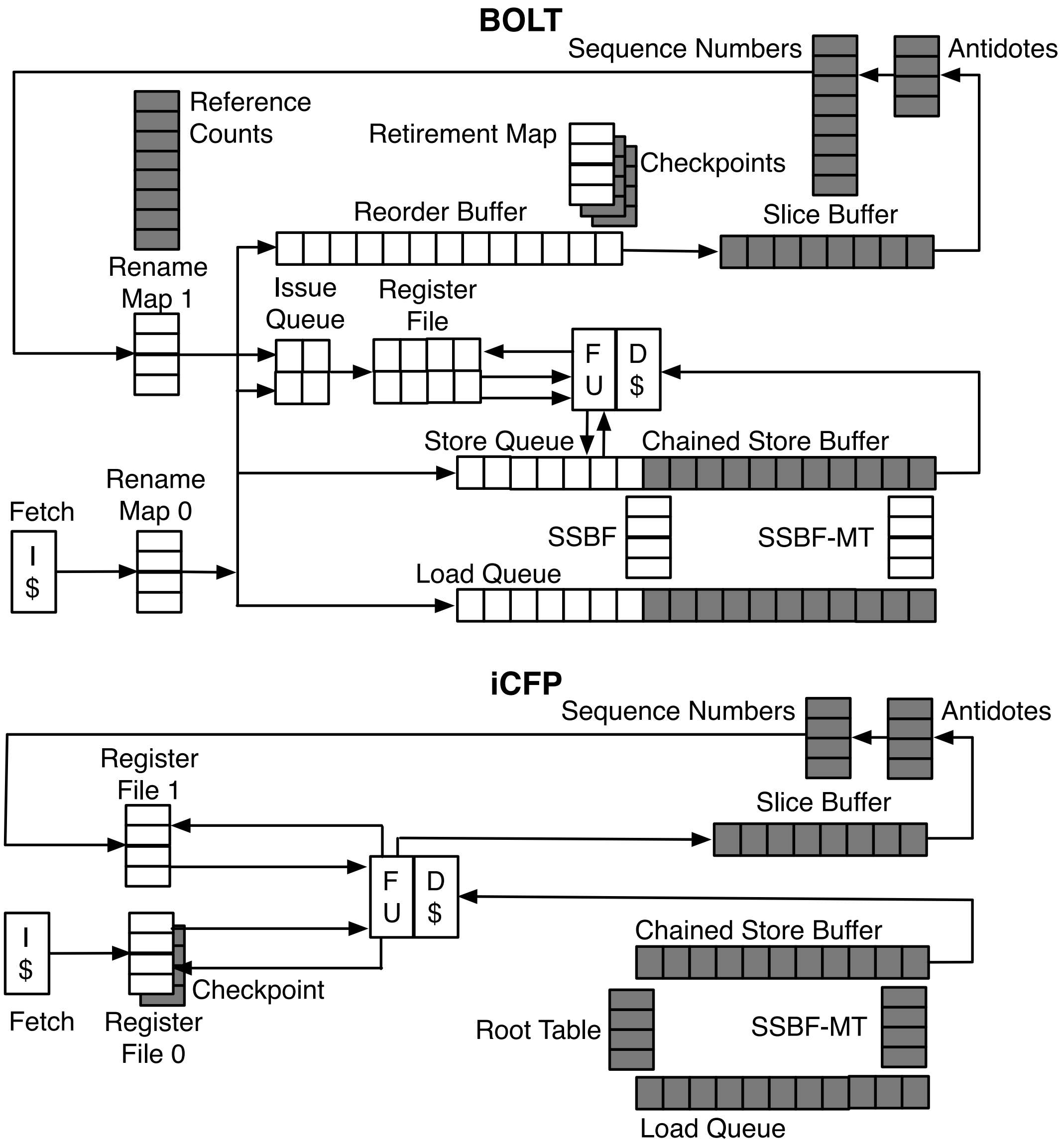


Figure 6.1: Similarities between BOLT (top) and iCFP (bottom). Additions for load latency tolerance are shaded grey.

map table.

Like BOLT, iCFP must not only ensure proper register communication *within* a slice but also that register values which are live-out of a slice are correctly integrated with tail register state. This means that re-executing slice instructions must occasionally write both register files—the “scratch” one used for slice communication and the “working” one used by tail instructions. A re-executing slice instruction must write the tail register file if-and-only-if its output value has not been overwritten by a younger instruction. To implement this test, iCFP tags each register with a sequence number. When an instruction initially executes, it writes its own sequence number to the register tag. Re-executing instructions then compare sequence numbers for equality to see if they should write the tail register file as well. This test is analogous BOLTs handling of live-outs (Section 3.2.3). It is slightly different in that BOLT performs the live-out checks when re-renaming instructions and leverages renaming by assigning the re-executing instruction the physical register still mapped by the tail. iCFP performs the live-out check as instructions prepare to write back their output values, and writes the value back into both the working register file and the tail register file if needed. Although it may appear that this design places the access latency of the sequence number table in the critical execution path, this is not the case. Because the sequence number operations are independent of the computation, the sequence number tables may be manipulated one pipeline stage before writeback—in parallel with the last computation stage—to avoid adding latency.

Another difference between iCFP and BOLT is that iCFP must checkpoint register values instead of mappings. Consequently, iCFP only uses a single checkpoint, whereas BOLT uses two. This difference occurs because iCFP’s checkpoints are not only more expensive—containing 64-bit values instead of 8-bit mappings—but also because if iCFP used multiple checkpoints, it would need the ability to update the values contained in the intermediate ones as slice instructions re-execute. The use of a single checkpoint allows iCFP to implement its register checkpoints using low cost “shadow bit-cells” [22], which do not support incremental updates.

6.1.2 Store-Load Forwarding

The chained store buffer (CSB) used in BOLT was originally proposed as part of iCFP [31]. CSB's non-speculative forwarding is especially important in iCFP. Conventional in-order processors do not speculatively re-order loads with respect to older stores from the same thread. Consequently, they do not have mechanisms in place to verify same-thread store-load speculation. The non-speculative nature of CSB's forwarding spares iCFP from needing to add mechanisms to detect mis-speculations. The fact that the CSB can forward in all cases—even if it takes a few extra cycles—is also important for the in-order processor. If difficult forwarding cases (*i.e.* partial forwardings, un-aligned loads and stores, or forwarding from a store other than the youngest to an address) required stalling until certain stores completed, then the load may block the *entire* pipeline for a long time. This situation is worse than the analogous one in an out-of-order processor where instructions may be re-ordered around the stalled load, permitting execution to continue. This situation would be worse still if the stalling load were a re-executing slice instruction, in which case stores would not complete until it executes—requiring a squash to break the deadlock. With a CSB, these cases can all be handled within a few cycles, allowing the pipeline to stall briefly then resume execution. Low area and low energy overheads are even more important in in-order processors, so the fact that a CSB has is area and energy efficient is another benefit.

