Power-Efficient Loop Execution Techniques

by

Mitchell Bryan Hayenga

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

This thesis is dedicated to Aishwarya, my loving and patient wife.

ACKNOWLEDGMENTS

Without the help of friends, family, and professors, this thesis work would not have been possible. It is to these people I owe my deepest gratitude.

I would first like to thank my advisor, Mikko Lipasti, for his instruction and guidance throughout my graduate school career. His passion to explore new ideas as well as his vast knowledge on a wide variety of topics is truly inspiring. I am thankful for having been given the opportunity to be a member of his lab.

I would also like to thank Kewal Saluja, Michael Schulte, and Charles Fischer. Each of these professors care greatly about teaching and benefited me greatly through their courses. Additionally, I owe their sound advice for keeping me on the right track in graduate school.

My fellow graduate students have also played a great role in motivating and assisting with this thesis work. I would like to thank my PHARM lab elders Natalie Enright Jerger, Eric Hill, Erika Gunadi, and Dana Vantrease for welcoming me into their lab. I would also like to thank Jacob Adriaens, Chris Jenkins, Kyle Rupnow, and Daniel Chang for their fellowship during the early portion of my graduate life. Finally, I would like to express my thanks to Andrew Nere, Syed Gilani, Arslan Zulfiqar, David Palframan, Vignyan Reddy, Dibakar Gope, Tony Gregerson, Amin Farmahini-Farahani, Jie Liu, and Vinod Reddy.

Finally, I would like to thank my wife Aishwarya Nagarajan. Without her love and support, it would have taken me more time to finish this work. She was also, undoubtedly, the greatest discovery I made during my doctorate. I look forward to moving on and experiencing the next chapter of my life with her.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation is motivated the growing issue of power consumption in modern processors. Historically, the advent of new process technology resulted in improvements in transistor densities, functional latencies, and operational voltages. However, improvements in voltage scaling have stagnated, forcing architects to predominately focus on improving processor energy-efficiency. To mitigate this energy problem, this thesis proposes techniques to streamline processor activity during the execution of program-based loops.

During program execution on modern out-of-order processors, substantial power is consumed by the processor front-end. Additionally, the majority of executed instructions are contained within simple loop bodies. We propose modifying modern processors to enable "in-place execution" of loops within the out-of-order back-end. Essentially, a few static instances of each loop instruction are dispatched to the out-of-order execution core by the processor front-end. The static instruction instances may each be executed multiple times in order to complete all necessary loop iterations. During loop execution the processor front-end, including fetch, branch prediction, decode, allocation, and dispatch logic are completely clock gated to save energy.

Multiple techniques and structural modifications are performed in order to enable in-place execution. To determine the efficacy of moving

loop execution into the processor back-end, we evaluate the necessary modifications and benefits of in-place loop execution on traditional and non-traditional out-of-order processor substrates. Additionally, we propose a performance enhancing load pre-execution mechanism that improves performance during loop execution. Finally, we evaluate multiple alternative interconnection networks for the non-traditional out-of-order processor.

Overall we find in-place loop execution to be an attractive microchitectural technique that results in a 5.3%-18.3% energy-delay benefit. This benefit is observed through the successful elimination of 20, 55, and 84% of all front-end instruction dispatches on the SPEC CPU2006, MiBench, and SD-VBS benchmark suites.

1 INTRODUCTION

Although transistor densities continue to scale, the associated per-transistor energy benefit normally obtained from successive process generations is rapidly disappearing. This phenomenon, known as the end of Dennard scaling, forces architects to limit transistor switching by means of structural optimization, functional specialization, or clock regulation [22, 25]. Furthermore, the need for improved computational efficiency has been highlighted by increased demand in the power conscious mobile and server markets.

To cope with these increasing energy constraints, future out-of-order processors must further streamline common execution patterns, thereby

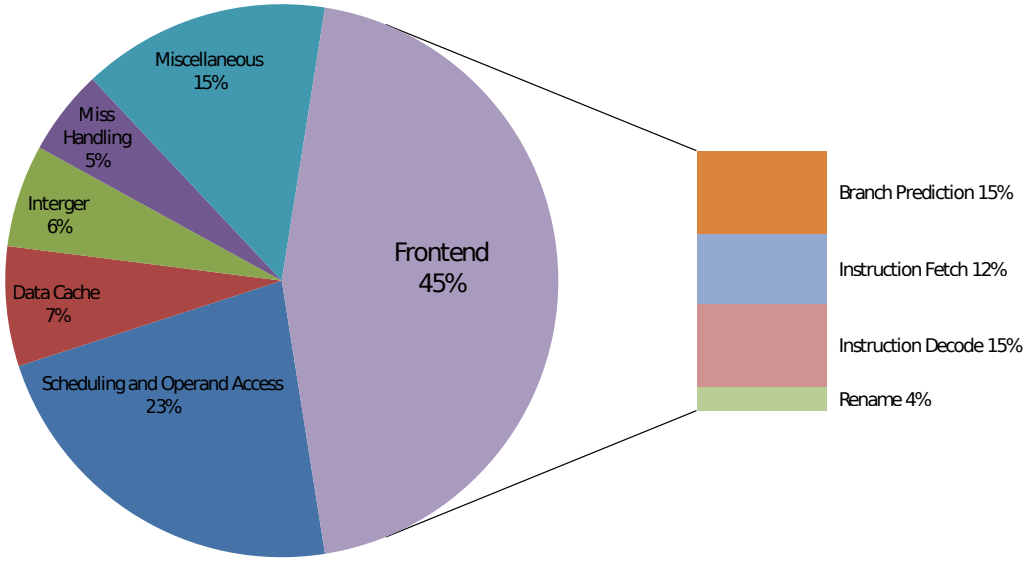


Figure 1.1: ARM Cortex-A15 Energy Consumption.

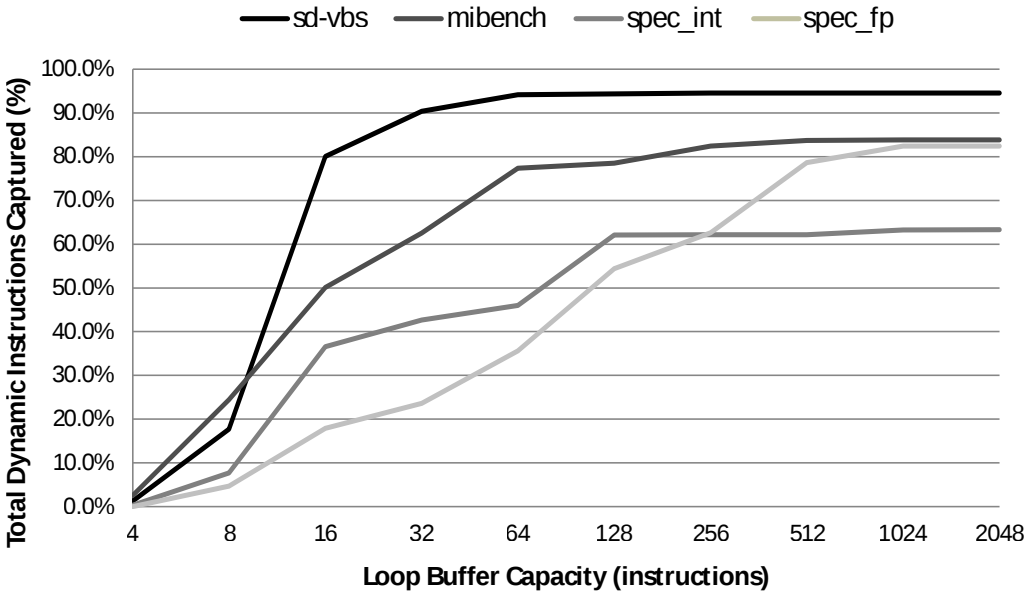


Figure 1.2: Ideal Loop Buffer Performance.

eliminating unnecessary pipeline activity. For common applications on modern processors, the energy required by instruction execution is relatively small. Instead these applications expend the majority of energy on control overheads, such as instruction fetch and scheduling [78]. This energy distribution is particularly visible in recent mobile processors, such as ARM's Cortex-A15 [52]. Shown in Figure 1.1, the Cortex-A15 processor only expends 6% of all power on integer execution, while the front-end is capable of consuming 45% of all core energy [60]. Therefore microarchitectural optimizations to lessen front-end activity can have significant impacts on overall power consumption.

To reduce these front-end energy overheads, architects have proposed many pipeline-centric instruction caching mechanisms that capitalize on

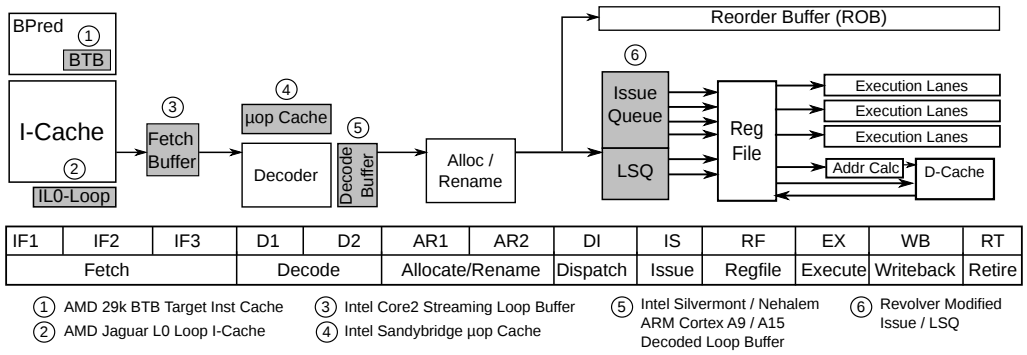


Figure 1.3: Instruction Reuse caching Methods.

temporal instruction locality [3, 11, 38, 49, 54]. The majority of these proposals target capturing loop instructions, in a decoded or encoded form, into a small buffer for inexpensive retreival on future iterations. Figure 1.2 shows the percentage of instruction accesses that can be serviced from loop buffers of varying sizes across multiple benchmark suites. As seen, loop buffers can be quite effective, with a common loop buffer size of 32 instructions able to capture 24-90% of all instruction accesses.

The observed effectiveness of loop buffers has resulted in their rapid industrial adoption [4, 26, 46, 52, 72]. Shown in Figure 1.3, industry has progressively implemented more deeply embedded loop buffers in processor pipelines as a means to reduce front-end activity. Notably, this trend has created the ability to bypass not only fetch, but decode overheads as well. Although early designs, like AMD's 29k, only eliminated instruction cache accesses and supported very small loops, more modern designs from Intel and ARM bypass more pipeline stages and store significantly

larger loops.

However, despite potential energy savings, no commercial out-of-order processor has attempted to bypass all front-end pipeline stages, including allocation and dispatch, during loop execution. This boundary has not been pushed because of obstacles related to program ordering, dependence linking, and resource allocation.

In this thesis we propose the Revolver architectures, aggressive out-of-order processors which obviate the complexities related to moving loop buffering into the processor back-end. During loop execution, only a few static instances of each loop instruction are dispatched to the out-of-order back-end. After which, each instruction instance may be executed in-place multiple times in order to complete all necessary loop iterations. During subsequent executions, no additional resources are allocated and no front-end structures are accessed. This operation is enabled through a series of insights and modifications to the traditional processor pipeline. We implement Revolver's in-place loop execution on two out-of-order architectures: A traditional physical register file (PRF) based out-of-order and the CRIB [32] out-of-order architecture. Additionally we propose a mechanism which enables the pre-execution of future loop iteration loads, effectively increasing memory level parallelism and enabling single cycle loads in many instances on future loop iterations. Finally, we evaluate multiple interconnection networks to better enable loop-mode execution on the CRIB-based architecture.

1.1 Revolver Architectures

To evaluate the effectiveness of in-place loop execution within a processor's out of order back-end, we propose extensions to two different out-of-order architectures: A traditional PRF-based out-of-order architecture and the CRIB architecture. These modified architectures are collectively referred to as Revolver architectures, due to the cyclical nature of their in-place loop execution.

Traditional Out-of-Order Based

Physical register file based out-of-order architectures are a predominant design within microprocessor industry. The overall structure and operation of such out-of-orders remains largely unchanged from earlier designs such as the MIPS R10000 [91].

Shown at the base of Figure 1.3, is a conventional pipeline structure for such out-of-orders. In general, the pipeline is composed of an in-order front-end, out-of-order execution, and in-order commit. To adapt such an architecture for in-place loop execution, three primary modifications are required: the addition of loop detection within the processor front-end, specialized resource allocation and dependency linking logic, and customized instruction issue logic.

During non-loop execution mode, instructions flow through the entire processor pipeline as in a conventional out-of-order core. The key struc-

tural difference between Revolver and a traditional out-of-order core is the lack of a register allocation table (RAT) within the processor front-end. Instead, dependence linking between instructions is performed in the processor back-end by a simple structure called the Tag Propagation Unit (TPU) that is accessed in parallel with issue select. Other than this structural modification, which is detailed further in Chapter 4, the Revolver back-end operates like a normal out-of-order processor during non-loop mode.

To enable loop mode, additional loop detection logic is placed at decode. Once a loop's starting address and number of required resources[1] have been calculated, a subsequent decoding of the first loop instruction initiates loop mode. During loop mode, the loop body is unrolled as many times as allowed by the resources present within the out-of-order back-end. Fundamental to Revolver's operation is its ability to eliminate the need for any additional resource allocation once a loop has been dispatched. With respect to the front-end, allocation of most resources proceeds normally. However, allocation of destination registers requires special handling.

With respect to Revolver's back-end, the primary innovation is the ability to allow loop instructions to maintain their provided resources across multiple executions. Instructions retain issue queue entries after issue select and reuse them immediately for the next loop iteration upon commit. The load/store queue is also modified to enable reuse of entries

[1]Resources being physical registers as well as issue queue, load queue, and store queue entries.

while properly maintaining program order. Finally, as detailed in Chapter 4, each result-producing loop instruction simply alternates writing one of two pre-allocated physical registers. Revolver's TPU is designed to allow dependent instructions to properly access source registers even with alternating register dependencies.

On loop exit, all instructions are removed from the out-of-order back-end and the loop fall through path is immediately dispatched. Immediate dispatch is possible since, before clock-gating, the processor front-end redirects to the fall through path after successful loop dispatch.

CRIB-Based Revolver

The CRIB architecture, which stands for Consolidating Rename, Issue, and Bypass, was introduced in [32] by Gunadi et al. This architecture represents a novel out-of-order design that eschews traditional register renaming and issue logic for spacial dataflow and distributed logic whereby each instruction is allocated an ALU and self-schedules. The primary motivation behind the CRIB architecture is that traditional out-of-order processors originated from a resource-constrained era where power consumption was a secondary design constraint. Thus CRIB is a fundamental redesign of a traditional out-of-order architecture with power efficiency, rather than resource utilization, as its primary objective.

The CRIB architecture was selected as a baseline for in-place execution due to its focus on power efficiency as well as its suitability for modification

to support in-place loop execution. To support in-place loop execution on the CRIB design, the primary modifications are the addition of loop detection logic and a redesign of the load-store queue. Full details of modifications are presented in Chapter 6. Chapter 7 investigates the potential benefit of alternative operand networks.

Load Pre-Execution

For both baseline architectures, we evaluate a novel extension that enables the pre-execution of future loop iteration loads during loop exection. This performance optimization originates from the observation that static load instructions within loops frequently exhibit predictable access patterns. Once a supported access pattern is detected, the load is triggered for pre-execution. Essentially, a future instance of the load is speculatively performed and the result is stored in a special buffer. Later, once the load is actually issued, it executes within a single cycle and returns the previously buffered value. If the load reads from a different address, single cycle operation is aborted and the load is re-issued normally. Load pre-execution enables performance beyond traditional prefetching, as it is capable of hiding even the latencies of L1 caches. Full details are presented within Chapter 5.

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Amortizing the energy from fetch, decoding, allocation, and dispatch of a single instruction across multiple loop iterations:** As single static instructions are dynamically re-executed multiple times in the out-of-order back-end, many front-end energy costs relating to allocation and instruction routing are eliminated.

- **Moving operand dependence linking into the out-of-order back-end:** Traditional out-of-orders establish program dependencies between instructions within the in-order processor front-end. To facilitate loop-mode operation and register reuse, Revolver architectures introduce novel dependence linking within the out-of-order back-end.

- **Eliminating the need to re-allocate resources between instruction re-executions:** Through allocation policies and structural modifications, in-place execution is enabled in such a manner that load queue, store queue, issue queue, and physical register can be reused through multiple iterations' executions of loop instructions.

- **Reducing the branch misprediction penalty for variable iteration count loops:** After completing loop dispatch, the processor front-end immediately redirects to the loop's predicted fall through path.

The loop end is effectively no longer predicted. This enables loops with unpredictable or variable iteration counts to immediately redirect to the fall through path upon loop exit.

- **Enabling the pre-execution of loads from future loop iterations:** Through the observation that most static loads within loops follow regular and predictable access patterns, we develop a pre-execution method that effectively enables single-cycle load execution in many instances.

- **Evaluating the bypass requirements and potential benefits of alternative interconnection substracts for the loop-enabled and baseline CRIB processor:** We evaluate the impact of interconnection network latency and bypass connectivity for enabling loop-mode execution within our CRIB-based design.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 provides a detailed overview of prior work on loop optimizations within processors, the baseline CRIB out-of-order architecture, and processor-based operand networks. Chapter 3 details how loop detection and dispatch are handled within modern processors. Chapter 4 details our implementation if in-place loop execution on a traditional out-of-order processor architecture. Chapter 5 introduces

our novel load pre-execution method and the structural modifications required to take advantage of it. Chapter 6 details the implementation of in-place loop execution on the baseline CRIB architecture. Chapter 7 evaluates the potential benefit of alternative operand networks for loop-mode execution on the CRIB-based architecture. Chapter 8 contains the evaluation methodology and results for all thesis contributions. Finally, Chapter 9 concludes the thesis and summarizes key observations and results.

2 BACKGROUND

This chapter explores related work and background material that is central to this thesis. Background material is divided into five primary sections. Section 2.1 covers previous loop buffering methods explored by academia and industry. Section 2.2 details multiple prefetching techniques and is provided as background for our load pre-execution technique. Section 2.3 surveys the development and current status of out-of-order processing cores. Section 2.4 provides an in-depth description of the CRIB microarchitecture, which is used as one of our baseline designs. Finally, section 2.5 provides an overview of operand networks.

2.1 Loop Caching and Buffering

The "von Neumann bottleneck", specifically the separation of instruction memory from processing elements, is often cited as the key limiter of performance and power efficiency in processor architecture [8, 89]. In modern out-of-order processors, loop bodies are speculatively unrolled in hardware to extract parallelism. Due to the temporal and spacial locality of instructions during loop execution, architects have devised multiple methods to make loop instruction retrieval fast and inexpensive.

The methods proposed by architects span a wide variety of designs, from general purpose to deeply embedded and highly specialized loop

caching methods. The remainder of this section details these methods, starting with general purpose and progressing to the more advanced techniques.