6.1.3 Load Verification

The CSB provides non-speculative forwarding, so iCFP does not need to verify loads with respect to stores from the same thread. It does, however, need to verify loads with respect to stores from *other* threads. This inter-thread verification is only applicable when latency tolerance is active and loads do not retire immediately after execution. The original iCFP proposal verified loads using a signature. The signature is a bloom filter of addresses read by loads in the deferred region. Whenever the data cache invalidates a block, it tests the block's address against the signature and squashes to the architectural checkpoint on a "hit". The signature is cleared whenever the deferred region is committed or squashed. A signature admits the possibility of false positives, but is simple and area efficient.

Another possibility would be to use "speculative read" bits in the data cache—when a load reads the data cache it sets a bit in the tag of the block that it reads. If the block is evicted while the speculative read bit is set, the deferred region is squashed. The speculative read bits are flash cleared when the deferred region is squashed or committed. This design could be "free" if speculative read bits already exist to support transactional memory [4, 8, 13, 51] or other forms of speculation [9, 64].

iCFP can also use SVW-MT. However, the area and energy overheads for SVW-MT may be undesirable in an in-order core. Unlike an out-of-order processor, a non-latency-tolerant in-order processor does not have a load queue. This means that the marginal area and energy overheads of SVW-MT have to include not only the load queue itself, but also the actions of capturing addresses and values in the load queue during load execution.

6.2 Other In-order Load Latency Tolerance Designs

There have been other designs which target long latency misses in in-order processors. These designs—Runahead Execution (Section 6.2.1), Flea-flicker Multi-pass Pipelining (Section 6.2.2), Simple Latency Tolerant Processor (Section 6.2.3), and Sun's Rock (Section 6.2.4)—are briefly discussed here. The differences between iCFP and these other designs are summarized at the end of this section in Table 6.1.

6.2.1 Runahead Execution

Runahead execution was initially introduced to exploit MLP in an in-order processor [20]. Runahead in an in-order processor is similar to out-of-order Runahead. On a long latency miss, the processor takes a register file checkpoint, propagates poison from the missing load, and proceeds in Runahead mode until the miss returns. A forwarding cache is used to handle store-to-load forwarding in Runahead mode. When the primary "triggering" miss returns, the processor flushes the pipeline, restores the register checkpoint, and resumes conventional execution. The need to flush the pipeline limits in-order Runahead's applicability to short L1 cache misses because it is rare for the performance gain (*i.e.* MLP exposed under an L2 hit) to overcome the performance penalty (*i.e.* pipeline refill time). Runahead's pruning techniques [53] are applicable to both in-order and out-of-order Runahead. Experiments show that iCFP out-performs in-order Runahead—in part because iCFP can be profitably applied to L2 hits—with lower re-execution overhead.

An alternative implementation of Runahead. IBM's POWER6 processor is in-order and uses Runahead [43], although POWER6's implementation of Runahead is slightly different from that described in academic literature and here. The first difference is that POWER6 does not require a complete pipeline flush when exiting Runahead mode. Instead, it uses a 64-entry buffer of decoded instructions from which re-execution can begin. This buffering makes POWER6's Runahead more useful under shorter misses because there is almost no pipeline refill latency. The second difference is that POWER6 does not checkpoint the register file. POWER6 handles registers differently depending on whether one or two threads are active. If two threads are active, then Runahead mode results may be bypassed to consumers while the producing instruction is in-flight, but are simply discarded when the producer reaches writeback. Any instructions which depend on discarded results may not execute. While this occurs, the thread in Runahead mode is given lower priority. If only one thread is active, then Runahead mode borrows the other register file.

6.2.2 "Flea-flicker" Multipass Pipelining

Flea-flicker Multipass Pipelining [6] targets ILP under short misses. On a miss, Multipass checkpoints register state *and* the pipeline latches and begins propagates poison. Like Runahead, Multipass re-executes all instructions after a miss returns. Unlike Runahead, Multipass buffers these instructions during their initial execution and records the output values of miss-independent instructions. When the miss returns, Multipass restarts execution from the register checkpoint. As it re-executes, it uses the recorded output values of completed instructions to execute dependent instructions in parallel. With each pass, more values become known as the misses they depend on return. The final pass over a given dynamic instruction region executes with significantly more parallelism than conventional execution permits. Multipass must re-execute all instructions in a given region—potentially multiple times—meaning it exploits ILP under a miss, but only partially. Multipass performs "best effort" forwarding under a miss and relies on the fact that all loads will re-execute to ensure correctness. Experiments show that Multipass typically (but not always) slightly out-performs Runahead [31]. iCFP out-performs Multipass and has lower re-execution overhead.

6.2.3 Simple Latency Tolerant Processor

Simply Latency Tolerant Processor (SLTP) is an adaptation of CFP to in-order processors [55]. Like CFP and iCFP, SLTP buffers and re-executes only miss-dependent instructions. Unlike CFP (and iCFP), SLTP is only capable of re-executing a given deferred instruction once. If a re-executing instruction experiences a long latency miss—either because it already initiated a miss which has not yet returned, or because it is a miss-dependent load which just learned its address—the pipeline blocks until the miss returns. This limitation comes from SLTP's register management scheme. Specifically, SLTP uses a single register file and two checkpoints. SLTP tracks only register poison status, it does not track the last writer instruction identity. SLTP uses the second checkpoint to capture the current tail register state immediately before beginning re-execution—the first checkpoint captures architectural state prior to the "primary" miss. SLTP then re-executes *all* instructions in the slice buffer and performs a conditional restore operation on the second checkpoint to merge the register state. This operation restores the checkpointed value for any register which was not poisoned in the checkpoint, and keeps the value in the register file—produced by slice re-execution—for any instruction which was poisoned in the checkpoint.

For store-to-load forwarding, SLTP combines a Store Redo Log (SRL) [26] with a transactional data cache—a cache which can support speculative writes which are either committed or aborted in bulk. While executing under a miss, stores are buffered in the SRL and speculatively written to the transactional data cache. Loads do not need to forward from the SRL, as they can receive speculatively written values from the data cache. However, the data cache does not track poison information, so SLTP uses memory dependence prediction to determine which loads are likely to be miss-dependent via store-to-load communication. SLTP verifies this speculation with a set-associative load queue. Then the primary miss returns, SLTP aborts all speculative writes to the data cache, invalidating all speculatively written lines and causing them to be re-fetched from the L2 when they are next accessed. As SLTP re-executes miss-dependent instructions from the slice buffer, it speculatively re-completes stores from the SRL into the data cache, and searches the set-associative load queue for ordering violations. When re-execution of all miss-dependent instructions and re-completion of all stores finishes, the data cache writes are committed in bulk.

Experiments show that SLTP frequently under-performs not only iCFP, but also Runahead and Multipass [31]. SLTP's performance suffers for two reasons. First, SLTP's slice re-execution is blocking—it is unable to reapply latency tolerance to any long latency misses which occur during slice re-execution. Second, SLTP is unable to forward from the SRL during slice re-execution. Re-executing loads must wait for any stores that they forward from to re-complete to the data cache prior to execution. These re-completions are performed from the SRL in program order and are slow—the blocks they affect have all been invalidated in the L1 cache by the transactional abort at the start of re-execution and must be re-fetched from the L2.

6.2.4 Sun's Rock

Sun's Rock processor is another in-order load latency tolerant design based on CFP [16]. Like iCFP, Rock buffers and re-executes miss-dependent instructions. Both designs multithread the re-execution of deferred slices with the execution of new instructions at the tail. The major differences between iCFP and Rock are the way they handle the register state of re-executing instructions. Additionally, Rock was implemented in silicon, while iCFP has only been simulated.

Rock's register state management scheme is based on multiple register checkpoints which divide the deferred region into multiple segments. Rock only re-executes deferred instructions in the *oldest* checkpoint (*i.e.*, deferred region segment) at any given time. As it does so, it fills in the register value "holes" in the next checkpoint corresponding to outputs that were previously poisoned. When all instructions in the oldest checkpoint have completed, Rock releases the oldest checkpoint, then makes a re-execution pass in the newly oldest checkpoint. When *all* outstanding instructions have been re-executed, Rock merges the register state generated by the slice instructions' outputs with the working tail register file.

There are two disadvantages to this register management scheme relative to iCFP's. First, a deferred instruction may not re-execute as soon as possible. Re-execution of any instruction in the second checkpoint is serialized with the return of *all* misses in the oldest checkpoint. Importantly, the re-execution of misses which return out-of-order (*i.e.* those which hit different levels of the memory hierarchy) is serialized on all loads in previous checkpoints. Second, even when an instruction has re-executed, its output is not immediately available to new instructions at the tail. This unavailability results in un-necessary deferrals, as in D-KIP. These problems are most significant when they affect mis-predicted branches—either delaying their resolution or causing them to needlessly defer and squash a checkpoint rather than just flush the front-end of the pipeline. Although mis-predicted branches which depend on last-level cache misses are relatively rare [76], those that depend on L1 misses are fairly common.

	Runahead	Multi-Pass	SLTP	Rock	iCFP
Register management					
Checkpoints	1	1 + latches	2	0	1
Register files	1	1	1	10	2
Load and store management					
Store forwarding	Forwarding $	Forwarding $	Transactional D$	Associative store buffer	Chained store buffer
Load verification	None	Re-exec	Set-assoc load queue	D$ bits	Indexed load queue, signature, *or* D$
Re-execution					
Insns buffered	None	All	Miss-dep	Miss-dep	Miss-dep
Insns re-executed	All	All	Miss-dep	Miss-dep	Miss-dep
Multi-threaded	No	No	No	Yes	Yes
Non-blocking	N/A	Yes	No	Yes	Yes
Start at load	Oldest	Oldest	Oldest	Oldest chkpt	Any
Visible to tail	N/A	N/A	N/A	On final pass	Immediately
Pruning	Useless/ Overlapping	None	None	None	Miss/Join/ Pointer
Overall Effects					
Performance	Low	Low	Low	Moderate	High
Static cost	Low	High	High	Moderate	Moderate
Dynamic cost	High	High	High	Moderate	Low

Table 6.1: Comparison and contrast of in-order load latency tolerant designs.

Rock's register state management limits its ability to expand the window—it squashes to checkpoints due to deferred mis-predicted branches relatively frequently. Because of this limitation, Rock does not need a scalable store buffer design. Instead, it uses a 64-entry conventional associative store buffer. This store buffer provides non-speculative forwarding, so Rock does not need to verify loads with respect to same-thread stores. Rock uses speculative read bits in the data cache to verify loads with respect to stores from other threads.

6.2.5 Qualitative Comparison Summary

Table 6.1 summarizes the differences between these in-order load latency tolerant designs. The bottom section provides a high-level comparison of the performance gains and energy costs associated with each design. Note that Rock uses eight "checkpoints" and two working register files, however, Rock's checkpoints require updates so all ten of these are listed as register files in the table.

Performance. The performance categorizations for Runahead, Multi-Pass, SLTP, and iCFP come from experiments performed in previously published work [31]. At the time of that work, Rock's details were not publicly available. Based the details in a subsequent publication about Rock [16], Rock should outperform Runahead, Multi-pass, and SLTP, but under-perform iCFP. Rock shares many of iCFP's advantages over the other three designs, but Rock's slice re-execution has limitations which iCFP's does not.

Static area and energy overhead. Runahead clearly has the lowest static area and energy overheads, as it requires only a register checkpoint and a forwarding cache. iCFP and Rock both have moderate overheads. They require a slice buffer, register checkpoints, a larger store buffer, and a load verification mechanism. Of the two, Rock has slightly higher overheads as it requires more checkpoints and uses an associative store buffer—although there is no reason Rock could not use CSB instead. Multi-pass has higher structural overheads than these two designs. It must buffer *all* instructions under a miss, and relies on the ability to checkpoint all of the pipeline latches to restart execution quickly. Although SLTP only buffers miss-dependent instructions in its slice buffer, its other requirements—memory dependence prediction, set-associative load queue, and transactional data cache—give it the largest structural overheads.

Dynamic energy overhead. iCFP has the lowest dynamic energy overheads. First, it has the lowest re-execution rates, as it re-executes only miss-dependent instructions, *and* applies pruning mechanisms to reduce re-executions. Second, iCFP's new structures (*e.g.*, CSB) have low per-access energy. Rock has slightly worse dynamic energy consumption than iCFP in both regards. Rock does not have pruning mechanisms (although it could use them), *and* its re-execution limitations lead to more checkpoint squashes than iCFP. Rock's use of an associative store buffer also consumes more energy per-access than iCFP's CSB.