General Purpose Caching

It is widely known that most program execution patterns follow the Pareto Principle, where the majority of execution time is expended within a small subset of code [88]. Following this principle and the results from Figure 1.2, it is evident that the majority of execution is spent within compact loop bodies.

Introduced by Kin et al. [49], the Filter Cache is a general purpose cache aimed at minimizing instruction fetch energy. Effectively the Filter Cache serves as an L0 instruction cache, where the energies and latencies of instruction accesses are less than a common L1 cache size. Filter Caches effectively add an additional level of cache hierarchy to conventional processing systems. Additional system complexity is justified through the high hit rate and low access cost of such caches. Qualcomm's Krait CPU architecture is a modern example of a design that utilizes a Filter Cache [65]. The Krait CPU architecture utilizes a 4KB direct mapped L0 instruction cache in conjunction with a traditional 16KB, 4-way associative L1 cache. As most loops are small in size, Filter Caches successfully capture most loop bodies at significantly less cost than a traditional L1 cache. It should be noted that the original Filter Cache design primarily targeted instruc-

tion fetch energy costs while negatively impacting performance when instructions were sourced from the L1 cache. Later Filter Cache proposals introduced speculative loading techniques to lessen the frequency of Filter Cache misses [83].

AMD's 29000 microprocessor provides another historical example of instruction caching that successfully reduces the latency and energy consumption of cache access [26]. This processor utilizes a customized Branch Target Buffer (BTB) that, in addition to supplying a branch target address, provides up to the next four instructions. The customized BTB was added as a means to eliminate pipeline stalls due to long cache access latencies on taken branches. The byproduct of such a design is that for codes with frequently taken jumps, it is possible to completely execute out of the BTB without ever accessing the L1 cache. Thus, for small or branchy loops, the 29000's BTB effectively resulted in great energy and performance gains.

Amongst the earliest of designs utilizing caching of instructions for the purpose of loop reuse are the CDC-6600 and CRAY-1 [53]. These machines maintained small, contiguous buffers of program instructions for the purpose of loop caching[1]

[1]We do not consider these to be general "loop buffers" due to the requirement that all instructions be sequential.

Pre-Decode Loop Buffers

Although general purpose caching techniques provide some benefits for loop execution, significantly greater energy gains can be obtained by designing specialized instruction buffers that explicitly target loops. Specialized loop buffers provide significant energy and performance gains over caches by utilizing significantly smaller SRAM structures and frequently eliminating the energy costs of address tag verification.

Loop buffers as an instruction caching technique have a long history within both industry and academia. Shown in Figure 1.3, loop buffers are placed alongside or after the L1 instruction cache. Once a loop is successfully buffered, all instruction accesses are serviced from this buffer until the loop exits or an unpredicted execution path is followed.

Initially utilized within the embedded area, Lucent Technologies' DSP16000 is the first industry example of a loop buffer [3]. The DSP16000, as an embedded processor, relied upon programmer specified loop-caching instructions. These loop-caching instructions allowed DSP programmers to explicitly identify loop bodies and initiate control circuitry's buffering of given loops. Once loop bodies were fully captured, loops could iterate up to 65,535 times without any dispatch overheads.

In the realm of general purpose processors, multiple concurrent academic efforts for fully automated loop buffers exist. Fully automated loop buffers rely on dynamic loop detection and buffering without the necessity of programmer specification. In [11], Bellas et al. describe a loop cache

as a logical extension of Filter Caches. In this design the Filter Cache is replaced by a customized buffer for holding loop bodies. Additionally, hardware/software codesign is utilized to customize loop size and code placement for effective loop cache utilization. Lee et al. [54], expand upon the work of [11] by explicitly defining the mechanism that loop bodies are dynamically detected and placed into loop buffers.

Within industry, loop buffers placed before decode are widely used and present within many designs. Examples of designs containing pre-decode loop buffers are ARM's Cortex A9, Intel's Core 2, and AMD's Jaguar [6, 4, 72]. The A9 and Core 2 are more specialized than the AMD Jaguar, capable of buffering 16-32 general instructions. The AMD Jaguar loop buffer acts as a subset of the L1 cache, effectively buffering instructions from up to two 64 byte cache lines.

Post-Decode Loop Buffers

To eliminate additional front-end power architects have designed even further embedded loop buffers. Placing loop buffers after decode enables architects to eliminate the energy resulting from more pipeline stages as well as eliminating repetitive work done by decode. Due to their post-decode placement, structure and instruction retrieval differs significantly from pre-decode loop buffers. Primarily, these buffers provide μOps instead instructions and bundle instructions across predicted branch boundaries. A side effect of these methods is that the buffers must be flushed

and reloaded on any branch misprediction. However, loop detection and buffering circuitry is identical to that of pre-decode loop buffers.

Decoded loop buffers were initially proposed by Bajwa and Hiraki [10, 38]. Referred to as the Decoded Instruction Buffer (DIB), outside of loop execution instructions simply flow from the decoder to the execution back-end. After a loop has been identified, during the first iteration, decoded instructions are queued into the DIB at the rate of dispatch. For all subsequent executions the DIB provides all decoded instructions. Testing across DSP and RISC processors showed the DIB capable of saving up to 40% of dynamic energy.

Within industry, the majority of recently introduced high performance microprocessors support decoded loop buffers. Predominate examples include the Intel Nehalem, Intel Silvermont, and ARM Cortex A15 microprocessors [46, 52, 72]. With the Intel Sandybridge [45] processor, Intel introduced a μop cache instead of a loop buffer. μop caches tradeoff some of the power efficiency of loop caches in exchange for capturing more instructions and behaviors. Thus codes which frequent and simple loops may be better served by a traditional loop cache, however μop caches are more robust and able to derive benefit more irregular codes. Essentially, μop caches operate as traditional caches which hold decoded instructions. However, they share some characteristics with loop caches. In current commercial implementations, μop caches encode predicted branch paths. If branch paths differ from previously predicted paths, like loop caches,

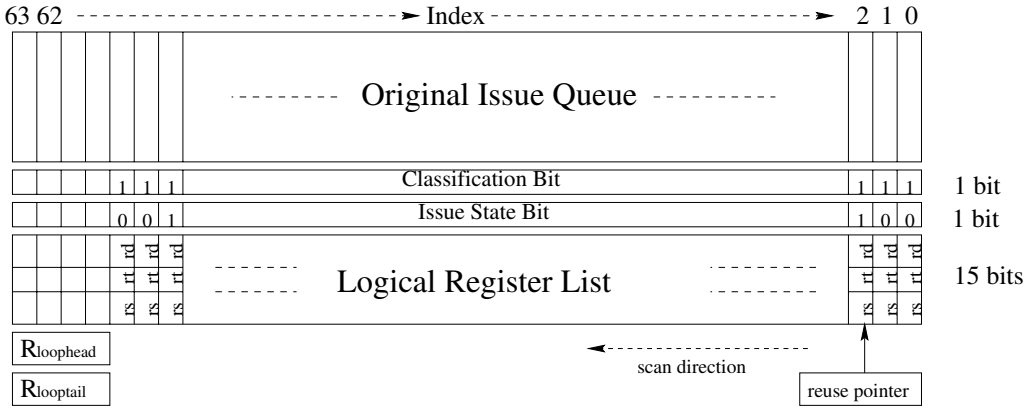


Figure 2.1: Issue Queue Design from [40].

the µop cache must be flushed and refilled.

Back-end Loop Buffering

Although no commercial attempt has been made to buffer loops within an out-of-order processor's execution back-end, multiple academic proposals have been made to perform such an operation [40, 63, 90]. However, it should be noted that unlike our Revolver architectures, all previous designs require front-end communication for resource allocation and renaming.

Of all proposals, the earliest work by Hu et al. [40], is the most similar to our proposed in-place execution method. This design works by dynamically detecting loops and buffering them within the processor's issue queue. To achieve this, significant modifications to the issue queue and resource allocation logic are required. Recreated from [40], Figure 2.1 shows the modified issue queue for buffering loops within the processor

back-end. Essentially, loops of suitable size are placed within the issue queue and will be executed multiple times in order to satisfy all necessary loop iterations. To enable the dynamic reuse of issue queue entries, [40] utilizes additional pointers and information storage. The two additional pointers, *Rloophead* and *Rlooptail*, enable the commit pointer to wrap upon loop commit. Additionally, in addition to physical renaming information, each issue queue entry holds the instruction's logical register identifiers. As issue queue entries execute, they retain their entries until commit. Upon commit, if the finished instruction belonged to a loop, the logical register identifiers would be used to communicate with the front-end renaming and allocation hardware. Using the logical register identifiers, new physical register identifiers are obtained and the issue queue entry is reused for the next iteration's instance of the given instruction. It is unspecified in [40] how loads and stores are handled during loop execution.

In [90], rather than sourcing all instructions from decode or a post-decode buffer, the reorder buffer (ROB) is used as an instruction buffering resource. Rather than fetching and decoding all instructions, the ROB is speculatively searched to identify prior occurrences of newly required instructions. To enable this, the ROB is scanned for instructions matching those currently being fetched. Upon locating a match, the contents of the corresponding ROB entry are copied to a new entry. However, traditional resource allocation and renaming must take place for physical registers and load/store queue entries. This optimization primarily works to save fetch

and decode energy relating to instruction opcodes and control. However the feasibility of such a design is unclear given the design complexity and power consumed in order to dynamically detect instruction reuse in such a way.

Finally in terms of buffering instructions within the out-of-order back-end, Pratas et al. [63] propose an alternative method to reuse reorder buffer entries during loop execution. In this design loops are dynamically detected and ROB entries for a given loop body are cyclically reused to execute all all necessary iterations. An additional structure is appended to track the physical register instances of all loop instructions. Unlike [40, 90] and our own work, [63] disallows dynamically unrolling loops in hardware. Instead physical resources are held until loop commit for all loop instructions. Upon loop commit, resources are freed and registers for the next iteration are allocated. Thus no speculative register allocation or parallelism across loop iterations are allowed. [63] notes no significant impact on performance due to this limitation, however our own results presented in Chapters 4 and 6 significantly differ.

Miscellaneous Loop Techniques

In addition to techniques that buffer loop instructions within the processor pipeline, other optimizations for bettering loop performance have been proposed.

VEAL, proposed by Clark et al. [20], is a special purpose accelerator

that targets frequently executed loops. In VEAL, loops are dynamically detected at runtime through a just-in-time (JIT) compilation process. During the JIT process, suitable loops are translated for later offloading to a special purpose accelerator. This custom accelerator breaks loops into memory access streams and computation, allowing loads to be streamed far in advance of execution without the complexity of traditional out-of-order processing cores. Through this translation and offloading, great performance and energy gains are demonstrated on a variety of application codes in comparison to a conventional dual-issue in-order processor.

Wish Loops, proposed by Kim et al. [48], represent an attempt to dynamically predicate extraneous loop iterations. In the event of loops with variable trip counts, Wish Loops enable the processor to dynamically predicate extra iterations without having to perform a full squash operation. By buffering loops in-place in our Revolver architectures, we obtain similar benefits on variable iteration loops as the loop termination is no longer predicted and the fall through path is immediately available on loop exit.

2.2 Prefetching Techniques

As processor performance outpaces improvements in memory technology, the fraction of time modern processors remain stalled awaiting in memory data has dramatically increased [89]. Additionally, with the advent of chip multiprocessors (CMPs), contention and latency for main memory are

only likely to worsen [61]. To obviate the disparity between processor and memory performance, architects have investigated many techniques to address the growing problem of data latency. While caches and memory hierarchies greatly reduce the latency of spatial and temporal data reuse, prefetching techniques serve the purpose of reducing or eliminating the penalty of compulsory cache misses [37].

In this thesis we introduce a new microarchitectural technique known as Load Pre-Execution. Load Pre-Execution has many similarities to traditional prefetching, however extends the prefetching benefit beyond the systems caches and into the processor execution core. In this section we survey the history and state of data prefetching mechanisms.

Cache Prefetching

Prefetching within the memory system has long been viewed as a simple and straightforward means to increase system performance. Early cache designs noted the benefit from spatially prefetching multiple contiguous words from memory [5]. Prefetching of adjacent cache lines was later implemented in the commercial IBM 370/168 and Amdhal 470V systems [74].

Sequential Prefetching

Taking advantage of spatial locality, multiple methods exist to perform sequential prefetching of cache lines. In sequential prefetch methods,

during the processing of a current cache line, the fetching of additional, contiguous cache lines are speculatively performed.

The simplest form of sequential prefetch is one block lookahead (OBL). In OBL schemes, cache line $b + 1$ is speculatively prefetched during the processing of cache line b. In [75], Smith defines multiple implementations of OBL prefetching. In the *prefetch-on-miss* implementation, speculative prefetchs for contiguous cache lines are only generated on cache misses. This results in simple hardware that initiates two cache line fetches on every cache miss. However, *prefetch-on-miss* does not prevent demand misses to contiguous cache lines of currently cached data. *Tagged prefetch* exists as an extension of *prefetch-on-miss*. In *tagged prefetching*, every cacheblock has an associated bit that identifies the given block as demand fetched or prefetched. In addition to generating sequential prefetches on cache misses, *tagged prefetching* generates prefetches for contiguous lines whenever prefetched cache lines are accessed. For streaming access patterns, *tagged prefetching* has the potential to eliminate almost all cache misses whereas the *prefetch-on-miss* technique would only eliminate approximately half of all demand cache misses.

Although sequential prefetch recognizes a common memory access pattern, OBL prefetching may not generate cache accesses in a timely manner in order to completely hide main memory latency. To combat the issue of prefetch timeliness, architects have investigated increasing the distance and number of prefetches generated on cache access. Known as the degree

of prefetch, a cache of degree N will intiate the prefetch of N sequential cache lines from memory at a time. It should be noted that OBL is a special case where the prefetch degree N equals one. In [64], Przybylski noted that prefetching degree beyond $N = 1$ frequently lead to performance degradations and excessive power consumption due to cache pollution. To address this issue, *adaptive sequential prefetching* was proposed to only initiate high degree prefetches when profitable [21]. In *adaptive sequential prefetch*, the utility of prefetches are periodically measured and impact the allowed degree of prefetching. When prefetched cache lines utilized the allowed degree of prefetching is increased. In the event that prefetched cache lines go utilized, the allowed prefetching degree is reduced and prefetching may even be disabled entirely. Such schemes can be found even in modern processors such as ARM's Cortex A15 [52].

To enable high degrees of prefetching, without adaptivity, Jouppi proposes the use of stream buffers [42]. Stream buffers exist as FIFO buffers alongside caches. By placing prefetched cache lines into a stream buffer instead of directly into a cache, potential cache pollution is avoided even with high degrees of prefetch.

Stride and Table-based Prefetching

In addition to sequential prefetching, architects have designed prefetches to identify and service regular memory access patterns that exhibit stride-based or otherwise predictable memory access patterns.

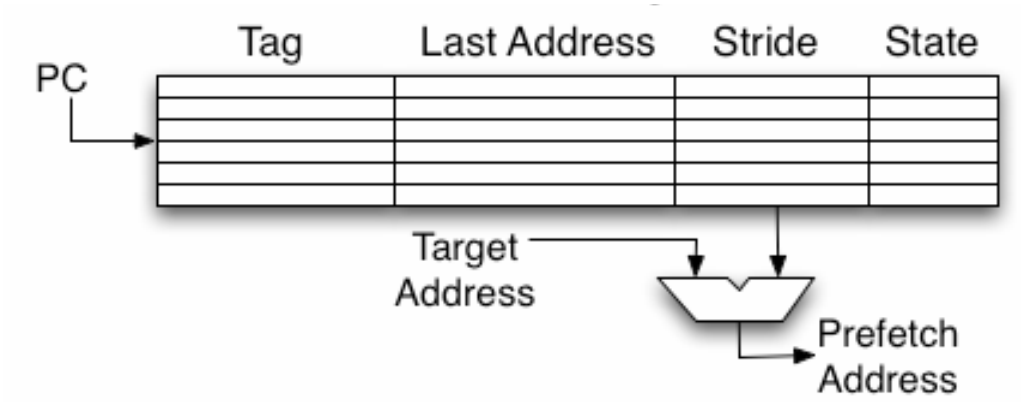


Figure 2.2: Stride Prefetching Reference Prediction Table (RPT).

In [9] Baer and Chen introduce what has become known as conventional stride-based prefetching. The predictor in [9] identifies memory patterns where static load or store instructions within loops exhibit a fixed-sized stride through memory over the course of execution. This is achieved by looking up a given load/store address in the Reference Prediction Table (RPT). Shown in Figure 2.2, the RPT is indexed by the load/store instruction address and contains fields for the last address accessed, currently predicted stride, and state of the given RPT entry. For a given load/store the prediction table can be in one of three primary states: *untrained*, *training*, or *trained*. Initially *untrained*, on the initial access by a memory instructions program counter, the address is recorded in the corresponding RPT entry and the state transitions to training. On subsequent accesses, the address-delta between prior and current memory accesses are computed and stored wthin the RPT. If the previous access plus the predicted stride match the current stride, the entry transitions

to the *trained* state. On future access, the RPT is used to speculatively prefetch ahead by the predicted stride.

Markov prefetching [41] is a type of correlated prefetching. In Markov prefetching, a history table records consecutive address pairs. In the event of a cache miss, the history table is probed using the miss address. Each entry in the Markov correlation table holds a list of addresses previously accessed following the current miss address. This address list is used as a source of potential prefetch targets.

Distance prefetching [43] is a generalization of Markov prefetching. In distance prefetching, the history table is accessed by the address delta between two consecutive global misses. Instead of identifying potential target addresses, the history table contains a list of previosuly observed address strides to use for the generation of prefetch addresses.

Finally, Nesbit et al [59] propose using a Global History Buffer to identify and initiate data cache prefetching. Using a linked-list structure, [59] demonstrates how the prefetch effectiveness of stride, markov, and distance prefetching can be obtained using an alternative structure.

Software Prefetching

In addition to hardware mechanisms designed to prefetch data from memory, architects have proposed mechanisms to enable programmer or compiler driven prefetch mechanisms [12, 17, 50].

For later reference it is improtant to note that software prefetch mech-

anisms exist in one of two forms: binding and non-binding. In binding prefetch mechanisms, prefetches are performed to a named register [30]. After access from memory the obtained value exists outside of conventional cache coherence mechanisms and will not be invalidated. Alternatively, non-binding software prefetch instructions load instructions from memory, likely resulting in their caching or otherwise buffering, but do not specifically name a destination register.

Microarchitectural Load Acceleration

Although most architectural prefetching mechanisms exist at the cache or software level, multiple microarchitctures effectively prefetch the required address stream of programs. In decoupled architectures, such as the Astronautics ZS-1 [77], two instruction streams for computation and memory access are processed in parallel. The independent memory access stream allows the queueing of required load data without the complexities of conventional out-of-order architectures, essentially resulting in single-cycle loads like our Load Pre-Execution technique.