Runahead and Multi-pass have higher re-execution overheads because they re-execute *all* instructions rather than just the miss-dependent ones. Although SLTP only re-executes miss-dependent instructions, it has high dynamic overheads. First, it uses memory dependence prediction for loads. Second, it must complete all stores under a miss *twice*. Third, as it completes stores a second time, it must search a set-associative load queue. Fourth and finally, the manner in which it uses a transactional data cache increases L2-cache accesses. Dirty blocks must be cleaned by writing them back to the L2 before they may be speculatively written. The speculative writes are then always aborted, forcing L2 reads to re-fill the affected blocks.

Chapter 7

Conclusions

Load latency tolerance—the ability to continue execution in the presence of long latency cache misses—is a previously proposed technique to improve the performance of memory-bound programs. However, previous load latency tolerant designs have been energy inefficient—adding too many structures and re-executing too many instructions to justify their performance gains. BOLT and iCFP are energy efficient load latency tolerant designs for out-of-order and in-order load latency tolerance respectively. These designs frame slice re-execution as an alternative use of existing multi-threading hardware, use a chained store buffer for efficient store-to-load forwarding, and use pruning mechanisms to reduce re-execution overheads.

Experiments presented here show that BOLT obtains 29% speedup on the memory-bound subset of SPEC and 11% overall. These speedups are approximately twice those of CFP (15% and 5% respectively) , the prior state-of-the-art load latency tolerant design. The same experiments show that BOLT is more energy efficient, as it reduces (improves) ED^2 by 30% on the memory-bound programs (13% overall) while CFP only improves it by 11% (1% overall). BOLT is also more robust than CFP, which is susceptible to slowdowns on branch mis-prediction bound programs. iCFP has previously been shown to outperform other in-order load latency tolerant designs. No quantitative energy analysis has been performed to compare iCFP to other designs, but qualitatively, it should be superior.

BOLT moves out-of-order load latency tolerance from an interesting research idea to a design which can actually be implemented. One key part of this is that BOLT adds onto—rather than completely replaces—a conventional core. This reduces cost, simplifies the design, and allows execution to fall back to a conventional mode as needed. BOLT also includes an efficient implementable design for the load and store queues. Finally, BOLT's deferral and re-execution pruning mechanisms prevent pathological energy overheads. iCFP represents a similar transition for load latency tolerance in in-order processors.

7.1 Future Work

There are many potential directions to explore or further evaluate for both iCFP and BOLT. Many of these depend on having a more detailed "bottom-up" energy model which can accurately model more significant changes to various aspects of the pipeline. A detailed energy model would also allow for a quantitative energy evaluation of iCFP.

Impact on out-of-order design parameters. The results in Sections 5.7.4 and 5.7.5 suggest that BOLT may shift the "sweet spot" for an out-of-order processor's design parameters. Specifically, the issue queue, register file, and ROB can be reduced because load latency tolerance machinery can provide performance under L2 misses (L3 hits). It is possible that BOLT could change the capacity/performance trade-offs of the window structures in such a way that it "pays for" its other structures. A detailed energy model is required to perform this analysis.

Dynamic application of DVFS. The results in Chapter 5 for DVFS ("TurboBoost") assumed the voltage and clock frequency were fixed for the entire duration of the application. A more detailed analysis dynamically change the clock frequency according to the current situation. It is possible that load latency tolerance may benefit from different policies to adjust voltage and frequency than conventional processors (*e.g.*, scaling frequency and voltage up under MLP and down under pointer chasing). Experiments on such policies would likely benefit from a more detailed energy model, which can accurately account for different activities at different voltages.

Pointer chasing. Although load latency tolerance can efficiently handle most long latency misses, it is helpless against pointer chasing. In addition to its implications for DVFS, this suggests that resources should be allocated to reduce the number of pointer chasing misses. Such re-allocation could include a redesign of the prefetching mechanisms to specifically target pointer chasing loads [21,70,71,83], or new cache replacement policies to prioritize retention of pointer chasing related blocks.

Combination of value prediction and slice re-execution. Although value prediction and slice re-execution are two different approaches to load latency tolerance, they are not incompatible. Combining the two approaches could lead to a design which benefits from the strengths of both. If a load's output is correctly predicted, then no re-execution overhead is incurred. If a load's output is incorrectly predicted, then the load and its dependents are re-executed and no squash is required.

Synergy with other research proposals. Chapter 5 alluded to two other research ideas which are potentially synergistic with load latency tolerance—Control Independence [3, 33, 65, 66] and Wrong Path Events [5]. Although Control Independence (CI) and BOLT improve performance in different ways—one attacks memory-bound programs while the other attacks branch mis-predictions—their mechanisms cannot be directly combined. Combining BOLT with Wrong Path Events and shutting down latency tolerance when execution is predicted to be on the wrong path should be straight forward, although issues may arise from if CI is also included in the design.

Appendices

Appendix A

SVW-MT Overhead and Optimizations

Both the baseline and BOLT (and Runahead) use SVW and SVW-MT for their load verification. While SVW has been evaluated in previous work, the multi-threaded aspects of it in SVW-MT have not. Although the SPEC benchmarks are single threaded, SVW-MT may produce re-executions in the presence of data cache replacement evictions. It is therefore important to establish that SVW-MT has minimal (negative) performance impact with a small SSBF-MT.

To measure the performance impact of SVW-MT, comparisons are made against an ideal SSBF-MT—all loads skip SVW-MT, producing the effect of a perfect SSBF-MT with infinite bandwidth. As both the baseline ROB processor and BOLT use SVW-MT, either could be used for these experiments, as long as one micro-architecture is used consistently throughout. The choice is not significant—the primary factor is the time between a cache block's use and its eviction. BOLT is selected here because it is more important to show that SVW-MT is effective in BOLT than it is to show it is effective in a ROB processor.

Figure A.1 shows BOLT's performance with varying SSBF-MT sizes both with and without the optimizations described in Section 3.4.2. The data is shown as the slowdown relative to BOLT with ideal SVW-MT. The left graph shows the *average* performance impact of BOLT with realistic SVW-MT. Here, the SSBF-MT's size is varied from eight