Runahead execution [24] exists as a microarchtiectural mechanism that enables processor cores to generate data prefetches under the shadow of a cache miss. Runahead execution works by evaluating the predicted execution path with speculative values during a cache miss stall. As all work performed during the cache stall is discarded, the sole objective of runahead execution is to generate data prefetches. Although runa-

head execution has been used in many academic works, recent industry publications [16] have observed that well tuned conventional hardware and software prefetching can obtain almost all the benefits of runahead execution with few overheads.

Additionally, work to speculatively accelerate load address calculation within a processor core has been performed. With zero-cycle loads [7], fast address calculation, pre-decode information, and register caching are used in conjunction to eliminate load pipeline latencies relating to effective address calculation.

2.3 Out-of-Order Microarchitecture

Out-of-order execution has become ubiquitous within the microprocessor industry. Today out-of-order processors can be found in all market segments from deeply embedded mobile processors to high performance servers and workstations. This section provides an overview of the history of out-of-order processors as well as their current structural design.

Register Renaming

Foremost to enabling out-of-order execution is the ability to of a microarchitecture to eliminate false register dependencies while maintaining true data dependencies between instructions. Examples of false dependencies

include anti-dependencies[2] and output dependencies[3]. True dependencies are read-after-write (RAW) dependencies where results produced by one instruction are directly consumed by a later instruction. Historically, register renaming has served the purpose of eliminating false dependencies and linking operand dependencies between instructions. Register renaming was first introduced by Robert Tomasulo in what has become known as Tomasulo's algorithm [86].

Tomasulo's algorithm works by dividing a processor design into two halves: An in-order front-end and an out-order back-end. As instructions progress through the in-order front-end, their logical source identifiers are used to look up the corresponding physical locations of sources. The structure preforming this mapping of logical sources to physical destinations is commonly known as the register allocation table (RAT) in modern out-of-order processors. After having been allocated a storage resource in the out-of-order back-end, this mapping table is updated and the instruction is allowed to proceed to execution. The back-end storage resource for instructions awaiting dependencies for execution is commonly known as a reservation station. After allocation into the reservation station, instructions await for the broadcast of their source operands. In addition to data, source operands are broadcast with the associated physical identifier tag. The use of the physical identification tag enables dependent instructions only source true dependents. After all source dependencies are satisfied,

[2] Also known as write-after-read (WAR) dependencies.
[3] Also known as write-after-write (WAW) dependencies.

instructions proceed to execution. After execution, the result is broadcast on a common data bus (CDB) for dependent instructions to utilize.

Although Tomasulo's design embodies the central ideas of register renaming and out-of-order execution, much progress has been made in refining the structure and functionality of out-of-order processors. A primary functional extension to this is then enablement of precise interrupts and exceptions. In [79], Sohi and Vajapeyam introduce the Register Update Unit (RUU), to enable the elimination of false dependencies while maintaining precise microarchitectural state. With the RUU, age-based resolution of register results is performed by associating version numbers with each logical register. Through such ordering, instructions can execute in an out-of-order fashion while still accessing the appropriate version of source operands. The RUU operates as a combination of a reservation station and a reorder buffer (ROB). Instructions are maintained within the RUU until commit. At commit results are written in-order to the architectural register file. As results are written in an in-order manner after commit, designs utilizing an RUU are able to maintain a precise microarchitectural state.

Since [79], multiple different structural ways to maintain a precise architectural state have been proposed. In [76], Smith and Pleszkun detail the reorder buffer, history buffer, and future file methods of maintaining or restoring precise microarchitectural state in the presence of interrupts or exceptions. Over the years many refinements have been made, although

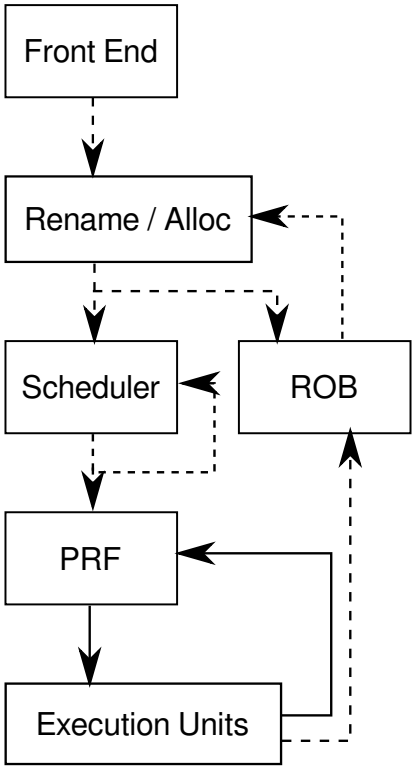


Figure 2.3: PRF-based Out-of-Order Structure.

the separate scheduler, physical register file (PRF), and reorder buffer approach is adopted by most current out-of-order processors. Figure 2.3 shows the high level structure of a modern out-of-order processor, such as Intel's Haswell microarchitecture [44].

Load/Store Ordering

In addition to maintaining order with respect to register operands, aggressive out-of-orders must maintain ordering with respect to memory across multiple instructions. Memory disambiguation consists of multi-

ple techniques to enable the simultaneous execution of multiple memory operations in an out-of-order fashion. The ordering of multiple operations with respect to memory, frequently referred to as a microprocessor's consistency model, directly impacts the structure and operation of the load/store unit [51, 1].

Although many alternative structures and techniques have been proposed, the load/store units are generally maintained as two separate queues: one for loads and one for stores [19, 58, 71, 82]. The load queue primarily contains a load's effective address and valid bits. The store queue additionally maintains a data buffer for written values. As memory operations enter the machine, each is assigned a monotonically increasing *store color*. Fundamental to the load and store queue is maintaining the order of operations between multiple operations to the same address. Upon load issue, loads associatively probe the store queue to find any older stores to the same address region being loaded. The store color is used to limit the subset of candidate stores. Essentially any store queue entry with an older store color is a candidate for data forwarding. Upon a data match, the value is forwarded from the youngest store, older than the given load. In the event of partial data and overlapping address regions, a merge operation between store queue entry data and cache data may be required. After load commit, the corresponding load queue entry is reclaimed and used for future instruction execution.

When stores issue, the load queue is searched to detect the situation

where a younger load to the same address region has executed. In this situation the load may have sourced stale data and must be triggered for re-execution. Otherwise, the store places its data in the store queue. Once the store commits, the store is considered *finished* or *senior* and may be written back to memory. Some architectures, such as PowerPC and ARM, utilize relaxed memory consistency models allowing multiple stores to be merged and written back simultaneously to memory in almost all cases. The structure which typically sits between a processor and its caches that performs this operation is a *store buffer*. Other architectures, like x86, more closely follow total store order (TSO), thus stores may not be combined in the presence of intervening loads. Eventually completed stores drain into memory and are removed from the store queue.

It should be noted that although associatively searched, load and store queues maintain a precise program-based ordering of memory operations. Thus the search logic must both maintain this ordering and be able to handle the wrapping of these circular queues.

2.4 CRIB Microarchitecture

For an alternative out-of-order substrate, this thesis evalutes the effectiveness of in-place loop execution on the CRIB architecture [32]. The CRIB architecture constitutes an effort to eliminate complexities related to register renaming, issue logic, and bypass networks in out of order

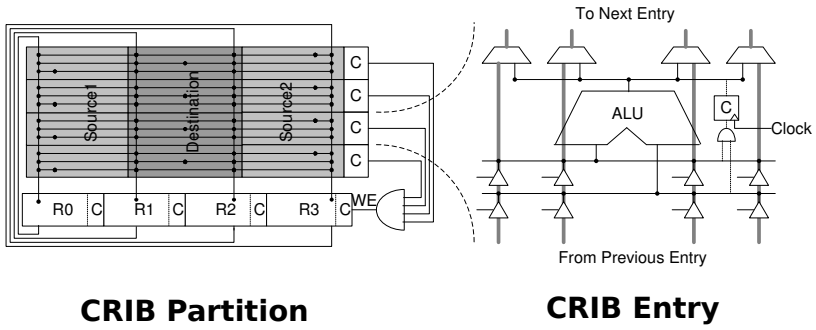


Figure 2.4: CRIB Structures.

architectures. Register renaming, although enabling elimination of false dependencies, introduces additional complexity, energy, and latency to the front end of modern processors. Additionally, modern out of order issue logic faces great challenges in terms of scalability and the generation of efficient schedules [14].

CRIB Design

CRIB achieves the objectives of eliminating renaming and issue logic complexity by eschewing traditional register renaming for a spatially-oriented operand network. This spatially-oriented operand network is organized as a unidirectional ring with a dedicated channel per architectural register. Figure 2.4 details the primary structures involved in CRIB. Register operands are received from dedicated channels routed along columns while instructions occupy entries within the rows. As instructions are fetched they are inserted in program order into these dedicated CRIB en-

tries. CRIB entries comprise a dedicated ALU, routing structures to receive and transmit sources and destinations, and completion logic. CRIB entries perform many functions: instructions wait until each input operand is ready, mux inputs from the register columns, execute on the dedicated ALU, drive their results onto the appropriate output column, and mark the destination output column as ready so later instructions may execute.

Although conceptually simple, these structures eliminate the need for traditional renaming and scheduling logic. By ordering the instructions and operand network according to the program sequence, CRIB guarantees that an instruction's operands will only be marked ready by the closest producing instruction in program order. This results on only true dependencies being maintained, without the need for a mapping table to help distinguish logical instances of an architected register. Additionally, global scheduling logic is unnecessary as CRIB entries are effectively self-scheduling. Once inserted, CRIB entries sit idle until all source operands are ready before executing and marking their output as ready, thus enabling later instructions to begin execution.

CRIB Operand Network

The operation of CRIB, due to its operand network, has many interesting tradeoffs with respect to superscalar architectures. Due to its organization as a unidirectional ring, CRIB exposes communication latency in routing results to later instructions. In CRIB, once produced, register values

propagate along the ring at the rate of four entries per cycle. If a dependent instruction is located within four entries of a producer, the result is available the cycle it is produced, and no delay due to operand routing is experienced. However, if dependent instructions are located far ahead in the program sequence from a producer instruction, this operand routing latency will be exposed. For conventional out-of-order processors, as long as dependent instructions are within the current instruction window, these far out consumer instructions do not face additional latencies and are scheduled for execution immediately following producer instructions. Although conventional superscalar processors perform well when executing instructions where an operand is produced much earlier in program order, many program-level behaviors work to mitigate this benefit. If the time critical operand for an instruction is located closely within the sequential instruction stream, the negative impact of far operand latency in CRIB will be masked. Secondly, previous work [29] has shown that the useful lifetime of register values are quite small. In the majority of cases a register value is last used within 10 instructions of its production. Thus, far communication amongst instructions is not the common case. Finally, instructions slowed by operand propagation may not be on the critical instruction path. CRIB has abundant execution resources, so serialized scheduling of these resources, as done in conventional processors, is unnecessary. Thus, delaying non-critical path instructions has impact on overall performance.

2.5 Operand Networks

As processors architectures evolved, they underwent a progression from simple multi-cycle processors, to pipelined scalar cores, and finally to the wide-issue out-of-order processors in existence today. To enhance the performance of these pipelined architectures, hardware bypassing was employed to eliminate certain pipeline hazards. Consequently, most instruction operands are obtained from bypass networks instead of the register file. However, due to the wide-issue and deep pipelined nature of modern processors, the complexity of these bypass networks can impose limits on achievable frequencies. With respect to our CRIB baseline architecture, we explore the impact of operand latency and necessary bypass paths for supporting in-place loop execution. In this section we explore operand network techniques that have been applied in traditional and non-traditional processor architectures.

Traditional and Clustered Bypass Networks

Traditional pipelined processors employ single-cycle bypass networks which forward results to dependent pipeline stages in the same cycle as they are produced. The latency of the bypass network and execution units in these traditional processors places a limit on the maximum processor frequency. This latency grows with respect to the issue width and pipeline depth of a processor. As issue width and frequency greatly impact the

performance of modern processors, this gives rise to two distinct processor design styles [34]. In the “brainiac" design style, a processor tries to achieve performance by extracting the maximum level of parallelism through a wide issue design – at the expense of a slow clock cycle. “Speed demons" conversely comprise simple, narrow width pipelines that are capable of operating at high frequency.

Architects have proposed many methods to reduce the negative impacts of complex bypass networks. Ahuja et al. investigate the performance impact limited bypass has on the performance of a scalar processor core [2]. Although limited bypass causes significant degradation in processor performance, Ahuja et al. find that intelligent compiler scheduling can help mitigate the negative impact. Commercially, the IBM POWER4 [85] and IBM POWER5 [73] designers took the ultimate “speed demon" approach and elected not to implement any bypass networks. To help mask the forwarding latency to dependent instructions, these designs rely on their out of order schedulers fill resultant pipeline bubbles with independent instructions.

Trying to break the dichotomy between these two design styles, multiple researchers have proposed clustered processor architectures [18, 27, 62, 92]. Clustered architectures serve as a mean of expanding the maximum width of a processor core while limiting the required number of bypass paths. In these architectures, bypass networks and register files are divided amongst two or more execution clusters. For instructions

within the same execution cluster, fully bypass paths are instrumented. Between execution clusters, limited bypass paths exist. This clustering approach has the effect of allowing back-to-back dependent instruction execution within a cluster. However, additional latency will be incurred if dependent instructions exist within other clusters. The performance limiting impact of inter-cluster dependent instructions can be mitigated by steering chains of dependent instructions to the same clusters. For well behaved codes, this mechanism can effectively result in the apparent performance of a fully bypassed processor core, without having to incur slow cycle times. Commercially, the Alpha 21264 was the first processor to implement clustering with two separate execution clusters [47].

Scalable Operand Networks

Although clustered architectures allow the issue width of a processor to be increased without directly impacting the frequency of the resulting processor core, there are limitations to the benefits of clustering. Additionally, alternative processor designs are able to benefit processor performance in non-traditional ways. These benefits encompass techniques such as making the instruction set architecture (ISA) aware of distributed processing models, exposing parallelism not captured by traditional out of order processors, and eliminating pipeline complexities. To this end, we provide a background on a relevant selection of distributed processor models with scalable operand networks.

RAW

As technology scaling progresses, the relative delay of on-chip wires grows [39]. Whereas the delay of wires in previous architectures was negligible, they may diminish the performance in future architectures. With this in mind, the RAW [84] processor architecture works to directly support inter-core communication by incorporating it within the ISA. By promoting the interfaces required for inter-core communication directly to the ISA level, it is hoped that software interfaces can achieve greater scalability and communication is made more efficient. In traditional chip multiprocessors, inter-core communication is carried out through coherent memory interfaces that incur more latency and energy than direct core to core communication.

The RAW processor architecture is a tiled architecture, where processor tiles are connected directly to their adjacent neighbors through two dimensional mesh networks. The RAW processor is a 16-core chip which utilizes four 32-bit interconnection networks to directly communicate between scalar processor cores. Two of the networks are statically routed with messages and routes being configured at compile time. The other two networks are dynamically configured at runtime. These networks are exposed through the ISA by mapping specific registers to the four on-chip operand networks. These registers are constructed as FIFO queues. If an instruction writes to a given register, the value is enqueued in the output for the selected network. When reading a network mapped register, the

most recently received value will be extracted from the incoming network. If no register value is present, the scalar processor stalls at fetch until a value is available.

Through a platform specific compiler, the RAW architecture is capable of making use of these operand networks. The explored region of codes within the RAW work include traditional scalar codes and enhanced multi-threaded applications communicating via its ISA level network.

Multiscalar

The Multiscalar processor architecture [80] is a design paradigm for extracting large quantities of instruction level parallelism from single-threaded programs. The multiscalar paradigm works by dividing single-threaded programs into sets of tasks and distributing these tasks across multiple processing elements. Tasks in multiscalar correspond to a contiguous region of the program control flow graph – potentially encapsulating many instructions and complex control flow. Once distributed, these tasks operate in unison, executing different portions of the program executable in a parallel fashion. Although tasks are distributed and executed in parallel, they are not required to be fully independent from each other as they represent subsections of a larger sequential instruction stream. Dependencies across tasks are supported in multiscalar processors by presenting the appearance of a unified logical register file and sharing a memory dependence unit across all processing elements. Dependencies and the

sequential commitment of tasks are efficiently supported in multiscalar by organizing processing elements along a unidirectional ring network. As tasks produce values needed by later sequential tasks, compiler generated masks control the forwarding and reception of values along this network. Additionally, as tasks complete a commit pointer advances along the ring-organized processing elements to reflect the committed architectural state.

Due to its interaction with high level program semantics, the multiscalar paradigm results in multiple benefits not commonly realized by traditional processor architectures. First, due to the task-oriented structure of multiscalar, multiscalar processors are capable of performing selective branch prediction. As a global sequencer provides each processing element with tasks, the global control flow path remains valid as long as no branches invalidate the currently executing tasks. Within tasks, branches may be mispredicted, but this has no impact on global control flow and the other tasks. This property allows multiscalar processors to effectively tolerate poor branch prediction within some tasks while extracting parallelism from others. This inter-task misprediction property is a form of control independence [66]. Second, as a task-oriented architecture, multiscalar can effectively extract parallelism over a very large contigious region of the sequential execution stream. Traditional out of order processors must hold all unexecuted instructions within issue queues, limiting the program region where parallelism can be extracted, whereas in multiscalar's task-based structure, only subsets of each task need be under consideration for

execution. Finally, due to the distributed nature of multiscalar's processing elements and its ring-based interconnect, multiscalar effectively implements software-controlled clustering. Inter-cluster, dataflow dependencies are tightly coupled via non-scalable bypass paths. Intra-cluster, the ring is orchestrated by mutiscalar's compiler controlled register passing.

TRIPS

The TRIPS architecture [68] is a recent initiative to build a scalable out of order processor through the use of a distributed architecture and a new ISA that explicitly encodes dataflow dependencies between large batches of instructions. In TRIPS, the compiler constructs blocks of 128 instructions and distributes them in groups of eight across 16 processing tiles. These processing elements are organized in a two dimensional mesh with single-cycle interconnections between nodes. As instruction placement within tiles is compiler orchestrated and instruction dependencies explicitly encoded through the ISA, the compiler can effectively minimize exposed communication latencies between dependent instructions. Like multiscalar and RAW, this leads to a scalable bypass network where latencies between processing elements are exposed and communication is expressed within the ISA. TRIPS operates as a dynamic issue machine, whereas the RAW architecture's instruction issue amongst its scalar cores is statically determined. Although a goal of TRIPS was to create a scalable and low latency operand network, performance of programs on TRIPS

is heavily limited by the latency and contention of its interconnection network. As shown in [68], the latency and contention within the TRIPS interconnection network can account for large fractions of program execution time. This observation motivates the importance in optimizing the communication latency of a distributed processor's operand network.

2.6 Summary

This chapter presented related work and background material central to the implementation of in-place loop execution. The exploration of loop caching shows the progression of loop buffers from general structures to highly optimized and embedded parts of modern microarchitecture. The history of prefetching provided context for the proposed load pre-execution technique enabled by in-place loop execution. Next the structure and design of traditional and non-traditional out-of-order processors was presented to give an overview of the structures being modified by our techniques. Finally, background on operand networks was provide to motivate our investigation into improving the CRIB operand network.

3 LOOP DETECTION AND TRAINING

To identify and eliminate redundant instruction dispatches during loop execution, the processor front-end requires the presence of dynamic loop detection hardware. Front-end loop detection and dispatch hardware operates in two primary phases. In the first phase the front-end must identify loops that are capable of being dispatched in loop-mode to the out-of-order back-end. In the second phase of operation, the front-end must determine which candidate loops should utilize loop-mode dispatch. In this section we detail the structure of the loop detection hardware and what factors determine the probability that a undergoes loop-mode dispatch.

3.1 Identifiable Loops

This section provides an overview of the types of loops which our Revolver architectures support for loop-mode dispatch. Figure 3.1 presents examples of multiple loops which are potentially supported in our Revolver designs. Simple conditional loops that have a single point of entry and exit, like Figure 3.1a, are easily identified and handled by the loop hardware. Our designs also support the dispatch of loops like Figure 3.1b that contain nested control. These loops are handled by predicting that all iterations of a given loop follow the same inner control path. Thus

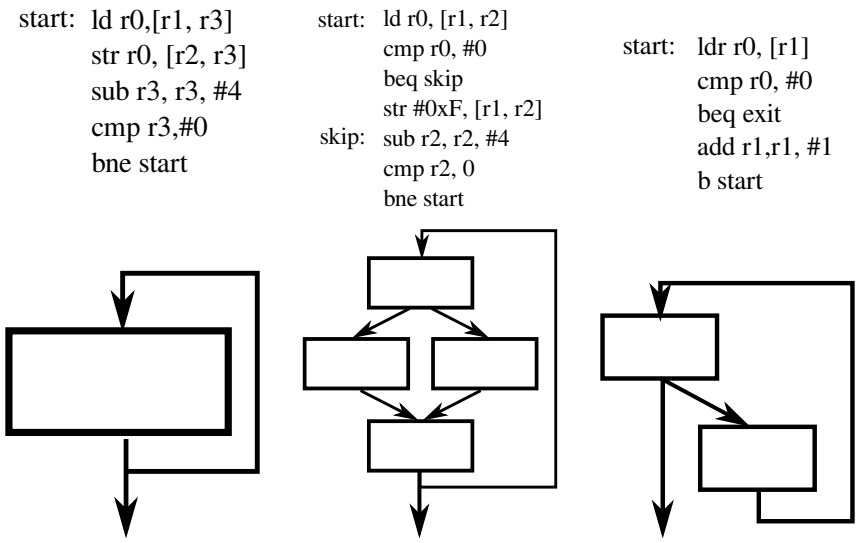


Figure 3.1: Loop Types

the Revolver architectures can dispatch loops with almost unlimited inner control, provided the same path is taken across many iterations. For loops with unstable control, feedback described in Section 3.4 is necessary to disable loop-mode dispatch. Finally, our loop detection hardware is also capable of identifying loops that utilize early exits like in Figure 3.1c. It should be noted however that this loop structure is quite rare as it is not often generated by the GCC 4.7.2 compiler utilized in benchmark compilation.

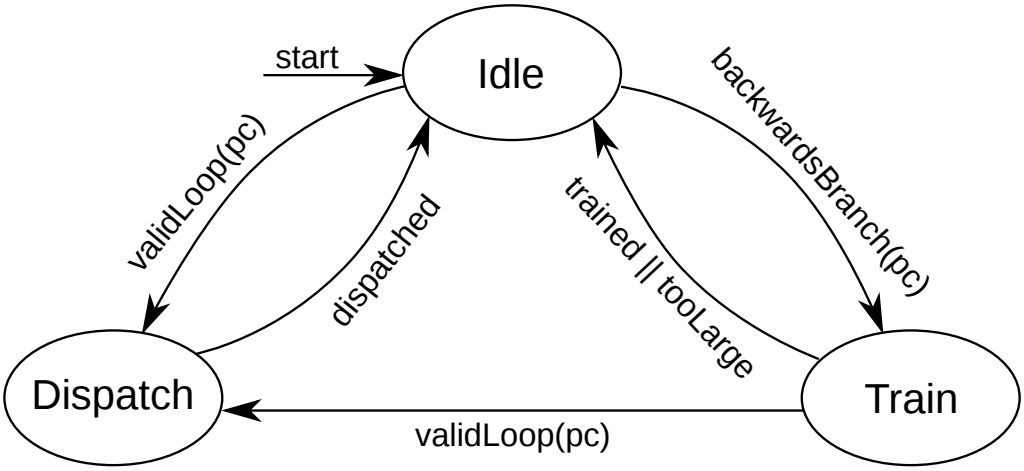


Figure 3.2: Loop Detection Finite State Machine.

v	start addr	fallthrough addr	num_insts	max_unroll	profitability

Figure 3.3: Loop Address Table (LAT) Structure.

3.2 Detection Operation

The loop detection logic in the Revolver architectures is similar to that used in previous loop buffer proposals [40, 54]. Loop detection is controlled by the simple finite state machine (FSM) shown in Figure 3.2. This state machine operates in one of three possible states: `Idle`, `Train`, or `Dispatch`.

In the `Idle` state, instructions propagate normally through decode until the start of a profitable loop is identified or a taken PC-relative backwards[1]

[1]Branch target instruction address less than current instruction addresss.

branch/jump is encountered. Profitable loops are identified by consulting a small structure containing known loops called the Loop Address Table (LAT). The LAT, detailed in Figure 3.3, is a small direct-mapped structure that records information relating to the loops composition and profitability. In the event a profitable loop is encountered, the detection FSM transitions to the Dispatch state and begins loop-mode dispatch. If no profitable loop is identified and a backwards branch or jump is encountered, the detection FSM instead transitions to the Train state.

The Train state exists to record a previously unknown loop's start address, end address, and allowable unroll factor[2] in the LAT. Once entering the Train state, until the loop ending branch, resources required by the loop body are recorded. After the ending branch is encountered, the loop information is entered into the LAT and the FSM transitions to the Idle state again. If a loop requires too many resources to be contained by the back-end, the LAT is not updated. The fall through address for a loop is set to the next sequential memory address. It should be noted that, if a loop start instruction is encountered at any time, training will be aborted and the FSM will immediately transition to the Dispatch state. Finally, if another backwards control instruction is encountered, the resource usage information is reset and the Train state is re-entered.

In the Dispatch state, the decode logic guides the dispatch of loop instructions into the out-of-order back-end by specially tagging them as

[2] As constrained by physical resources.

loop instructions. The loop body is unrolled as many times as possible, subject to available back-end resources. After unrolling all loop instances, the front-end is redirected to the fall through path. Once the fall through path fills the front-end, the front-end stalls and clock gates.

3.3 Detection Discussion

In this section we highlight multiple aspects of the previously described loop detection mechanism.

First, the Revolver architectures allow almost unlimited control flow, including function calls/returns, within a loop body. The only limitation is that predicted execution paths are statically determined at the time of loop dispatch. Thus loops with unstable control flow make poor candidates for loop-mode and back-end feedback is responsible for eventually disabling loop-mode dispatch of such loops.

Secondly, loop-mode is disabled for a given loop if it contains *serializing* instructions. Examples of *serializing* instructions include system calls, memory barriers, and load-linked/store-conditional pairs.

3.4 Training Feedback

The back-end feedback serves one primary purpose: relaying information about the profitability of a loop body. The use of feedback enables

the Revolver architectures to successfully eliminate instruction dispatch overheads while limiting any potential negative performance impacts. In Chapter 8 we evaluate the impact of no feedback on loop-mode dispatch. The remainder of this section describes our developed feedback mechanism.

Shown in Figure 3.3, the LAT contains a *profitability* field that acts as a 4-bit saturating counter. Upon insertion into the LAT, loops receive a default profitability of 8. Loop-mode dispatch is enabled if profitability is greater than or equal to 8. Feedback from the back-end adjusts a loops profitability to impact its likelihood of loop-mode dispatch.

The following factors impact a trained loops profitability. If the dispatched unrolled loop body iterates more than twice the profitability is incremented by 2, otherwise it is decremented by 2. If a branch within the loop body mispredicts to an address that other than the fall through, the loops profitability is set to zero. For disabled loops, the front-end increments the profitability by 1 for every two sequential successful dispatches observed. The absolute values of these profitability factors was determined through iterative simulation and performance measurement across three benchmark suites.

Adjusting by these factors ensures that only highly profitable loops are enabled for loop-mode dispatch, thus mitigating any potential negative performance impact while capturing the majority of potential benefit.

3.5 Summary

In this chapter the capabilities and operation of Revolver's loop detection hadware were described. Additionally, the necessity and operation of back-end feedback in order to restrict loop mode execution to profitable loops was described.

4 CONVENTIONAL BACK-END LOOP EXECUTION

Revolver's conventional out-of-order back-end supports loop-mode execution through a series of simple modifications to the issue queue, load/store queue, and commit logic. These modifications allow loop instructions within the back-end to be executed multiple times in order to complete all necessary loop iterations. During subsequent executions, all instructions retain their initially allocated resources. In this chapter we provide an overview of back-end operation as well as the required structural modifications.

4.1 Overview

To summarize back-end functionality, Figure 4.1 provides an example of loop-mode execution performing a string copy operation[1]. In this example, the six instruction string copy loop is unrolled twice into the issue queue. The first (green) loop body performs all odd-numbered iterations while the second (blue) loop body completes all even-numbered loop iterations. This partial unrolling allows parallelism across iterations during loop-mode execution.

[1]Copy bytes from source array to destination array until encountering null.

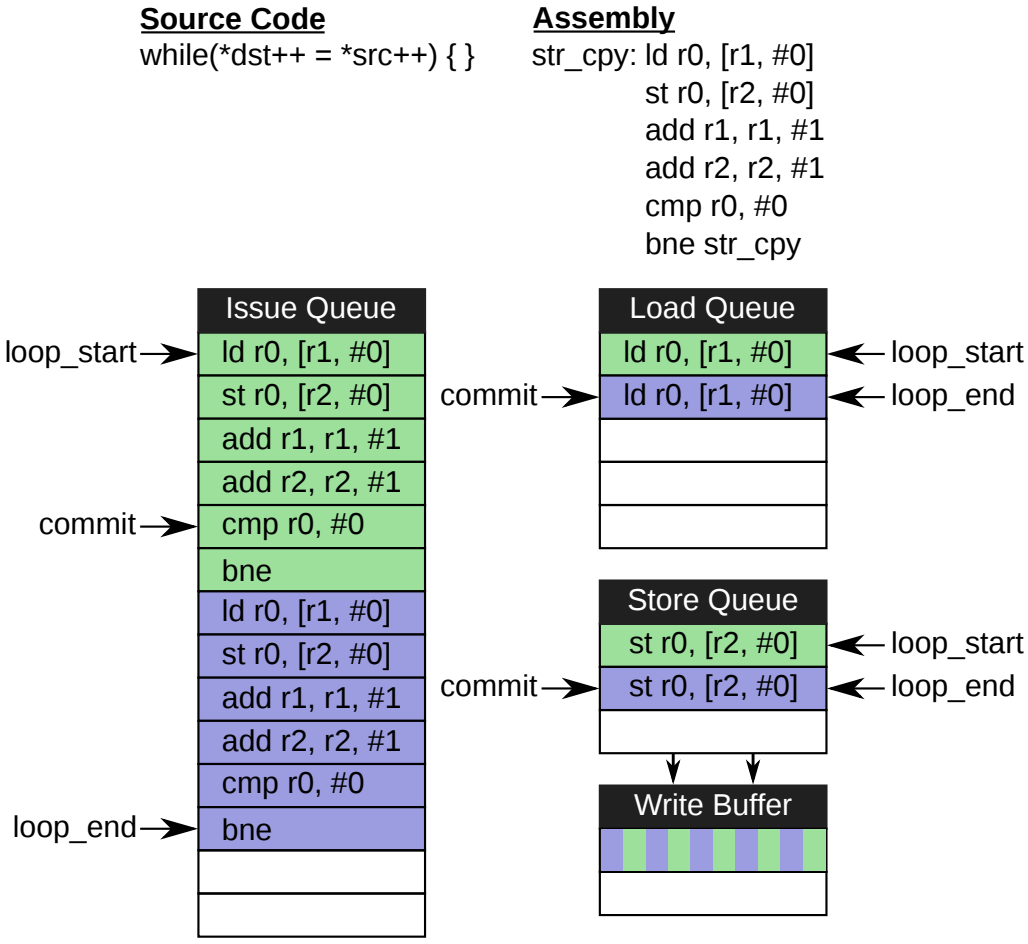


Figure 4.1: Revolver Out-of-Order Back-end Example.

For maintenance and ordering, loop start and end pointers are tracked by each back-end queue. The queues also maintain a commit pointer that identifies their oldest, uncommitted entry. Shown in Figure 4.1, the commit pointer walks from the loop start until the loop end entry. After committing the loop end instruction, the commit pointer wraps to the loop start to begin committing the next loop iteration. Upon commit, issue

queue entries are reset and can be immediately reused for the next loop iteration. Load queue entries are simply invalidated on commit, while store queue entries drain into a small write combining buffer. Draining stores into a write buffer allows the store queue entry to be immediately reused in the next loop iteration. In the rare instance when a store cannot drain into the write buffer, commit stalls. Finally, loop-mode reuse of LSQ entries requires no modification to the age-based ordering logic of the LSQ. LSQ ordering logic must already support wrap-around based upon the relative position of a commit pointer in a conventional out-of-order. To demonstrate this, given the example's relative position of the commit pointer in Figure 4.1, the second (blue) loop body store is properly ordered as older than the first (green) loop body store.

Loop-mode execution completes when any branch, loop terminating or otherwise, resolves to the loop's fall through path. Allowing any branch which resolves to the fall through path to terminate loop-mode execution means that loops end gracefully even on iteration counts that are not evenly divisible by the unrolling factor. Additionally, this resolution handling allows *break* statements within loop bodies to quickly resolve without being treated as mispredicts. After termination, the loop's out-of-order resources may be freed and the fall through path immediately proceeds through dispatch.

In the remainder of this chapter we describe the precise structural modifications necessary to support this operation.

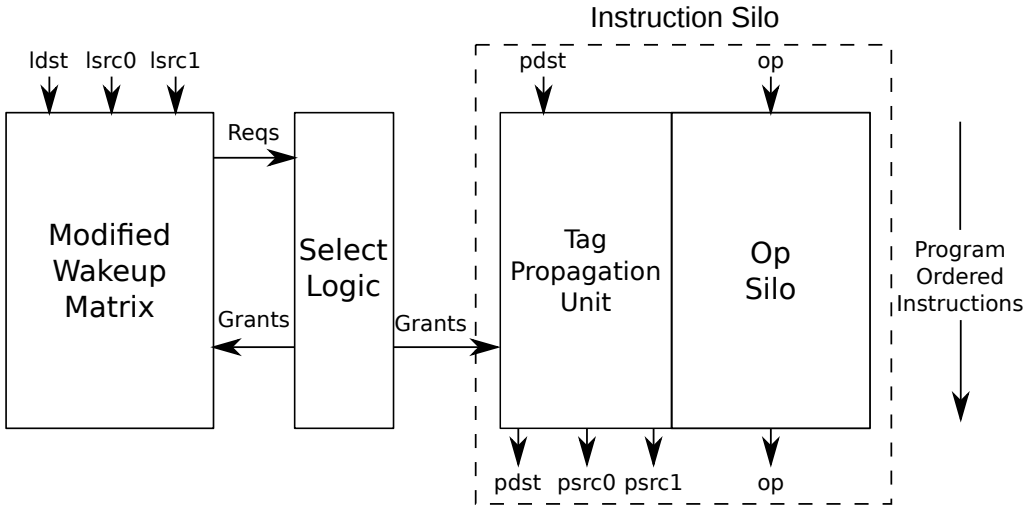


Figure 4.2: Revolver Out-of-Order Issue Queue Design.

4.2 Scheduler Modifications

Key to supporting loop-mode execution in Revolver is the structure and operation of the instruction scheduler. The overall design of the Revolver's scheduler, shown in Figure 4.2, is similar to the matrix scheduler presented in [69] by Sassone et al. The main components of this scheduler are a wakeup array for identifying ready instructions, select logic for arbitration between ready instructions, and the instruction silo for producing the opcode and physical register identifiers of selected instructions.

Three primary modifications make the Revolver scheduler distinctive from [69]. First, Revolver's scheduler is strictly managed as a queue and maintains program order among entries. Secondly, the wakeup array utilizes logical register identifiers and position dependence, rather than

physical register identifiers for wakeup. Finally, Revolver uses a Tag Propagation Unit (TPU) to provide physical register mappings, instead of a front-end RAT combined with back-end tag storage.

With the high level structure of the scheduler defined, the following Sections 4.3 and 4.4 explain the function and operation of the Revolver's instruction wakeup and Tag Propagation Unit.

4.3 Wakeup Logic

The purpose of instruction wakeup in an out-of-order processor is to observe results generated by scheduled instructions in order to identify new instructions capable of being executed. To perform this task, Revolver's instruction wakeup utilizes program-based ordering of instructions and logical register identifier broadcasts. This differs from many conventional schedulers, which use physical register-based broadcasts and do not require ordering. The primary benefit from this organization is that Revolver is able to perform instruction wakeup without requiring front-end renaming.

Wakeup Operation

Figure 4.3 shows an overview of the Revolver wakeup logic. At the highest level, instruction wakeup is organized as a segmented, program-ordered circular queue. The segmented wakeup arrays within the scheduler are

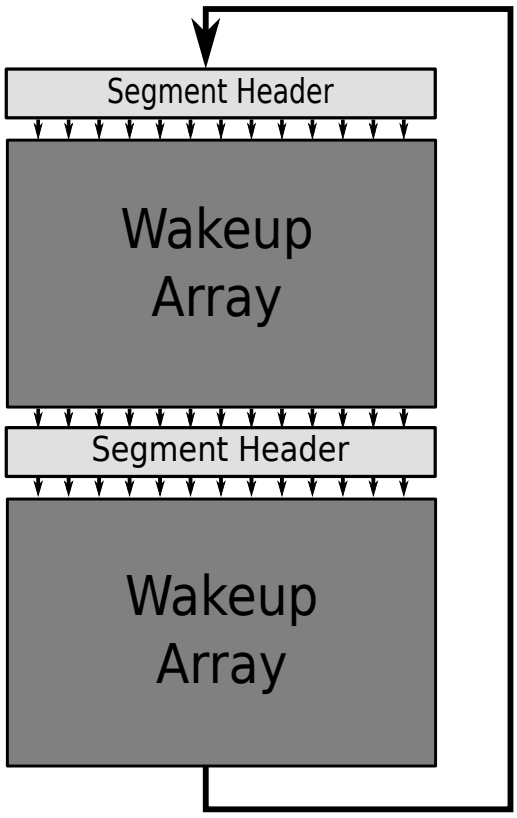


Figure 4.3: Wakeup Overview

interconnected via a unidirectional ring that transmits logical register broadcasts. In our designs, segments are sized equal to the machine's dispatch width and broadcasts along the ring interconnect travel at the rate of eight instruction entries per cycle. At any given time, one segment in the machine will be designated the *architected* segment, with incoming operands implicitly ready.