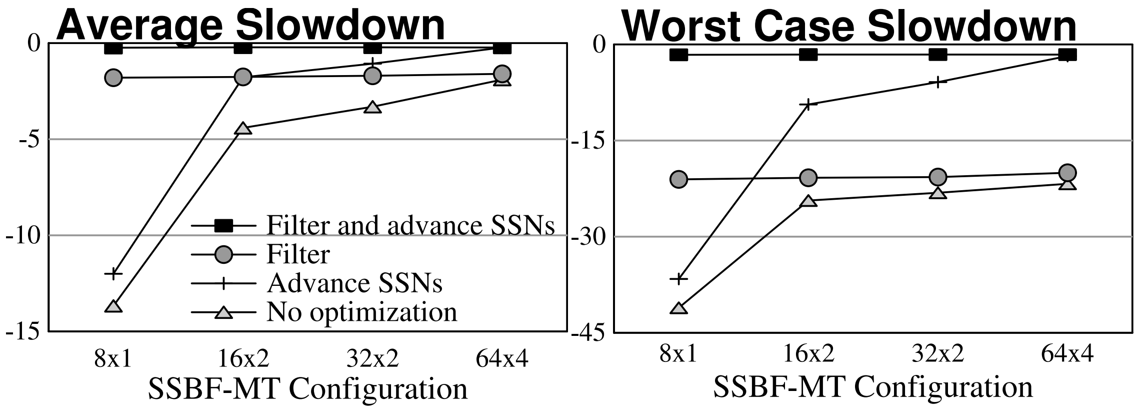


Figure A.1: The impact of SVW-MT on performance for varying SSBF-MT sizes, with and without optimizations. Each is compared against BOLT with ideal SVW-MT. Left: Average performance impact of realistic SSBF-MTs. Right: Worst case performance impact.

entries to 256 entries, with larger sizes having higher associativity as well. Each configuration is represented on the x-axis as (Sets x Ways). The right graph shows data from the same experiments, but plots *worst case* slowdown. Note that the right graph has a different scale than the left.

With both optimizations—SSBF-MT write filtering and SSN advancement—SVW-MT poses little performance overhead and is virtually insensitive to SSBF-MT size and configuration. This configuration is shown on the graphs with the black boxes. Here, only two benchmarks see slowdowns of 1% or more compared to the ideal SVW-MT. These benchmarks (*cactus* (1.4%) and *gems* (1.5%)) suffer from the limited bandwidth of SVW-MT rather than from excessive re-executions. With these optimizations re-execution rates are quite low—the highest in BOLT's default 16-set, 2-way configuration is *milc* with 36 re-executions per 1,000,000 μopss committed. Using the smaller 8-set direct-mapped configuration has no significant impact on performance. The baseline and BOLT use the 16-set, 2-way table as their default configuration on the assumption that multi-threaded programs may use the SSBF-MT somewhat more aggressively.

The low re-execution rates and performance impacts come from the efficiency of the write filtering mechanism. With the SSN advancement optimization turned on as well, the filtering mechanism eliminates over 99.7% of all SSBF-MT writes in *every* benchmark compared to no filtering at all. Here, there are on average two SSBF-MT writes *per million* μops, with the worst case benchmark (*milc*) seeing only 16 writes per million μops.

The filtering mechanism is less effective without the SSN advancement optimization—filtering relies on the advancement of $\text{SSN}_{\textsf{store-complete}}$ to determine when a write may safely be eliminated. With filtering and no SSN advancement, SSBF-MT writes increase by three orders of magnitude—to 3,017 per million μops, with the worst case benchmark (*mcf*) seeing 67,887 writes per million μops. The performance of this configuration is shown on the graphs of Figure A.1 by the dark grey circles. Here, benchmarks suffer performance degradation when they enter long periods with no stores—even if an eviction occurs well before a load enters the window, SVW-MT cannot ascertain safety because its notion of "time" is tied to stores. The worst case for this is *soplex* which suffers a 20% slowdown over the ideal SVW-MT configuration, although *libquantum* suffers a 7% slowdown, and several other benchmarks degrade slightly. This configuration is also insensitive to SSBF-MT size.

The line marked with crosses shows performance when only the SSN advancement optimization is applied, with no filtering. Here, performance approaches the ideal case with a large, 4-way associative SSBF-MT. SSN advancement prevents the pathological behaviors which can occur in long periods without stores, but small SSBF-MTs suffer from capacity problems. Writes increase yet another order of magnitude—to an average of 12,660 SSBF-MT writes per million μops.

The final line, marked with a light grey triangle, shows the performance with no optimizations. Here, both problems manifest—long periods without stores can cause pathological re-executions, and smaller SSBF-MTs suffer from conflict induced re-executions.

Storing partial SSNs. The storage space required for these optimizations can be cut in half by storing only the upper eight bits of the 16-bit SSNs in the data cache blocks. Figure A.2 shows the performance impact of storing varying numbers of SSN bits in the data cache. Here, performance impacts are shown as the largest (right) and average (left) slowdown compared to storing all 16 bits. All configurations use the same 16x2 SSBF-MT

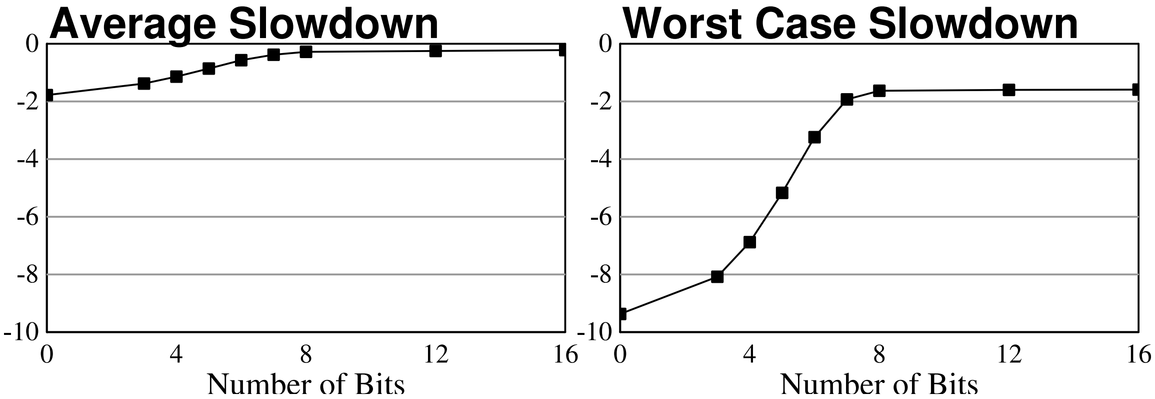


Figure A.2: The impact of number of bits stored for SVW-MT's filtering. Each is compared against storing the full 16 bits. Left: Average performance impact of realistic SSBF-MTs. Right: Worst case performance impact.

and the SSN counter advancement optimizations. The most important observation from these graphs is that using only eight bits results in nearly identical performance to using all 16 bits. The most significant impact of using eight bits is a 0.86% slowdown on *mcf*, which has bad locality and thus replaces cache blocks relatively quickly. The next largest slowdown is 0.17% on both *milc* and *soplex*. Storing only seven bits would also produce acceptable performance. Here, *mcf*'s slowdown increases to 0.98%, closely followed by *sphinx* at 0.93%. In total, seven of the benchmarks observe a slowdown of 0.2% or more when only seven bits are stored. With fewer than seven bits, however, performance begins to drop off rapidly.