Inside the wakeup array, shown in Figure 4.4, instructions are distributed along rows, while columns correspond to logical registers. Upon

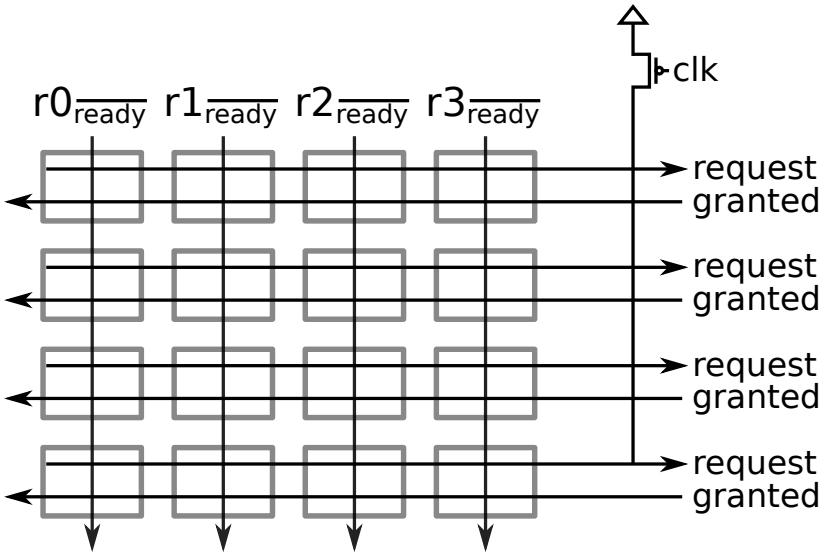


Figure 4.4: Wakeup Array

allocation into the wakeup array, instructions mark their respective logical source and destination registers. Unscheduled instructions within the wakeup array cause their downstream logical destination register column to be deasserted. This deassertion prevents younger, downstream dependent instructions from waking up. Once all necessary source register broadcasts are received, an instruction requests scheduling. After being granted by select, the instruction asserts its destination register to wakeup younger dependent instructions.

Close examination of the wakeup array's logic cell, shown in Figure 4.5, demonstrates how the wakeup operation is possible. Our modified wakeup cell design draws from earlier work in [69]. The wakeup cell holds two state bits that designate the instruction as sourcing or producing a

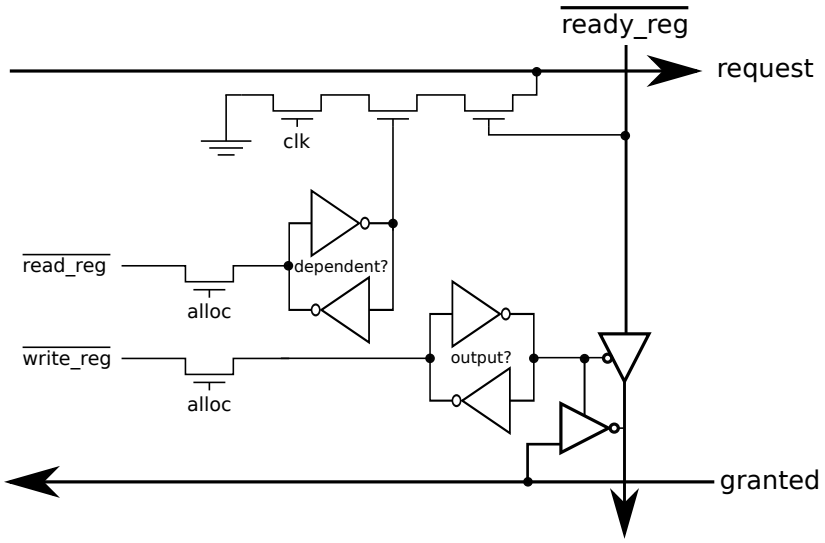
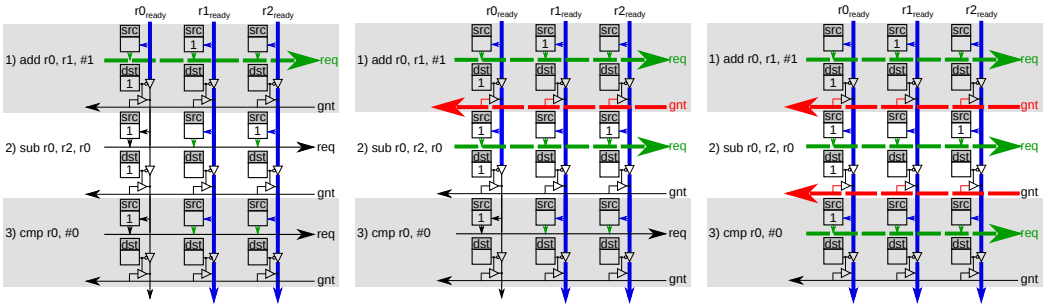


Figure 4.5: Wakeup Cell

logical register. The request signal to the select logic is implemented in dynamic logic. If an instruction is dependent and the incoming ready signal is not asserted, the request signal is pulled down. This pulldown operation works as a logical NOR. If no un-broadcast dependents remain, the request signal to select will remain asserted. With respect to outputs, if an instruction has not been scheduled and produces the logical register, the outgoing ready signal will be deasserted. The ready signal and grant signal are implemented with static logic. Once the result producing instruction is granted, the grant signal will result in the downstream ready being asserted. For loop-mode operation, after commit, the grant signal is deasserted and the cell is free to reevaluate based upon a new incoming ready signal.



(a) Cycle 0 - Inst1 ready. (b) Cycle 1 - Grant Inst1. Inst2 ready. (c) Cycle 2 - Grant Inst2. Inst3 ready.

Figure 4.6: Revolver Wakeup Example

Wakeup Example

In this section we work through a simple example of wakeup logic operation. Figure 4.6 provides an example with three instructions. All three instructions must be serially executed due to dependencies on logical register *r0*. The diagram is color coded with ready register columns colored in blue, active request signals represented in green, and active grants represented in red.

On cycle 0, the ready signals for logical registers propagate downwards unless inhibited by an instruction. As *Instruction 1* produces register *r0*, it gates the downstream ready broadcast until it has been scheduled. This prevents improper wakeup of *Instruction 2* and *Instruction 3*. Logical registers *r1* and *r2* are produced by no instruction, thus their ready broadcasts are uninhibited and continue propagation. During cycle 0, *Instruction 1* satisfies all dependencies and asserts its request vector.

In cycle 1, the grant signal for *Instruction 1* returns acknowledging issue. Signalling issue, *Instruction 1* ceases inhibition of the downstream *r0* ready signal. With *r0* now asserted, *Instruction 2* asserts its request vector. As *Instruction 2* has not been granted, it maintains downstream inhibition of *r0*, preventing *Instruction 3* from waking up.

Finally on cycle 3, the grant for *Instruction 2* returns and *r0* is uninhibited. With *r0* now asserted, the final instruction wakes up and assets its request vector.

This example demonstrates how the use of program-based ordering enables Revolver to perform instruction wakeup on logical register identifiers, in contrast to conventional out-of-orders which require front-end renaming and physical register-based wakeup.

4.4 Tag Propagation Unit

Without front-end renaming, Revolver needs a mechanism to properly map logical registers to physical register identifiers. In this section we discuss the reasoning behind the use of the TPU in Revolver and its operation.

Enabler of loop-mode execution

The reason Revolver requires a TPU is to enable reuse of physical registers during loop-mode execution. As noted earlier, Revolver does not require any additional resource allocations between loop iterations. The largest

obstacle to avoiding allocation is physical register management. This is so because, after committing, a loop instruction should be free to begin speculative execution of the next loop iteration. This is impossible, however, if an instruction only has access to a single physical destination register. After commit, the contents of the physical register may be required by dependent instructions and are part of the architected state. Thus, to begin speculative execution of the next loop iteration, access to an alternative physical register identifier is required.

Revolver solves this issue by providing each result producing loop-mode instruction with two physical destination registers. As loop instructions iterate, they simply alternate writing between their two destination registers. This alternation of writes, known in other literature as double buffering, ensures the previous state is maintained while speculative computation is being performed. To clarify, after iteration $N + 1$ commits, an instruction may reuse the destination register from iteration N on iteration $N + 2$. This is safe because, upon commit, the $N + 1$ destination register holds the architectural state and no more instructions are dependent on the iteration N destination register. Any instructions dependent on the $N + 1$ value continue to source it from the alternative register.

This double buffering technique enables instructions to speculatively write output registers. However, by dynamically changing output registers, additional functionality must be added to properly maintain dependencies between instructions. The TPU's function is to perform this dynamic

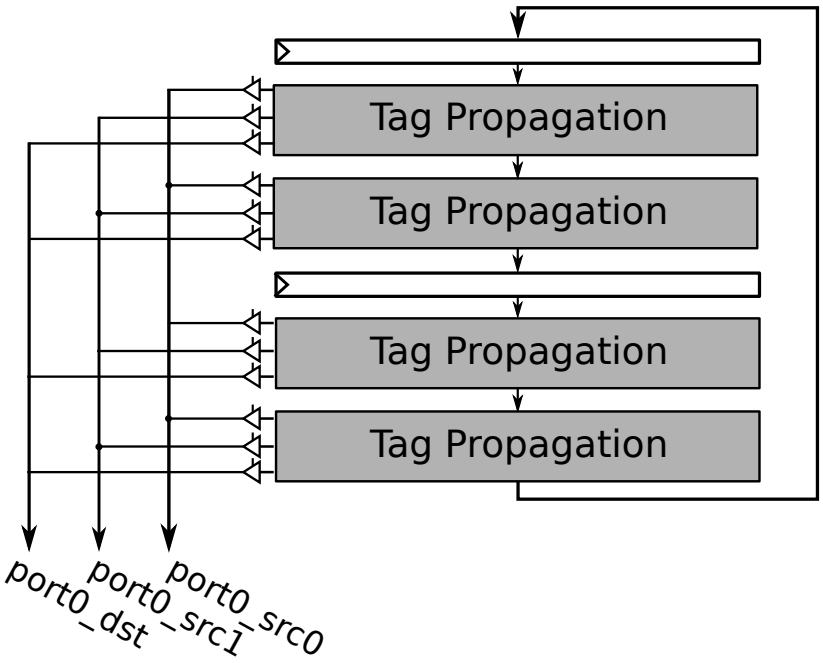


Figure 4.7: Tag Propagation Unit.

linkage between dependent instructions and source registers.

Structure and Operation

Figure 4.7 shows the high level structure of the TPU. Revolver's TPU is structured similarly to the wakeup logic discussed in Section 4.3. Like the wakeup logic, the TPU is composed of partitions interconnected along a unidirectional ring interconnect. The ring is composed of multiple channels, where each channel corresponds to a logical register and carries the current physical register mapping. Thus to obtain source register identifiers, all an instruction must do is source the appropriate logical register

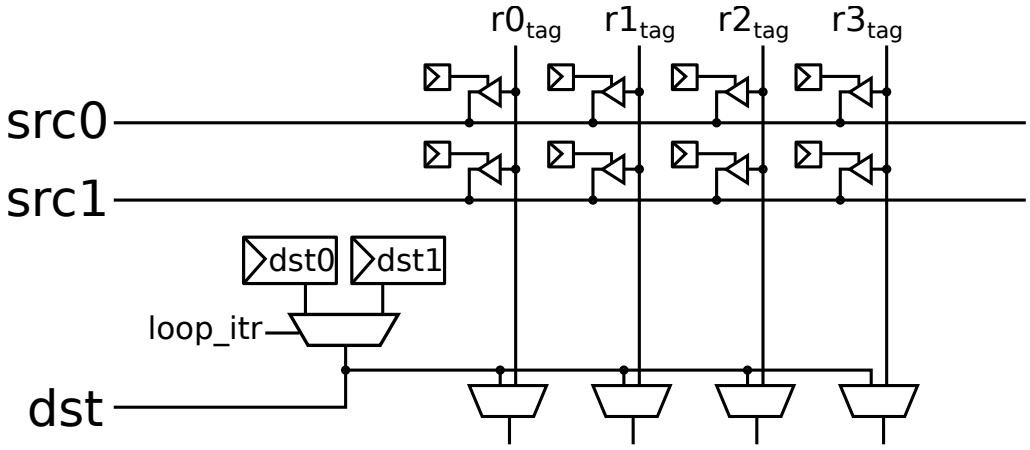


Figure 4.8: Tag Propagation Unit Cell Design.

channel. Instructions present in the scheduler change the logical register mapping of their output register by simply overwriting the appropriate output column. Since instructions are stored in program order, this operation guarantees downstream instructions will obtain proper source register mappings. At all times, one segment is deemed *architected* and retains the architected register mapping in latches. This *architected* latch rotates throughout the TPU as segments commit.

Figure 4.8 shows the propagation logic cell used in the construction of the TPU. As seen there are multiple logical register columns that carry physical register identifiers. Instructions source their operand tags from corresponding columns and drive their destination onto the appropriate output column. Also within the figure, we show how loop-mode instructions alternate between writing two physical destination registers. Essentially a single bit of state records if the instruction has executed

an odd or even number of times, this bit controls a mux that drives the appropriate destination register identifier onto the output.

Checkpoints and Register Reclamation

In addition to dynamic dependence linking, Revolver's TPU provides benefits relating to checkpointing and branch misprediction recovery. Every instruction within the TPU has access to a valid physical register mapping for every architectural register. Thus Revolver effectively provides per-instruction renaming checkpoints. If any instruction mispredicts, downstream instructions are simply flushed and the mappings from the branch instruction propagate to all newly scheduled instructions. In comparison, RAT-based renaming encounters significant additional complexity in order to support checkpoints [55]. This checkpoint support is largely a byproduct of Revolver's progam ordering within the issue queue.

In non-loop mode, overwritten registers are reclaimed on commit, as in conventional processors. On branch mispredictions or loop-mode exits, the TPU is walked forward from the terminating branch to reclaim physical registers.

Latency Impact

Amongst the structures modified within the scheduler, the TPU represents the most significant deviation from conventional design. As the

corresponding wakeup logic is simpler, the critical path of the scheduler is determined by the worst-case path delay of the TPU. This path delay is incurred when a newly produced destination tag is immediately sourced by a distant instruction. This cycle time is constrained by the maximum allowed 8-entry propagation path from a producer tag to dependent sourcing. The total delay is given in equation (1).

$$tpu\_latency = (register\_mux\_delay) + (8 * column\_mux\_delay) \quad (1)$$

Given the latency of 64-bit arithmetic operations and load/store queue complexity of modern processors, it is not expected for this delay to be a significant design limitation.

4.5 Load/Store Support

The overall structure of Revolver's LSQ is shown in Figure 4.9. Other than the additional tracking of loop start and end pointers, few differences exist between Revolver's LSQ and that of a conventional out-of-order. Loads and stores receive their respective LSQ entries prior to dispatch and retain them until the instruction exits the out-of-order back-end. As noted in Section 4.1, loop-mode load and store instructions are free to reuse their allocated LSQ entries to execute multiple loop iterations. This is due to two factors: 1) The immediate "freeing" of a LSQ entry upon commit and 2) The use of position-based age logic in modern processor's LSQs.

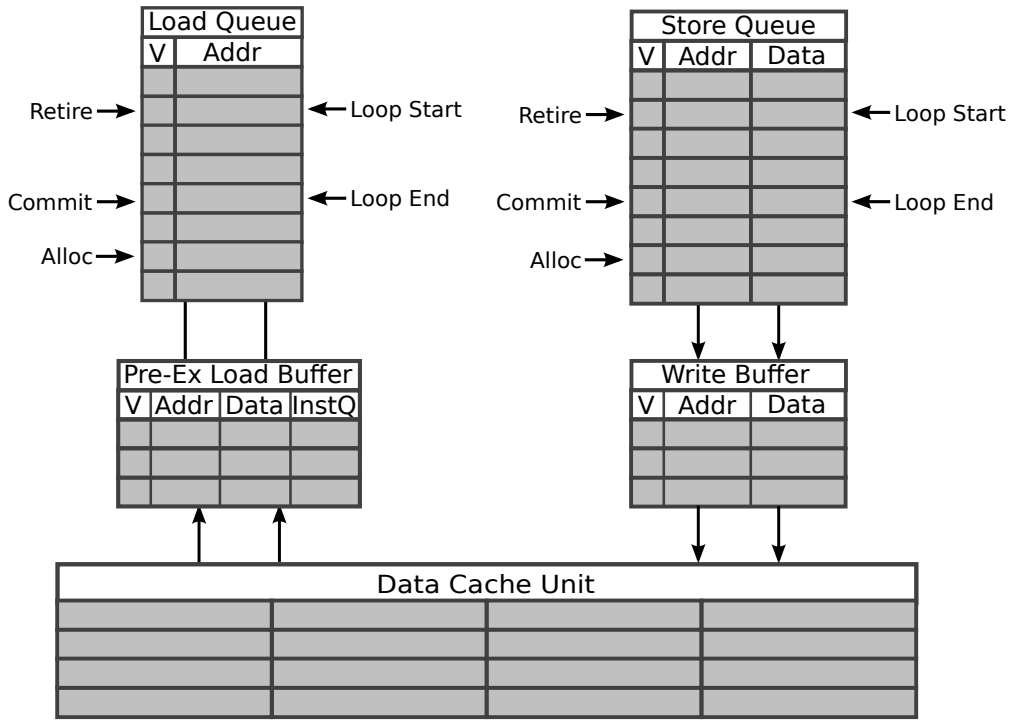


Figure 4.9: LSQ and Cache Interface.

LSQ entries are immediately freed upon commit by two means depending on whether the committing instruction is a load or store. In the event of a load, the load queue entry is simply reset to allow future load execution. Stores however must be written back to memory. To enable immediate freeing of store queue entries, stores are drained into a write-combining buffer that sits between the store queue and the L1 cache interface. If a store cannot drain into the write buffer, commit stalls. This is however a rare occurrence as due to the impact of write-combining and the fact we target an ISA with a relaxed consistency model that places very few ordering restrictions on write-combining. In ISAs with more restrictive memory

ordering, like total store ordering (TSO) or sequential consistency (SC), additional limitations are imposed. For TSO systems, a write-buffer may still be utilized, however write combining will be inhibited and ordering must exist between the completed stores. Additionally, due to the lack of write combining, a larger buffer will be required for equivalent performance. A processor implementing sequential consistency will be even more limited by the lack of a store buffer. However research proposals to address the lack of a store buffer with additional hardware resources do exist [56].

Existing age-based ordering techniques work in Revolver as they are based upon the relative position of a commit pointer. The only difference during loop-mode operation is that Revolver's commit pointer wraps from *loop end* to *loop start*. Whereas a normal LSQ only wraps the commit pointer based upon the physical end and start of the queue.

The final portion of Revolver's LSQ, the pre-executed load buffer, is an enhancement enabled by loop mode execution and will be discussed in the following chapter.