Structural Overheads. The filtering SVW-MT requires to be effective imposes some small area and energy overheads. Specifically, the SSNs used for filtering must either be added to the data cache tags, or stored in a separate structure. Since these bits are only accessed at times when the data cache tags would be accessed anyways, it is most likely more efficient to simply add them to the cache tags—saving the overhead of the additional "edge" logic for a new structure. Unfortunately, CACTI does not support an easy way to model such an addition. CACTI can, however, model the area and energy of a separate structure, accessed after the data cache tags, to track this information.

For a 2-thread SMT processor with a 512-block cache, storing eight bits of each SSN, the structure required to keep the filtering information has an area of 0.027 mm^2, a read energy of 0.174 pJ, and a write energy of 0.028 pJ. To put these numbers in perspective, the area is about 0.08% of the area of one core of a Core i7. The read energy—consumed when an eviction occurs to test if the SSBF-MT should be written—is roughly 4% of the energy required to search a conventional load queue. The write energy—consumed whenever a load executes—is roughly 0.6% of the energy required to search a conventional store queue. These numbers do not indicate area and energy overheads, but rather slight reductions in the area/energy improvements provided by SQIP and SVW.

Appendix B

Energy Model Sensitivity Analysis

The energy model used in Chapter 5 combines instruction execution overhead and cycle counts obtained from timing simulation with structural overheads derived from area models. This appendix examines the assumptions used in this model.

B.1 Dynamic Energy Conversion Factor

The energy model scales the dynamic energy overhead of new structures by a conservative factor of two relative to their area ($\delta = 2$ in the equations of Chapter 5). Figure B.1 shows the variation in energy (top) and ED^2 (bottom) for BOLT under various values of this constant—0, 1, 2, and 8.

The difference across the 0–2 range is negligible. Increasing the dynamic energy penalty to eight shows increases in the energy overhead of 3–5% on all benchmarks (including the non-memory-bound benchmarks not pictured). This insensitivity is because most of the dynamic overhead comes from executing additional instruction on the existing core—the overheads of the new structures are relatively small by comparison.

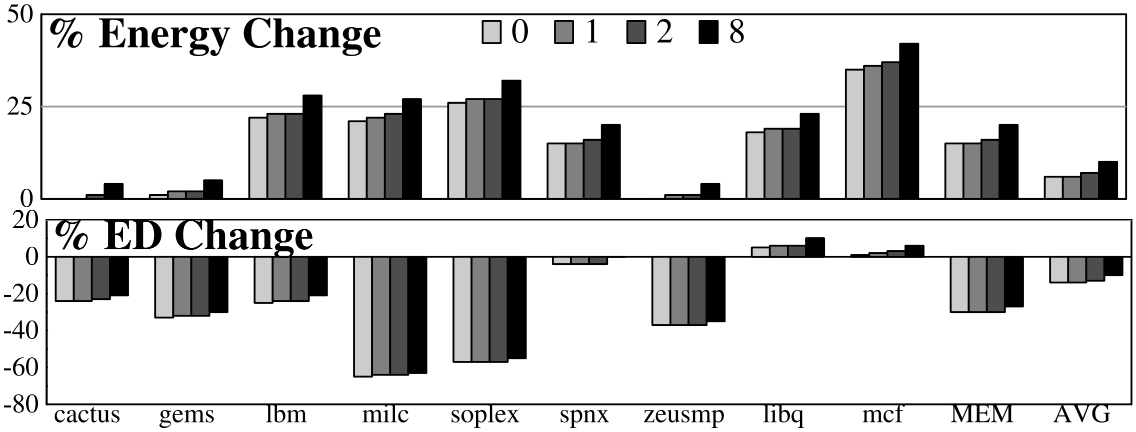


Figure B.1: Comparison of dynamic energy model parameter δ.

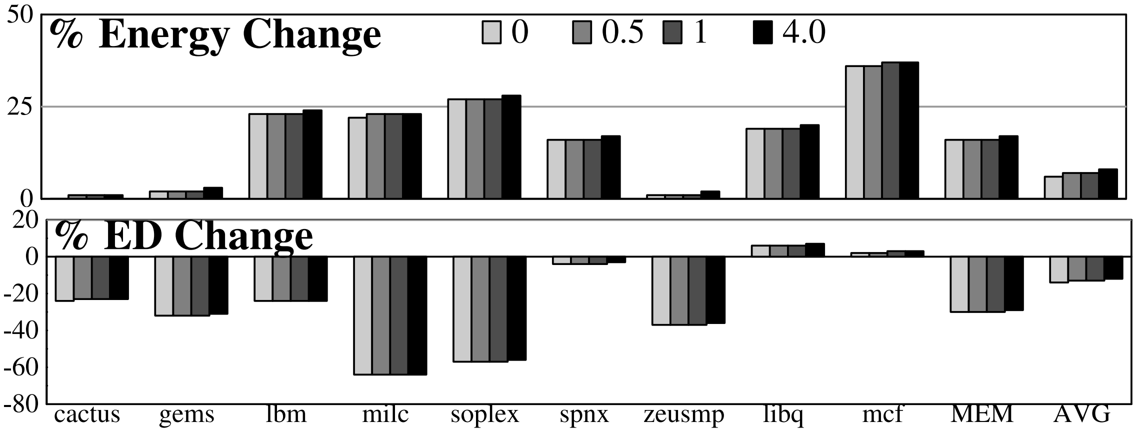


Figure B.2: Comparison of static energy model parameter σ.

B.2 Static Energy Conversion Factor

The energy model assumes that new structures leak at the same rate as the average point on the chip ($\sigma = 1$ in the equations of Chapter 5). Figure B.2 shows the sensitivity of the energy model to the static energy parameter σ. The graph shows the energy (top) and ED^2 (bottom) for when σ is set to 0, 0.5, 1, and 4. As with the dynamic energy scaling parameter, δ, varying σ slightly has negligible effects on total energy. Again, most of the static energy difference come from reducing execution time (*i.e.*, the time during which the existing core leaks) rather than from adding new structures.