4.6 Summary

In this chapter the Revolver architecture for a PRF-based out-of-order processor was introduced. It was shown that in-place loop execution is possible without the reallocation of physical resources. Necessary modifications to the operation of operand dependence linking, instruction

scheduling logic, and the load/store unit were also presented.

5 LOAD PRE-EXECUTION

In this section we detail an extension to loop-mode that enables the pre-execution of loads from future loop iterations. Pre-executing future loads realizes parallelism beyond the processor's instruction window and can be used to enable single-cycle loads. The remainder of this section covers the insight behind load pre-execution, the conditions where load pre-execution is possible, and why load pre-execution would be untenable in a conventional out-of-order without in-place loop execution.

5.1 Optimization Insight

During loop execution in an out-of-order processor, loads from within the loop body are repeatedly executed until all necessary iterations complete. Due to the recurrent nature of loops, these loads often have highly predictable address patterns. Our load pre-execution mechanism aims to exploit these predictable loads.

Revisiting the example from Figure 4.1, the string copy loop simply strides through memory copying bytes from a source array into a destination array. Thus, the load addresses in consecutive iterations are perfectly predictable. In a conventional processor, the dynamic instances of each load receive unique issue queue and load queue entries. In Revolver however, a load dispatched by the front-end is statically bound to fixed entries

for all loop iterations. This static binding makes it easy to observe when an entry is performing loads that follow a simple pattern. In Figure 4.1, since the string copy loop was unrolled by a factor of two, the first load queue entry will be observed striding through memory, reading consecutive even-addressed bytes from memory.

The insight behind load pre-execution is that, when these patterns are recognized, it is possible to speculatively initiate future iteration loads. On the next iteration, if a load was pre-executed, it will not pay the L1 cache access latency and will complete after verifying the pre-executed load address. This technique yields a performance benefit when the out-of-order execution window is insufficient to hide a load's latency.

Load pre-execution, by queuing future iteration loads in parallel with existing computation shares some similarities to decoupled architectures like the Astronautics ZS-1 [77]. Previously, decoupled architectures would rely upon explicit address and computation instruction streams. The address stream would be utilized to queue up future required loads. Through load pre-execution, the Revolver architectures are effectively extracting the address stream for load queuing for a subset of loads. The next sections detail the supported load pre-execution access patterns and hardware implementation.

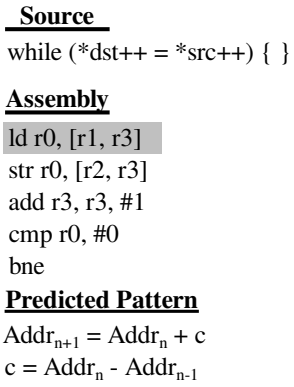


Figure 5.1: Strided Load Example.

5.2 Supported Address Patterns

In Revolver we support three primary access patterns for load pre-execution: stride, constant, and pointer-based addressing. For each of these access patterns we place simple pattern identification hardware alongside the pre-executed load buffer. Examples of each type of pre-executed load are shown in Figures 5.1, 5.2, and 5.3.

Striding memory accesses, shown in Figure 5.1, are the most common addressing pattern, as many loops iterate over arrays of data. To identify stride-based addressing we simply compute the address delta between two consecutive loads. If a third load from the same load queue entry matches the predicted stride, the stride is verified and the next load will be pre-executed. This method of stride detection is similar in structure to the earlier prefetching work done by Baer and Chen [9]. The primary difference is that strides are detected with respect to a specific dispatched

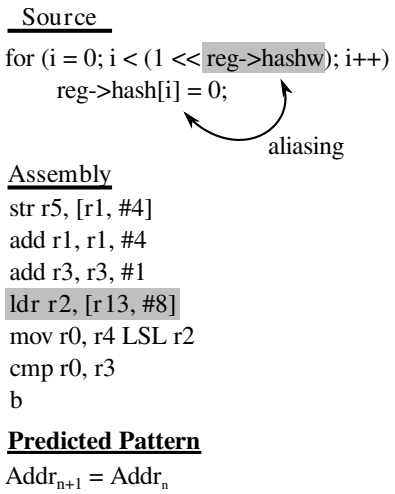


Figure 5.2: Constant Load Example.

instance of a load instruction, rather than performing a lookup operating using the load instruction's program address.

Constant loads, the second most common pattern, occur when loads continuously read from the same address. Constant loads exist primarily due to stack-allocated variables and pointer aliasing. These conditions force the compiler to spill registers and access variables through data memory. An example of a constant load from the SPEC CPU2006 benchmark *462.libquantum* is shown in Figure 5.2. Due to indirect pointer access, the compiler can not guarantee that the loop store does not alias with the *hashw* variable. This aliasing forces the *hashw* variable to be reloaded each iteration despite its constant value. The stride-based prediction hardware also handles constant loads, as they are a special case of a zero-sized stride.

Finally Revolver supports pointer-based addressing, where the value

Source

```
while (ptr->next) ptr = ptr->next;
```

Assembly

```
ldr r3, [r3, #4]
cmp r3, #0
b
```

Predicted Pattern

$Addr_{n+1} = Data_n + c$

$c = Addr_n - Data_{n-1}$

Figure 5.3: Pointer Load Example.

returned by the current load value is related to the next iteration's address. Shown in Figure 5.3, the next iteration's load address is predicted as the previous iterations data plus a constant offset. This captures many simple linked list traversals.

Once a pattern is recognized, the pre-executed load buffer speculatively initiates the next iteration memory access. This buffer, shown in Figure 4.9, sits between the load queue and the L1 cache interface. Once the next iteration load executes, the value is claimed and the buffer may initiate the next iteration load. If any store aliases with the pre-executed load, the entry is invalidated to maintain coherency. Loads to invalidated pre-executed loads execute normally, accessing memory through the normal cache structures.

5.3 Scheduler Modification

With the pre-execution buffer and supported access patterns defined, we now detail how Revolver's out-of-order scheduler can take advantage of pre-execution.

In many out-of-order designs, operations dependent on loads are speculatively scheduled assuming an L1 cache hit latency. When a pre-executed load returns from memory, the corresponding issue queue entry is notified that the load has been pre-executed. Once scheduled, this load will speculatively wake dependent operations on the next cycle instead of waiting for the L1 cache access latency. If the predicted address is incorrect, the scheduler must perform a cancel and re-issue operation. The design could be more aggressive than as described here and wake dependent operations before the load is capable of being scheduled, however our evaluated implementation does not support this. For designs to support this more aggressive operation, either fine grained recovery and re-execution of dependent instruction chains must be supported or an expensive flush recovery operation performed in the case of incorrect speculation.

For the CRIB-based architecture, described in Chapter 6, no scheduler modifications are necessary as CRIB-based architectures easily support variable latency operations. For this architecture, the pre-executed load buffer is simply probed when a load's effective address is written to the load queue. Upon a match, the pre-executed load value is forwarded

immediately marking the load as complete.

Finally, it should be noted that performing this scheduler optimization in a conventional out-of-order is untenable, as there is no relation between static load instructions and issue queue entries. By retaining specific load queue and issue queue indices across multiple executions, the scheduler can be easily modified to utilize the reduced load latency operation. To perform such an operation in a conventional out-of-order would require delta-based offsets between recurrences of load instructions and the corresponding issue queue entries, leading to significant predictor complexity.

5.4 Summary

In this chapter we introduced a new technique called load pre-execution to enable single-cycle loads, effectively hiding the latency of the L1 cache. In addition to the supported memory access patterns, necessary hardware modifications were discussed. Load pre-execution is evaluated in Section 8.4 and found to benefit some benchmarks by up to 10.6% in performance.

6 CRIB BACK-END LOOP EXECUTION

In this chapter we detail the necessary extensions to enable in-place execution on a CRIB-based out-of-order processor. In addition to describing necessary datapath modifications, we detail additional loop constructs that can be easily handled during loop-mode on the CRIB-based out-of-order. Finally in section 6.4 we describe an alternative implementation of the load store queue that effectively supports in-place execution.

6.1 Overview

The primary motivation behind the CRIB architecture is that conventional out-of-order processors were designed in an era where physical resources were scarce and power consumption was a secondary design constraint. The CRIB architecture in contrast is designed with dynamic energy consumption as a primary design constraint with physical resource utilization as a secondary design constraint. With respect to in-place loop execution, the resulting design offers four attractive architectural features: the avoidance of traditional register renaming, program-based ordering of instructions within the out-of-order execution core for resolving data dependencies, an LSQ that supports dynamic allocation within the out-of-order back-end, and a circular ring interconnect for propagation of register operands. Utilizing these features, this chapter dscribes how the CRIB

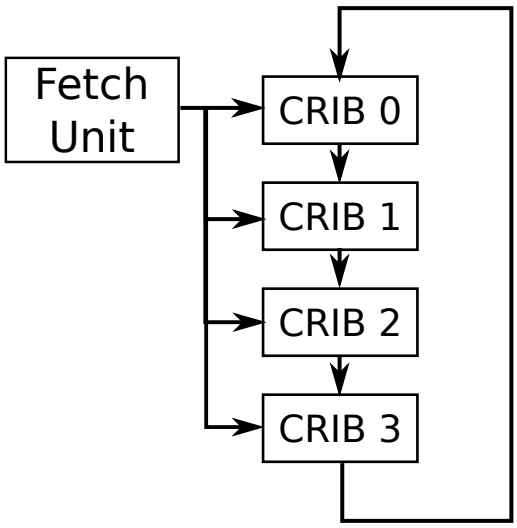


Figure 6.1: CRIB Overview

architecture can be extended to support in-place loop execution.

6.2 Datapath Modifications

Shown in Figures 6.1 and 2.4, the existing CRIB operand network is structured as a unidirectional ring. The simplest means of supporting in-place loop execution on CRIB is to place a single iteration of loop within CRIB instruction window, making all other CRIB entries no-op instructions. As instructions within the loop execute they retain their corresponding CRIB entries and await ready operands for the next loop iteration. Allocation, deallocation, and commit of CRIB partitions is guarded through three pointers as done for the issue queue in the PRF-based Revolver design in Figure 4.1. Otherwise, the unidirectional ring operand network and ready

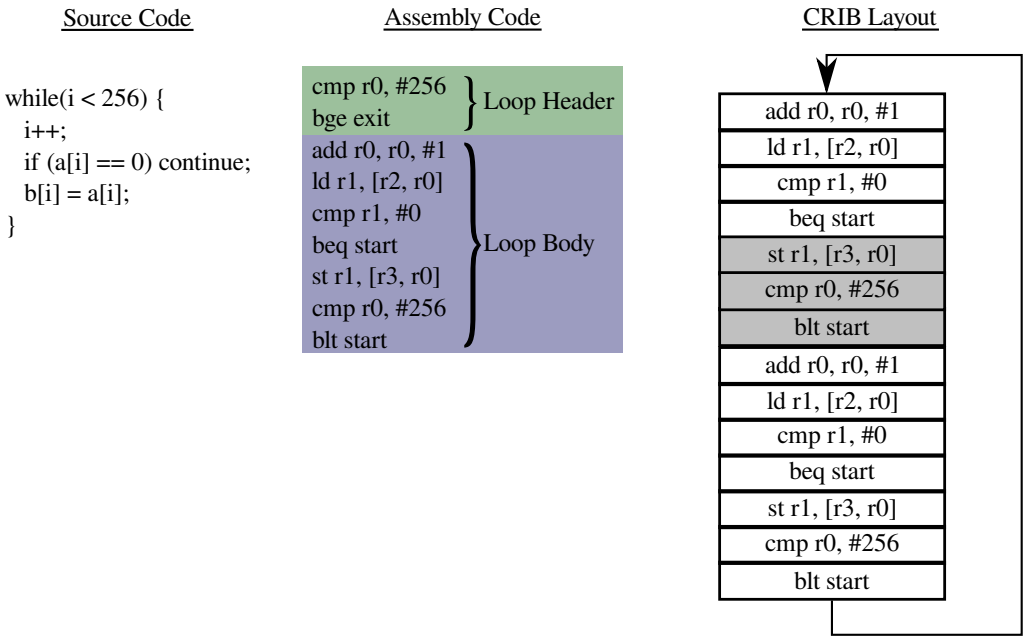


Figure 6.2: Loop Continue Example

logic of the existing CRIB baseline sufficiently supports in-place execution without modifications. In addition to loop support as described here, we also evaluate the impact of loop unrolling on in-place CRIB-based in-place execution in Chapter 8.

6.3 Additional Loop Support

The CRIB-based Revolver architecture supports all loop bodies detailed in section 3.1. In addition to these supported loop structures, the CRIB-based Revolver architecture also supports loops with continuation statements. Figure 6.2 provides a source code example of a loop that utilizes a continu-

ation statement. The resultant compilation of this code results in assembly code with two branches that redirect to the start of the loop body. The CRIB-based Revolver architecture can handle such loops due to its ability to easily dynamically predicate the execution of instructions. In the last panel of Figure 6.2 we show the loop body unrolled into CRIB by a factor of two. In situations where the continue is not taken, all instructions repeatedly execute. If any non-loop end branch resolves to the loop start address, the Revolver architecture simply dynamically predicates all instructions until the next loop start instruction. The figure shows this when the continue is taken during the first iteration. Dynamically predicated instructions are shown in grey. Support for early continuation statements is unique to the CRIB-based Revolver architecture as supporting them in the PRF-based design is untenable. Supporting dynamic predication in a conventional out-of-order would require architectural support and source operands similar to Wish Branches [48].

6.4 Load/Store Support

In this section we describe the differences between the baseline CRIB load/store queue (LSQ) and a conventional out-of-order processor's LSQ. After describing these differences we elaborate on the necessary changes to enable in-place execution utilzing a similar address banked LSQ.

CRIB Load/Store Queue

The original CRIB architecture utilized an position-insensitive, address banked load/store buffer, in contrast to a conventional load/store queue (LSQ) [32, 31]. In conventional architectures, LSQs are maintained as logically FIFO-like, monolithic structures where entries are dynamically allocated in-order upon instruction dispatch and reclaimed in an in-order fashion upon commit. The in-order and logical placement of loads/stores within their respective queues enables simple logic for searching to identify potential situations where load forwarding or bypassing is allowable.

In contrast to a conventional LSQ, CRIB's LSQ foregoes in-order allocation and deallocation. Similar to a conventional out-of-order, instructions maintain a monotonically increasing store color to maintain ordering between loads and stores. Within the load and store queues however there is no correlation between load/store entry placement and instruction age. Instead of allocating LSQ entries in-order upon instruction dispatch, entries are allocated out-of-order as memory instructions execute. The store color assigned at dispatch is utilized for age-based ordering resolution along with wrapping comparator logic from [15]. To clarify operation we now step through the execution of load and store instructions with an position-insensitive LSQ.

Allocation of loads in the CRIB architecture happens upon load execute. Through execution the load computes the effective address of required

data. CRIB's LSQ is banked on a cache line basis, thus the lower bits of the effective address are utilized to determine the required LSQ bank. After bank determination, an empty load queue entry is allocated at random from the selected bank. If no entry is free, one is reclaimed from a younger load. In the event no younger load exists, the load stalls until an entry becomes available. Loads which have their entries reclaimed, and thus invalidated, are required to re-execute. In parallel with allocation, the load's store color and effective address are used to probe the corresponding store queue bank for potential aliasing and forwarding.

The operation of stores is performed similarly to loads. Upon successful store queue entry allocation, the corresponding load queue bank is probed to determine if any younger loads that alias with the given store have executed. After stores execute and commit, entries are maintained until stores successfully drain to memory.

The position-insensitive, banked nature, and ease of instruction re-execution in CRIB's LSQ enable smaller load and store queues than in conventional architectures. Smaller sized queues are possible because memory instructions only hold instructions from the time of execution until retirement, unlike traditional designs which require LSQ entries be obtained at dispatch. These smaller queues enable area and power savings in the CRIB design. To retain these benefits, in the next subsection we investigate how the existing LSQ structure could be modified to support in-place loop execution.

Store Color	I-Stream	
0	ld r6, [r7]	Before Loop
1	st r8, [r9]	
1	ld r0, [r1, r3]	Iteration #1
2	st r0, [r2, r3]	
	add r3, r3, #1	
	cmp r0, #0	
	bne	
2	ld r0, [r1, r3]	Iteration #2
3	st r0, [r2, r3]	
	add r3, r3, #1	
	cmp r0, #0	
	bne	
3	ld r0, [r7]	After Loop
4	st r1, [r5]	

(a) Conventional Memory Ordering

Loop Color	Iter. Count	Store Color	I-Stream	
0	0	0	ld r6, [r7]	Before Loop
0	0	1	st r8, [r9]	
1	0	1	ld r0, [r1, r3]	Iteration #1
1	0	2	st r0, [r2, r3]	
			add r3, r3, #1	
			cmp r0, #0	
			bne	
1	1	1	ld r0, [r1, r3]	Iteration #2
1	1	2	st r0, [r2, r3]	
			add r3, r3, #1	
			cmp r0, #0	
			bne	
2	0	3	ld r0, [r7]	After Loop
2	0	4	st r1, [r5]	

(b) Loop Memory Ordering

Figure 6.3: Memory Ordering

In-place Execution Modifications

Although the LSQ structure derived for in-place execution in Section 4.5 could be utilized, such a design would forego the benefits of CRIB's LSQ design. As LSQ entries in CRIB are dynamically allocated at execute, rather than dispatch, the primary modification necessary to support in-place execution is the modification of age-based ordering logic to properly represent the relation between memory instructions.

In Figure 6.3, we show how memory instruction's *colors* are assigned for conventional LSQ's as well as for in-place execution CRIB. The code example consists of a simple loop that iterates twice along with instructions from before and after the loop. As seen in Figure 6.3a, store colors alone

are sufficient to provide ordering between memory instructions. Store colors are easily supported by simply increasing a monotonic counter with wrapping logic in the processor front-end upon every store encountered. This however is not sufficient for the CRIB-based Revolver architecture. This is because, as shown in Figure 6.3b, the instructions from different iterations of a loop share the same store color. To differentiate the age across loop iterations, additional information is required. Thus, whenever a load or store execute and require allocation, the loop iteration count is also sent along to the LSQ for ordering purposes. As shown in the figure, this would allow proper ordering of the memory instructions from the first and section loop iteration. A machine's loop iteration identifier field must be sufficiently large enough to handle as many stores may be present within an LSQ bank with conventional wrapping logic. Due to the small size of the LSQ banks in CRIB, this field is quite small however. Lastly shown in the example, there must be proper ordering of memory instructions from within a loop to those after a loop. As non-loop instructions do not iterate, with iteration counters and store colors alone, it would be possible to mis-order the last load in the example with a load from the second iteration of the loop. To solve this issue, the processor front-end also assigns a loop color along with a store color. The loop color is simply an identifier that is montonically incremented upon ever loop dispatch attempted by the processor front-end.