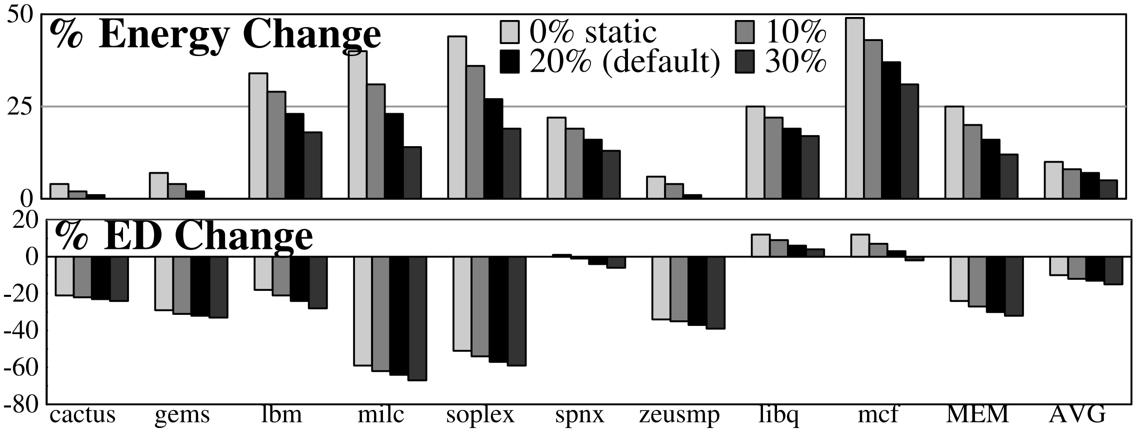


Figure B.3: Sensitivity of energy model to static/dynamic split.

B.3 Static vs. Dynamic Split

The final, more significant, assumption in the energy model is a 20%/80% split between static and dynamic energy. Figure B.3 explores the impact of this assumption by showing the energy (top) and ED^2 (bottom) for BOLT when the energy model assumes a 0/100 split (left), a 10/90 split ,the default 20/80 split, and a 30/70 split (right).

Energy consumption decreases as the static component increases. Instruction execution energy overheads decrease as dynamic energy is discounted. At the same time, BOLT's speedups reduce static energy which now accounts for a larger fraction of the total. The leakage of the new structures for BOLT has a larger weight, but the structures are small. The magnitude of the changes is proportional to the magnitude of BOLT's speedups and re-execution overheads. On programs where BOLT does very little, the assumptions about the static/dynamic split do not matter much—changing energy consumption by at most 2% between the 10/90 split and the 30/70 split on non-memory-bound programs. On memory-bound programs, the variation is larger, 3–17%. As with any technique that improves performance by executing more instructions, BOLT's energy and ED2 benefits increase as leakage becomes more dominant.

Bibliography

[1] Increasing Performance with Intel Turbo Boost Technology, 2008. http://www.intel.com/technology/turboboost/index.htm.

[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 423–434, Dec. 2003.

[3] A. Al-Zawawi, V. Reddy, E. Rotenberg, and H. Akkary. Transparent Control Independence (TCI). In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 448–459, Jun. 2007.

[4] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. 11th Intl. Symp. on High Performance Computer Architecture*, pages 316–327, Feb. 2005.

[5] D. Armstrong, H. Kim, O. Mutlu, and Y. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *Proc. 37th Intl. Symp. on Microarchitecture*, pages 119–128, Dec. 2004.

[6] R. Barnes, S. Ryoo, and W-M. Hwu. "Flea-Flicker" Multipass Pipelining: An Alternative to the High-Powered Out-of-Order Offense. In *Proc. 38th Intl. Symp. on Microarchitecture*, pages 319–330, Nov. 2005.

[7] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. In *2004 IBM P=AC^2 Conf.*, Oct. 2004.

[8] C. Blundell, J. Devietti, E Lewis, and M. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 24–34, Jun. 2007.

[9] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.

[10] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips. In *Proc. 8th Intl. Symp. on High Performance Computer Architecture*, Jan. 2002.

[11] A. Bracy, P. Prahlad, and A. Roth. Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. In *Proc. 37th Intl. Symp. on Microarchitecture*, pages 18–29, Dec. 2004.

[12] H. Cain and M. Lipasti. Memory Ordering: A Value Based Definition. In *Proc. 31st Intl. Symp. on Computer Architecture*, pages 90–101, Jun. 2004.

[13] B. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional Collection Classes. In *Proc. 12th Symp. on Principles and practice of parallel programming*, pages 56–67, Mar. 2007.

[14] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 Cache Misses with Checkpoint-Assisted Value Prediction. *IEEE Computer Architecture Letters*, 3(1), Dec. 2004.

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 278–289, Jun. 2007.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Architecture for Chip Multiprocessors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.

[17] T-F. Chen and J-L. Baer. A performance study of Software and Hardware Prefetching Techniques. In *Proc. 21st Intl. Symp. on Computer Architecture*, pages 223–232, Apr. 1994.

[18] T-F. Chen and J-L. Baer. Effective Hardware Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44:609–623, May. 1995.

[19] A. Cristal, O. Santana, M. Valero, and J. Martinez. Toward KILO-Instruction Processors. *ACM Trans. on Architecture and Code Optimization*, 1(4):389–417, Dec. 2004.

[20] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. 1997 Intl. Conf. on Supercomputing*, pages 68–75, Jun. 1997.

[21] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proc. 15th Intl. Symp. on High Performance Computer Architecture*, pages 7–17, Feb. 2009.

[22] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proc. 22nd IEEE Intl. Conf. on Computer Design*, pages 480–487, Oct. 2004.

[23] D. Ernst and T. Austin. Efficient Dynamic Scheduling Through Tag Elimination. In *Proc. 29th Intl. Symp on Computer Architecture*, pages 37–46, May 2002.

[24] D. Ernst, A. Hamel, and T. Austin. Cyclone: A Broadcast Free Dynamic Instruction Scheduler with Selective Replay. In *Proc. 30th Intl. Symp. on Computer Architecture*, pages 253–262, Jun. 2003.

[25] B. Fahs, T. Rafacz, S. Patel, and S. Lumetta. Continuous optimization. In *Proc. 32nd Intl Symp. on Computer Architecture*, pages 86–97, Jun. 2005.

[26] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 446–457, Jun. 2005.

[27] D. Gibson and D. Wood. Forwardflow: a scalable core for power-constrained cmps. In *Proc. 37th Intl. Symp. on Computer Architecture*, pages 14–25, Jun. 2010.

[28] A. Golander and S. Weiss. Hiding the Misprediction Penalty of a Resource-Efficient High-Performance Processor. *ACM Transactions on Architecture and Code Optimization*, 4(4), Jan. 2008.

[29] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, Jul. 2008.

[30] A. Hilton, N. Eswaran, and A. Roth. CPROB: Checkpoint Processing with Opportunistic Minimal Recovery. In *Proc. 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[31] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Pipelines. In *Proc. 15th Intl. Symp. on High Performance Computer Architecture*, pages 431–442, Feb. 2009.

[32] A. Hilton, S. Nagarakatte, and A. Roth. icfp: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30:12–19, 2010.

[33] A. Hilton and A. Roth. Ginger: Control Independence Using Tag Rewriting. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 436–447, Jun. 2007.

[34] A. Hilton and A. Roth. Decoupled Store Completion and Silent Deterministic Replay: Enabling Scalable Memory Systems for CPR/CFP Processors. In *Proc. 36th Intl. Symp. on Computer Architecture*, pages 245–254, Jun. 2009.

[35] A. Hilton and A. Roth. BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Processing. In *Proc. 16th Intl. Symp. on High Performance Computer Architecture*, Jan. 2010.

[36] A. Hilton and A. Roth. SMT-Directory: Efficient Enforcement of Load-Load Ordering for SMT. *Computer Architecture Letters*, 9(1), June 2010.

[37] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1), Feb. 2001.

[38] E. Jacobson, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proc. 29th Intl. Symp. on Microarchitecture*, pages 142–152, Dec. 1996.

[39] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proc. 24th Intl. Symp. on Computer Architecture*, pages 252–263, Jun. 1997.

[40] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 364–373, Jul. 1990.

[41] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *Proc. 31st Intl. Symp. on Microarchitecture*, pages 216–225, Dec. 1998.

[42] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed Early Load Retirement. In *Proc. 11th Intl. Symp. on High Performance Computer Architecture*, Feb. 2005.

[43] H. Le, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, and E. Schwartzand M. Vaden. POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, Nov. 2007.

[44] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. 29th Intl. Symp. on Computer Architecture*, pages 59–70, May 2002.

[45] H. Li, C-Y. Cher, T.N. Vijaykumar, and K. Roy. Vsv: L2-miss-driven variable supply-voltage scaling for low power. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 19–28, Dec. 2003.

[46] M. Lipasti, B. Mestan, and E. Gunadi. Physical Register Inlining. In *Proc. 31st Intl. Symp. on Computer Architecture*, pages 325–336, Jun. 2004.

[47] A. Martin, M. Nystroem, and P. Penzes. ET2: A Metric for Time and Energy Efficiency of Computation. Technical Report CSTR:2001.007, CalTech, 2001.

[48] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proc. 35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[49] P. Michaud. A PPM-like, Tag-Based Branch Predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, Apr. 2005.

[50] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation Through Virtual-Physical Registers. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 186–192, Nov. 1999.

[51] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-Based Transactional Memory. In *Proc. 12th Intl. Symp. on High-Performance Computer Architecture*, Jan. 2006.

[52] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. In *Proc. 26th Intl. Symp. on Microarchitecture*, pages 202–213, Dec. 1993.

[53] O. Mutlu, H. Kim, and Y. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 370–381, Jun. 2005.

[54] 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. 9th Intl. Symp. on High Performance Computer Architecture*, pages 129–140, Feb. 2003.

[55] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. A Simple Latency Tolerant Processor. In *Proc. 26th Intl. Conf. on Computer Design*, Oct. 2008.

[56] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proc. 24th Intl. Symp. on Computer Architecture*, pages 206–218, Jul. 1997.

[57] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 411–422, Dec. 2003.

[58] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, D. Jimenez, and M. Valero. A Flexible Heterogeneous Multi-Core Architecture. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2007.

[59] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, A. Veidenbaum, D. Jimenez, and M. Valero. A Two-Level Load/Store Queue Based on Execution Locality. In *Proc. 35th Intl. Symp. on Computer Architecture*, Jun. 2008.

[60] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. 12th Intl. Symp. on High Performance Computer Architecture*, pages 53–64, Feb. 2006.

[61] V. Petric, A. Bracy, and A. Roth. Three Extensions to Register Integration. In *Proc. 35th Intl. Symp. on Microarchitecture*, pages 37–47, Nov. 2002.

[62] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 98–109, Jun. 2005.

[63] S. Raasch, N. Binkert, and S. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In *Proc. 29th Intl. Symp. on Computer Architecture*, May 2002.

[64] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. 34th Intl. Symp. on Microarchitecture*, pages 294–305, Dec. 2001.

[65] E. Rotenberg, Q. Jacobsen, and J. Smith. A Study of Control Independence in Superscalar Processors. In *Proc. 5th Intl. Symp. on High Performance Computer Architecture*, pages 115–124, Jan. 1999.

[66] E. Rotenberg, Q. Jacobson, and J. Smith. A Study of Control Independence in Superscalar Processors. Technical Report CS-TR-98-1389, University of Wisconsin-Madison, Nov. 1998.

[67] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 458–468, Jun. 2005.

[68] A. Roth. Store Vulnerability Window (SVW): A Filter and Potential Replacement for Load Re-Execution. *Journal of Instruction Level Parallelism*, 8, 2006. (http://www.jilp.org/vol8/).

[69] A. Roth. Physical Register Reference Counting. *Computer Architecture Letters*, 7(1), Jan. 2008.

[70] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[71] A. Roth and G. Sohi. Effective Jump Pointer Prefetching for Linked Data Structures. In *Proc. 26th Intl. Symp. on Computer Architecture*, pages 111–121, May 1999.

[72] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 399–410, Dec. 2003.

[73] S. Sethumadhavan, F. Roesner, J. Emer, D. Burger, and S. Keckler. Late-Binding: Enabling Unordered Load-Store Queues. In *Proc. 34th Int'l Symposium on Computer Architecture*, pages 347–357, Jun. 2007.

[74] T. Sha, M. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *Proc. 38th Intl. Symp. on Microarchitecture*, pages 159–170, Nov. 2005.

[75] T. Sha, M. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. 39th Intl. Symp. on Microarchitecture*, pages 285–296, Dec. 2006.

[76] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.

[77] J. Stark, M. Brown, and Y. Patt. On Pipelining Dynamic Instruction Scheduling Logic. In *Proc. 33rd Intl. Symp. on Microarchitecture*, Dec. 2000.

[78] S. Subramaniam and G. Loh. Fire and Forget: Load/Store Scheduling with No Store Queue at All. In *Proc. 39th Intl. Symp. on Microarchitecture*, Dec. 2006.

[79] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs Technical Report, Jun. 2006.

[80] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 469–480, Jun. 2005.

[81] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 392–403, Jun. 1995.

[82] T. Wenisch, A. Ailmaki, and B. Falsafi. Mechanisms for Store-wait–free Multiprocessors. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 266–277, Jun. 2007.

[83] C-L. Yang and A. Lebeck. Push vs. Pull. In *Proc. 2000 Intl. Conf. on Supercomputing*, May 2000.

[84] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load-Related Instruction Scheduling. In *Proc. 26th Intl. Symp. on Computer Architecture*, pages 42–53, May 1999.