Through these simple additions to the store color identification logic, it

is possible to support in-place execution with the existing LSQ design. It should also be noted that unlike the conventional out-of-order based LSQ, there is no need to immediately free store queue entries upon commit by draining them into a store buffer. Instead of draining into a store buffer, new store queue entries can be dynamically allocated for future iteration stores alongside existing stores.

Load Pre-Execution Support

In the conventional out-of-order based Revolver architecture load pre-execution is supported through a special buffer to store pre-executed load data, as shown in Figure 4.9. In the CRIB-based Revolver architecture this additional buffer is unnecessary. Instead load pre-execution is supported through detection logic associated with each CRIB entry and the existing LSQ. When the pre-execution logic detects a supported address pattern, a load queue entry is speculatively allocated in parallel for the next loop iteration. The only difference between the two load queue entry allocations are the associated iteration counter. Once the next iteration of a given load instruction executes, the existence of a speculative load queue entry is realized. Upon execution, the speculative load's address is verified and the speculative load queue entry is claimed, thus becoming non-speculative. Prior to being claimed and made non-speculative, a speculative load may become invalidated for two primary reasons: coherence events and older loads reclaiming the entry. In either of these events, the next iteration's

load simply executes normally. Upon loop termination, speculatively allocated load queue entries for extraneous iterations are invalidated.

6.5 Conclusion

In this chapter we have given an overview of how in-place loop execution can effectively be supported on an CRIB-based out-of-order processor. The design as proposed utilizes the existing CRIB operand network. In Chapter 7 we evaluate the impact of operand network design and the impact of operand network limitations on loop-mode execution.

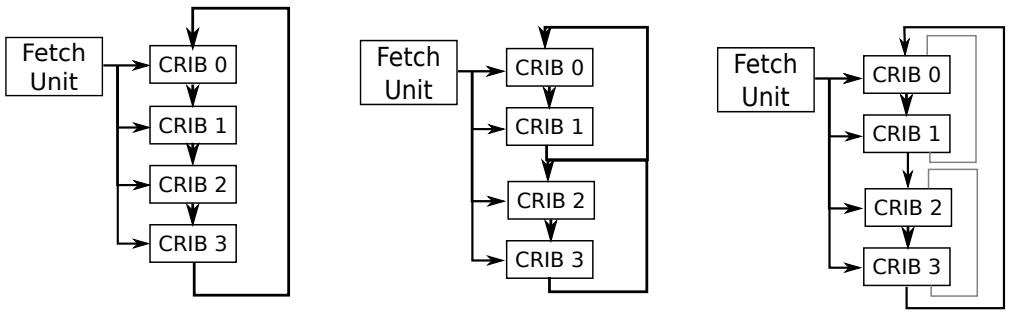
7 OPERAND NETWORK

This chapter investigates operand network design parameters and their impact on in-place loop execution. Additionally the potential performance benefit from an optimized operand interconnection network ins analyzed.

7.1 Overview

Supporting in-place execution of loops on the CRIB architecture involves multiple complications and necessary design modifications. When considering how to support in-place execution within the CRIB instruction window, multiple design alternatives in how to propagate operand values across loop iterations arise.

Shown in Figure 7.1a, CRIB's existing operand network is structured as a unidirectional ring. The simplest means of supporting in-place loop



a) No Loop Support b) Full-Width Loop c) Partial Loop Bypass

Figure 7.1: CRIB Alternative Operand Networks.

execution on CRIB is to utilize the existing operand network and unroll loops to fill as much of the instruction window as possible. Unutilized entries would be set as no-ops with register operands routed past along the ring network.

Utilizing an alternative network with additional bypass paths like Figure 7.1b and 7.1c, would allow performance improvements by rapidly routing around unused entries. Additionally, bypass networks could be simplified like in Figure 7.1c to carry only the loop carried dependencies rapidly rather than all operands like in Figure 7.1b.

This chapter contains a design space exploration that was conducted to evaluate the requirements of such bypass networks and the potential benefit that could be obtained from such networks. In the design exploration phase it was determined that such network modifications would result in small performance gains at the cost of additional design complexity.

7.2 Loop Carried Dependencies

Although loops within programs are expressed sequentially, loops are often largely independent with few communicated values between iterations. This property of loops was largely utilized in speculative multithreading, and can be utilized to simplify a network for in-place loop execution [81].

In Figures 7.2 and 7.3, the impact of limited allowed dependencies on in-place execution on the number of instruction dispatches are shown.

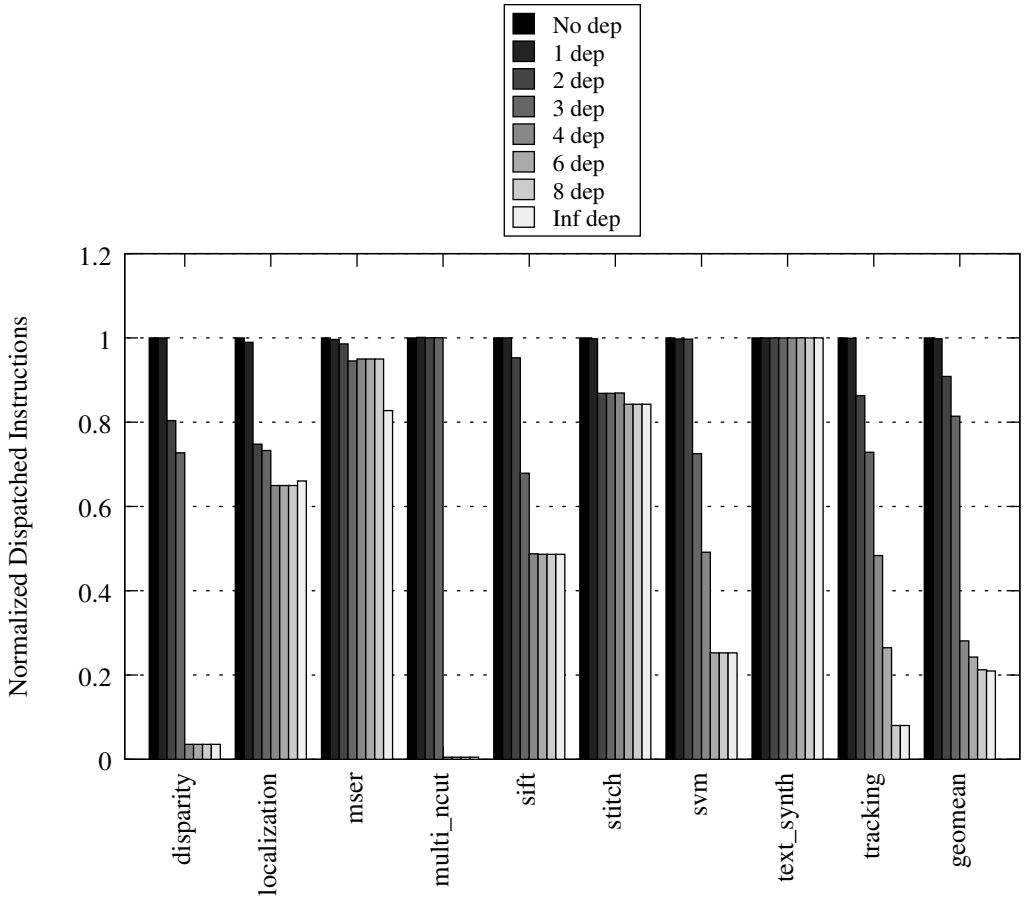


Figure 7.2: SD-VBS Dispatched Instructions vs. Bypass Dependencies.

The CRIB-based Revolver architecture, with parameters as specified in Chapter 8 are utilized.

In Figure 7.2 the reduction in number of dispatched instructions versus the number of allowed loop carried dependencies are shown for SD-VBS. The number of allowed dependencies is varied between zero and infinite allowed dependencies. It is observed that the majority of instruction dis-

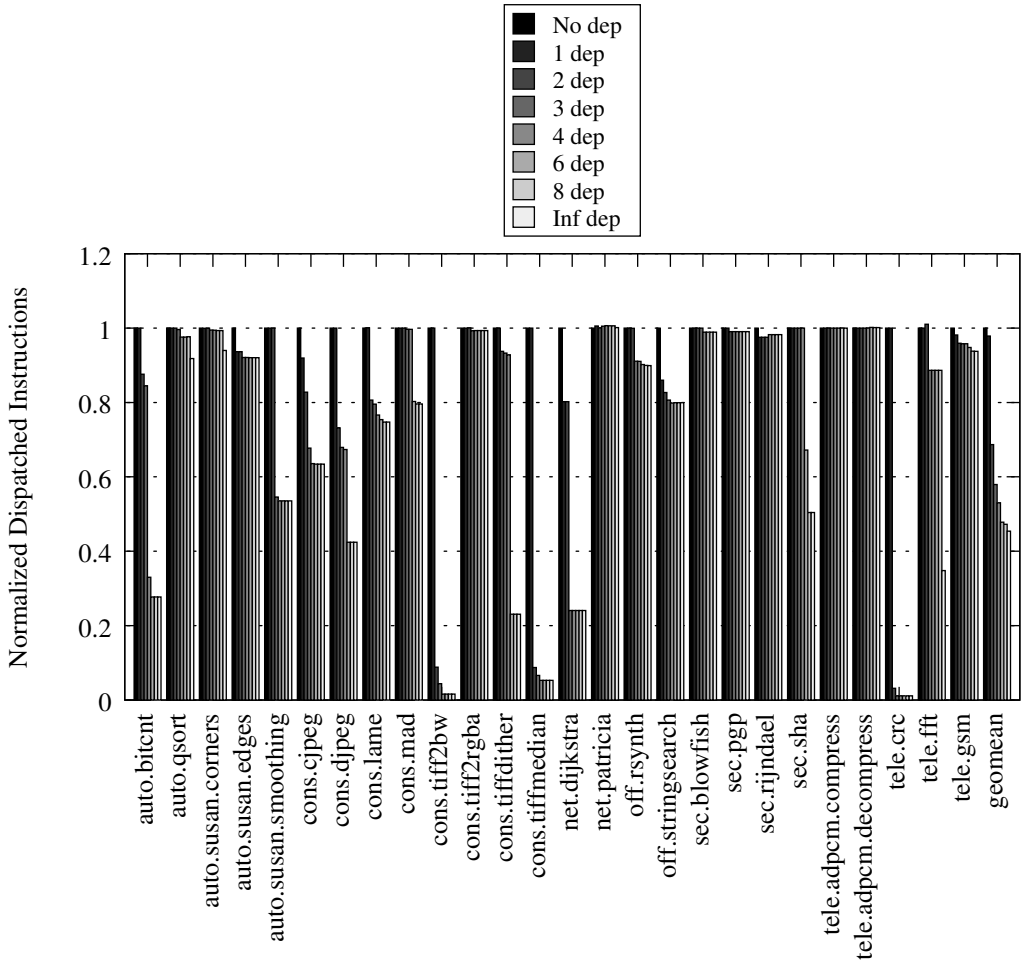


Figure 7.3: MiBench Dispatched Instructions vs. Bypass Dependencies.

patch benefit can be obtained by a bypass operand network that supports four or more loop carried dependencies.

Figure 7.3 shows how many instruction dispatches can be saved with limited allowed bypassed operands through in-place loop execution on the MiBench suite. Some benchmarks like, *tiff2bw* and *tele.crc*, sucessfully

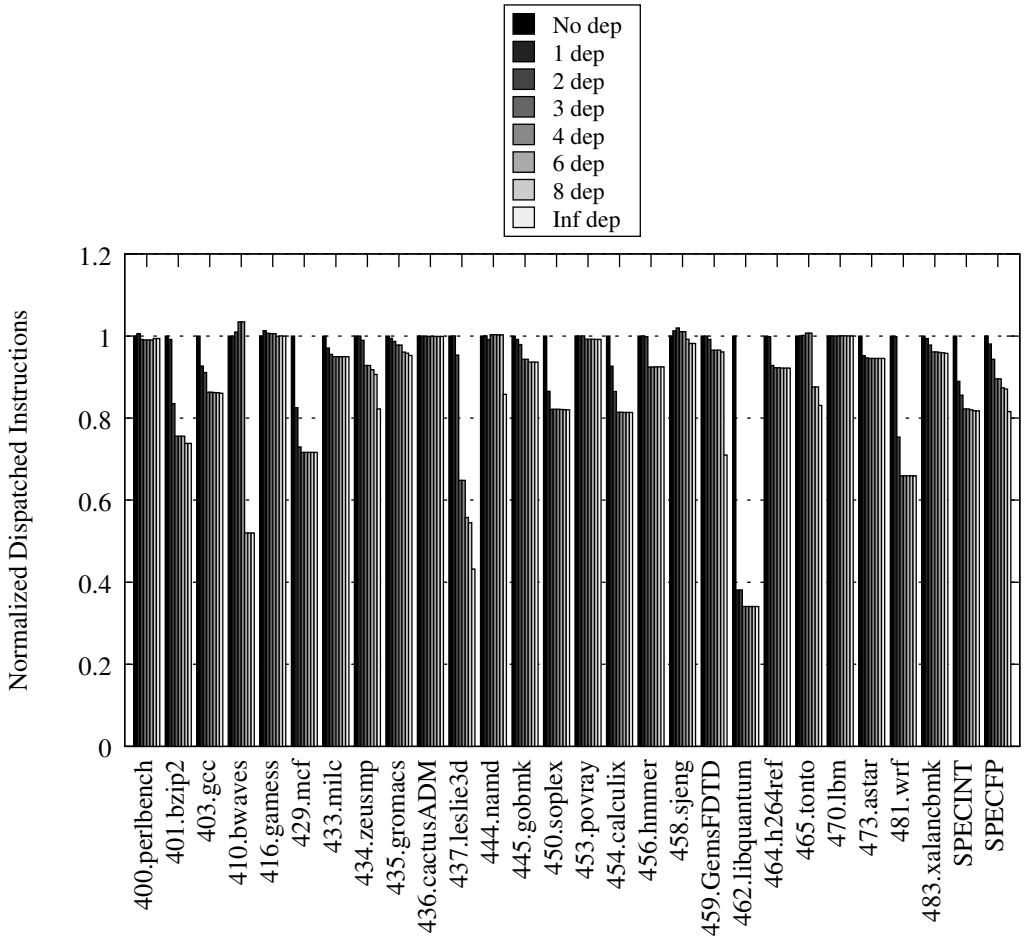


Figure 7.4: SPEC CPU2006 Dispatched Instructions vs. Bypass Dependencies.

eliminate the majority of instruction dispatches with only two allowed bypass paths. However to eliminate the majority of instruction dispatches, operand networks supporting four or more loop carried dependencies are required.

Finally, Figure 7.4, shows the dispatch reductions obtained through

supporting more loop carried dependencies on the SPEC2006 benchmark suite. In comparison to SD-VBS and MiBench, the SPECINT suite has fewer loop carried dependency requirements. Most benefit for the floating point subset is obtained with support for three loop carried dependencies.

With the impact of limited loop carried dependencies determined, the following section evaluates the potential benefit from interconnection network latency optimization.

7.3 Operand Latency

The primary goal of adding lower-latency bypass paths for loop execution is to improve performance during loop execution. To determine the benefit from faster operand networks, Figures 7.5, 7.6, and 7.7 show execution latencies of SD-VBS, MiBench, and SPEC CPU2006 with different speed operand networks. All designs are normalized against a baseline operand routing speed of 8 CRIB entries per cycle.

Figure 7.5, shows the reduction in execution time from increasing operand propagation to 12 or 16 entries per cycle on SD-VBS. Increasing operand propagation by 50% to 12 entries per cycle results on average in a 1.5% performance improvement, whereas doubling operand network speed to 16 entries per cycle results in a 1.8% improvement in overall performance.

Figure 7.6 shows the impact of operand network latency on execution

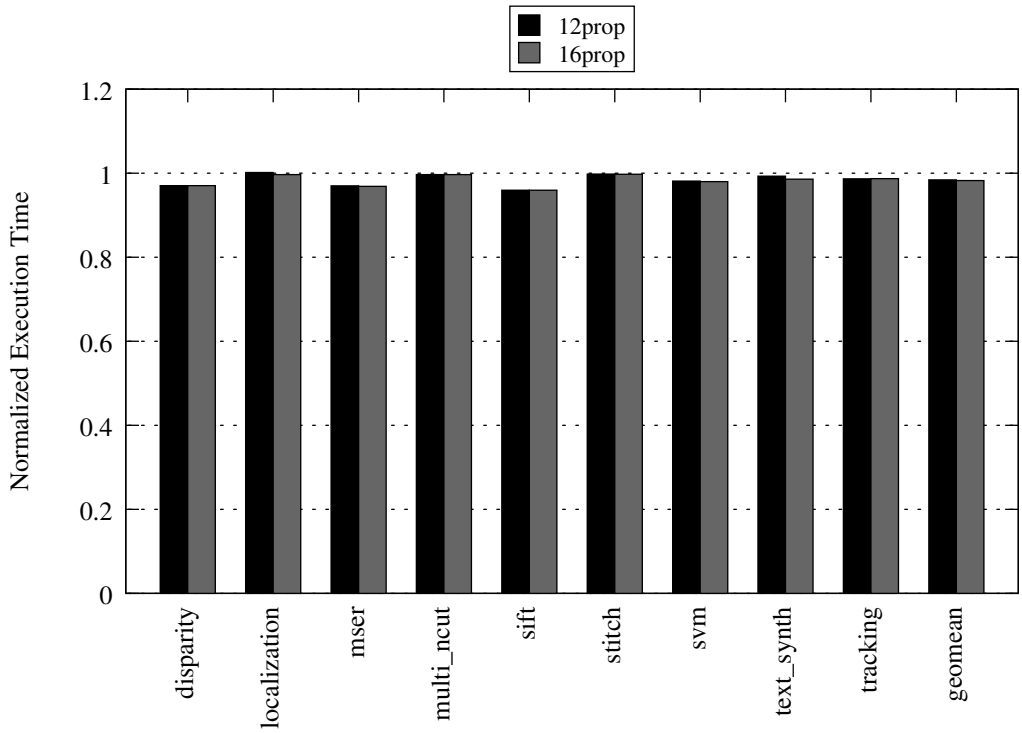


Figure 7.5: SD-VBS Bypass Latency Sensitivity.

time for the MiBench suite. Increasing the propagation rate to 12 or 16 entries per cycle improves overall performance by 1.4% and 1.6% respectively. The *auto.susan.smoothing* obtains the most benefit lessening execution time by 11.2% with a faster operand network.

Finally, Figure 7.7 shows the benefit from lower operand network latency on execution time for SPEC CPU2006. Overall the SPECINT workloads improve execution time by 1.1% when executed on an operand network with half the implementable latency. SPECFP workloads obtain more benefit, with up to a 1.7% increase in overall performance.

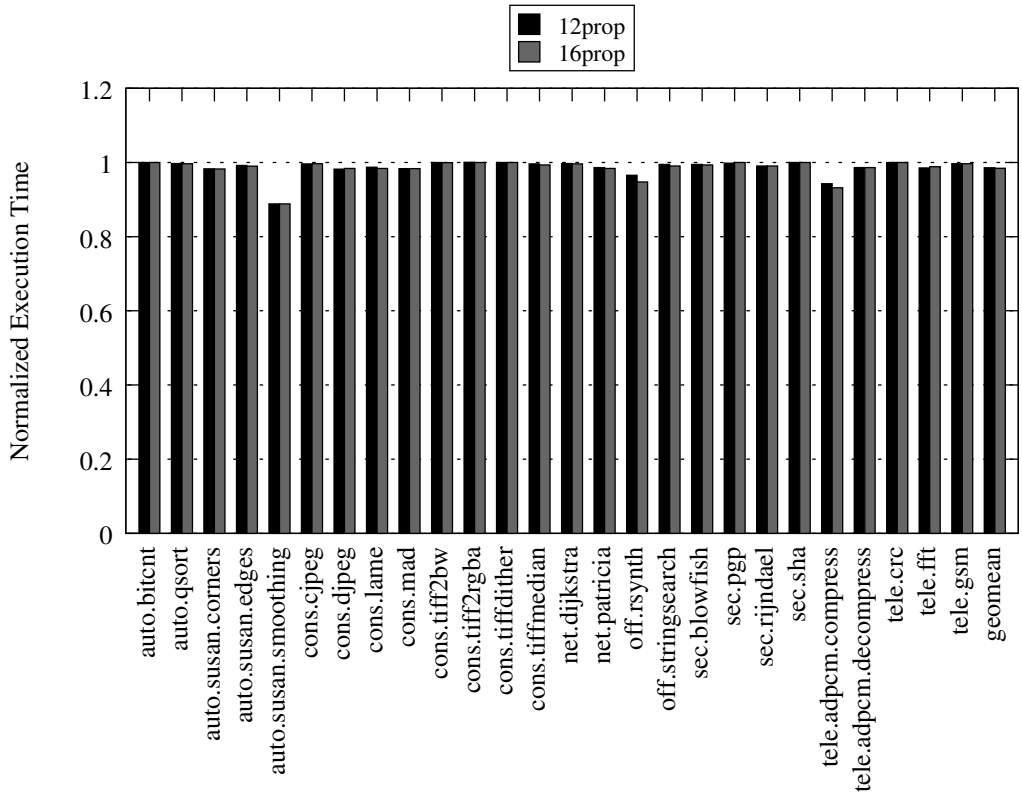


Figure 7.6: MiBench Bypass Latency Sensitivity.

7.4 Summary

Overall it is observed that an new network, with half the latency, would offer less than 2% performance benefit over the current implementation. Due to the additional multiplexing required by the earlier bypass alternatives, it was determined that it was such alternative operand network structures were unlikely to prove beneficial once fully realized.

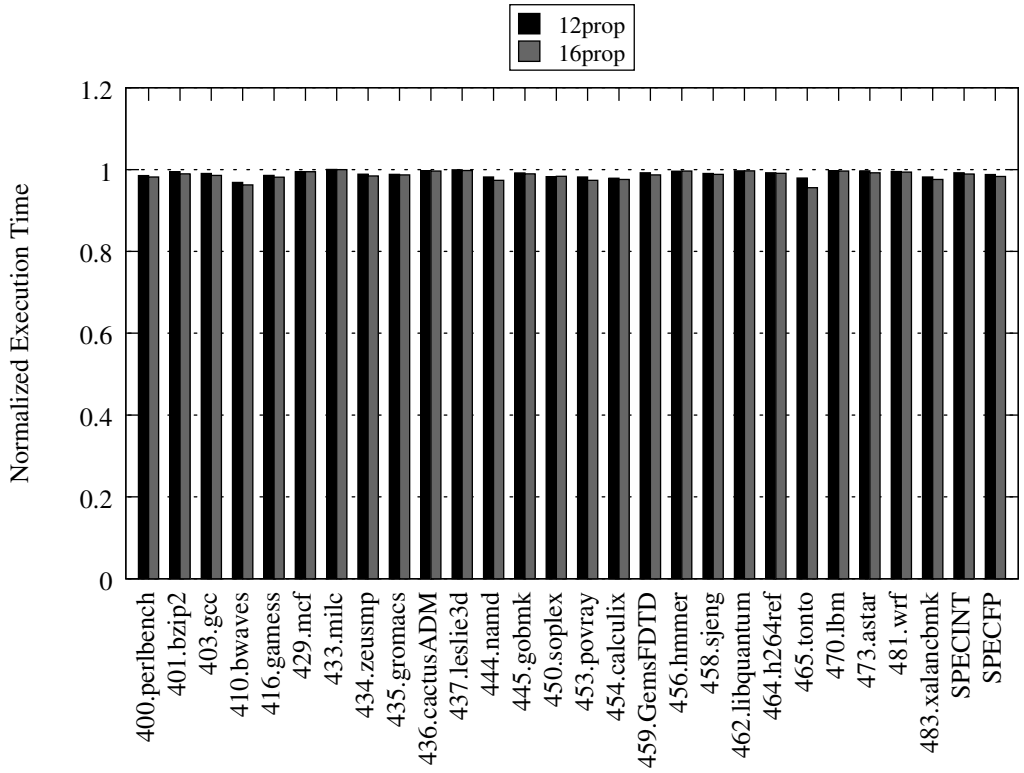


Figure 7.7: SPEC CPU2006 Bypass Latency Sensitivity.

8 EVALUATION

To evaluate the Revolver architectures, a combination of cycle accurate simulation and power modeling was utilized across a wide variety of benchmarks. Through the respective analysis tools, the overall effectiveness, possible design tradeoffs, and as well as the detailed microarchitectural impacts of in-place loop execution are presented. In this chapter the physical and performance modeling methodology is presented. Following the methodology, a thorough analysis of in-place execution on both the PRF-based and CRIB-based out-of-order processors is presented.

8.1 Methodology

Performance Simulation

All performance evaluation was performed within the gem5 simulator infrastructure [13]. Within the gem5 framework, custom cycle-accurate out-of-order core models were implemented to simulate the PRF-based and CRIB-based out-of-order processors.

For the PRF-based out-of-order evaluation, the two baseline configurations shown in Table 8.1 were used for comparison. The *OO2* configuration is a small 2-wide out-of-order processor configured similarly to the recently announced Intel Silvermont architecture [46]. The *OO4* design represents a more aggressive 4-wide architecture with the window size and execution

	OO2, Small Out-of-Order	OO4, Large Out-of-Order
Branch Predictor	Combined bimodal (16k entry) / gshare (16k entry), selector (16k entry), 32 entry RAS, 2k BTB	
Core	2GHz, 4-wide fetch/commit, 6-wide issue, 32 ROB/IQ, 12 LQ, 8 SQ, 8 WB, 48 Int PRF, 64 FP PRF, aggressive memory speculation, 13-stage pipeline	2GHz, 4-wide fetch/commit, 8-wide issue, 64 ROB/IQ, 24 LQ, 16 SQ, 8 WB, 80 Int PRF, 96 FP PRF, aggressive memory speculation, 13-stage pipeline
Functional Units	2 Int ALU (1-cycle), 1 Int Mult/Div (3-cycle/20-cycle), 1 LD (1-cycle AGU), 1 ST (1-cycle), 2 SIMD units (1-cycle), 2 FP Add/Mult (5-cycle), 1 FP Div/Sqrt (10-cycle)	3 Int ALU (1-cycle), 1 Int Mult/Div (3-cycle/20-cycle), 2 LD (1-cycle AGU), 1 ST (1-cycle), 2 SIMD units (1-cycle), 2 FP Add/Mult (5-cycle), 1 FP Div/Sqrt (10-cycle)
Memory System	L1 ICache 32KB, 2-way, 64B (2-cycle), 2-ahead sequential prefetch L1 DCache 32KB, 4-way, 64B (3-cycle), 2-ahead stride prefetch L2 Unified 256KB, 8-way, 64B (12-cycle), 2-ahead comb. prefetch L3 Unified 4MB, 16-way, 64B (24-cycle), 4-ahead comb. prefetch Off-Chip Memory: 2GB DDR3-1600	

Table 8.1: Common Processor Configurations.

resources scaled up from the OO2 configuration. Two PRF-based Revolver designs are compared against these baselines, a 2-wide (*Rev2*) and 4-wide (*Rev4*) configuration. All designs utilize aggressive memory systems with prefetchers at every cache level. Unless otherwise specified, all Revolver designs incorporate the load pre-execution optimization where up to 8 loads may be pre-executed.

For the CRIB-based out-of-order evaluation, an aggressive 64-instruction window configuration, presented in Table 8.2, was utilized. The increased physical resources and differing operand result latency from the original

Branch Predictor	Combined bimodal (16k entry) / gshare (16k entry), selector (16k entry), 32 entry RAS, 2k BTB
Core	2GHz, 4-wide fetch/commit, 64-entry CRIB, 8 entries/cycle operand propagation, 2 LSQ banks, 16 Loads/8 Stores per LSQ bank, aggressive memory speculation, 13-stage pipeline,
Functional Units	Int ALU (1-cycle), 1 Int Mult/Div (3-cycle/20-cycle), 2 LD/cycle (1-cycle AGU), 1 ST/cycle (1-cycle), 2 SIMD units (1-cycle), 2 FP Add/Mult (5-cycle), 1 FP Div/Sqrt (10-cycle)
Memory System	L1 ICache 32KB, 2-way, 64B (2-cycle), 2-ahead sequential prefetch L1 DCache 32KB, 4-way, 64B, (3-cycle), 2-ahead stride prefetch, 8-internal banks, 2-cycle bank row hit latency L2 Unified 256KB, 8-way, 64B (12-cycle), 2-ahead comb. prefetch L3 Unified 4MB, 16-way, 64B (24-cycle), 4-ahead comb. prefetch Off-Chip Memory: 2GB DDR3-1600

Table 8.2: CRIB Processor Configuration

CRIB proposal [32] result from moving between the 65nm to 28nm process technology nodes.

For competitive baselines, each processor configuration can optionally be equipped with a 32-μop loop buffer (*LB*) or a 1.5K μop cache (μ*C*), similar to recent Intel and ARM designs [45, 46, 52, 72].

A wide variety of applications from the San Diego Vision Benchmark Suite (SD-VBS), MiBench, and SPEC2006 were used for simulation [33, 36, 87]. All applications were compiled for the ARMv7 ISA on gcc 4.7.2 with full optimizations (*-O3*), vectorization, and link-time optimization(*-flto*) enabled. On SD-VBS, simulation was limited to the instrumented regions of interest. For MiBench, entire applications were simulated on the *large* input set. Finally for SPEC2006, a SimPoint simulation methodology was

Benchmark	Input Set	OO2 / OO4 IPC	OO2 / OO4 μpc
disparity	cif	1.564 / 2.368	1.654 / 2.505
localization	vga	0.807 / 1.165	1.446 / 2.086
mser	vga	1.110 / 1.449	1.124 / 1.467
multi_ncut	sim_fast	1.305 / 1.427	1.306 / 1.428
sift	qcif	1.628 / 1.921	1.643 / 1.939
stitch	qcif	1.598 / 2.612	1.709 / 2.793
svm	qcif	1.363 / 1.918	1.578 / 2.221
text_synth	vga	1.340 / 1.890	1.480 / 2.089
tracking	fullhd	1.115 / 1.571	1.245 / 1.755

Table 8.3: SD-VBS Benchmarks and Baseline Performance

employed, resulting in the suite being represented by 177 100M instruction simulation points [35]. The *train* input set was utilized for SPEC2006. Baseline performance for all benchmark suites are shown in Tables 8.3, 8.4, and 8.5.

Three different benchmark suites were chosen in order to evaluate differing workloads. Traditional computing is represented through the SPEC CPU2006 benchmark suite, whereas MiBench and SD-VBS represent embedded and emerging classes of processor workloads.

Physical Modeling

Power modeling was performed through a correlated and extended version of the McPAT power simulator [55]. In addition to the baseline McPAT simulator, microarchitectural models for loop buffers and μop caches were added in order to enable proper energy accounting. For energy

Benchmark	**OO2 / OO4 IPC**	**OO2 / OO4 μpc**
auto.bitcnt	1.526 / 1.959	1.607 / 2.064
auto.qsort	1.145 / 1.583	1.421 / 1.964
auto.susan.corners	1.637 / 2.491	1.652 / 2.513
auto.susan.edges	1.619 / 2.501	1.642 / 2.537
auto.susan.smoothing	1.827 / 2.440	1.828 / 2.441
cons.cjpeg	1.517 / 2.428	1.610 / 2.577
cons.djpeg	1.625 / 2.600	1.723 / 2.758
cons.lame	1.228 / 1.622	1.331 / 1.757
cons.mad	1.598 / 2.272	1.653 / 2.350
cons.tiff2bw	1.509 / 2.294	1.942 / 2.952
cons.tiff2rfba	1.399 / 2.169	1.506 / 2.336
cons.tiffdither	1.467 / 2.481	1.600 / 2.706
cons.tiffmedian	1.511 / 2.591	1.662 / 2.849
net.dijkstra	1.176 / 1.522	1.262 / 1.634
net.patricia	1.161 / 1.659	1.426 / 2.039
off.rsynth	1.299 / 1.522	1.371 / 1.607
off.stringsearch	1.299 / 1.895	1.559 / 2.251
sec.blowfish	1.309 / 1.681	1.496 / 1.921
sec.pgp	0.768 / 0.881	0.804 / 0.922
sec.rijndael	1.677 / 2.758	1.778 / 2.924
sec.sha	1.736 / 2.778	1.837 / 2.942
tele.adpcm.compress	1.571 / 1.679	1.623 / 1.734
tele.adpcm.decompress	1.486 / 2.334	1.557 / 2.445
tele.crc	0.839 / 0.877	1.027 / 1.073
tele.fft	1.486 / 2.112	1.596 / 2.269
tele.gsm	1.456 / 1.688	1.486 / 1.722

Table 8.4: MiBench Benchmarks and Baseline Performance

Benchmark	OO2 / OO4 IPC	OO2 / OO4 μpc
400.perlbench	1.004 / 1.260	1.275 / 1.600
401.bzip2	1.377 / 2.044	1.466 / 2.179
403.gcc	0.890 / 1.168	1.148 / 1.505
410.bwaves	1.104 / 1.452	1.117 / 1.469
416.gamess	1.559 / 2.319	1.590 / 2.366
429.mcf	0.641 / 0.729	0.653 / 0.746
433.milc	0.613 / 0.782	0.625 / 0.796
434.zeusmp	1.121 / 1.563	1.226 / 1.710
435.gromacs	1.325 / 1.949	1.341 / 1.971
436.cactusADM	1.659 / 2.808	1.711 / 2.897
437.leslie3d	1.042 / 1.390	1.256 / 1.677
444.namd	1.281 / 1.765	1.383 / 1.911
445.gobmk	0.946 / 1.188	1.121 / 1.408
450.soplex	0.841 / 1.035	0.895 / 1.104
453.povray	0.898 / 1.152	1.237 / 1.586
454.calculix	1.339 / 1.650	1.420 / 1.751
456.hmmer	1.716 / 2.739	1.797 / 2.869
458.sjeng	1.067 / 1.390	1.193 / 1.554
459.GemsFDTD	1.154 / 1.371	1.246 / 1.479
462.libquantum	1.523 / 2.551	1.580 / 2.646
464.h264ref	1.680 / 2.800	1.745 / 2.909
465.tonto	1.488 / 2.028	1.618 / 2.204
470.lbm	1.072 / 1.179	1.072 / 1.179
471.omnetpp	0.921 / 1.143	1.230 / 1.527
473.astar	1.019 / 1.276	1.026 / 1.284
481.wrf	1.130 / 1.578	1.437 / 2.042
483.xalancbmk	0.910 / 1.164	1.264 / 1.616

Table 8.5: SPEC CPU2006 Benchmarks and Baseline Performance

estimation, microarchectural event counters were embedded within the gem5 performance simulator. The resulting event frequencies were then utilized by McPAT in the generation of final energy estimates. All later energy comparisons represent core energy, including the L1 caches.

Power model validation was performed through an iterative process. Initial model validation was performed by configuring McPAT and gem5 to emulate an ARM Cortex-A15 microprocessor [52]. After configuration, the SPECINT benchmark suite and used to generate power estimates. These power estimates where compared to published SPECINT power data from nVidia [60]. From these initial energy estimates, it was discovered that McPAT significantly underestimates decode and branch prediction power. Source code level inspection of McPAT revealed that these are partially known issues. To enable realistic modeling, compensation factors were added to bring McPAT's branch prediction and decode energy estimates in line with measured energies. As our power model is correlated with a Cortex-A15, McPAT was configured to model a low power technology node. Low power technology nodes are frequently used in the mobile segment as they prioritize minimizing leakage in exchange for higher dynamic energy costs. This energy prioritization increases the benefit of in-place loop execution as it is primarily a technique to save dynamic energy. If a high performance technology node were utilized, the expected benefit from in-place execution can expected to be less due to proportionally less dynamic energy consumption. It should be noted that most of our

benchmark evaluation, other than the SPEC CPU2006 benchmark suite, utilizes relevant to mobile workloads.

The complexity and power consumption of the PRF-based Revolver architecture's tag propagation unit (TPU) is approximated as an SRAM-based RAT with checkpoints. This approximation is made due to the structural similarities between the two structures. To maintain instruction-level checkpoints, the *OO4* RAT would require 8.75Kbits of storage for the SRAM-based RAT with checkpointing [67]. Equivalently, Revolver's integer TPU is organized as 16 register columns, each carrying a 7-bit register identifier for the entire 64 instruction window. This structural representiation also requires an effective 8.75Kbits of storage. By maintaining a valid architectural register mappings at each instruction boundary, Revolver's TPU complexity ends up comparable to a conventional RAT. Thus, energy savings from TPU usage result from pipeline routing, control, and allocation overheads rather than direct energy reduction.

For the estimation of CRIB power consumption, relative energies for the line-banked data cache and execution core were utilized alongside the McPAT frontend power model. This operation was performed by scaling estimated McPAT execution energies to be representative of those presented in [32].

8.2 Conventional Out-of-Order

This section presents an evaluation of in-place loop execution against conventional out-of-order processor designs. Designs are evaluated in terms of their impact on the numbers of instructions dispatched by the processor front-end, the performance impact of in-place loop execution, energy consumption, and combined energy-delay product.

Instruction Dispatch

Through loop execution mode, the Revolver architecture is capable of eliminating many front-end instruction dispatches. Removing front-end dispatches allows Revolver architectures to save energy, even beyond loop buffers and μop caches, as multiple pipeline stages between decode and execute are elided.

To demonstrate this benefit, Figures 8.1, 8.2, and 8.3 detail the fraction of instructions dispatched by the PRF-based Revolver designs in comparison to traditional out-of-order cores across all three benchmark suites. Each configuration is normalized against the equivalent width out-of-order baseline.

In Figure 8.1, the instruction dispatch results for SD-VBS are presented. As shown earlier in Figure 1.2, SD-VBS benefits the most from loop buffering out of all three benchmark suites. Overall the *Rev2* and *Rev4* processor configurations successfully eliminate 83.7% and 78.7% of all front-end

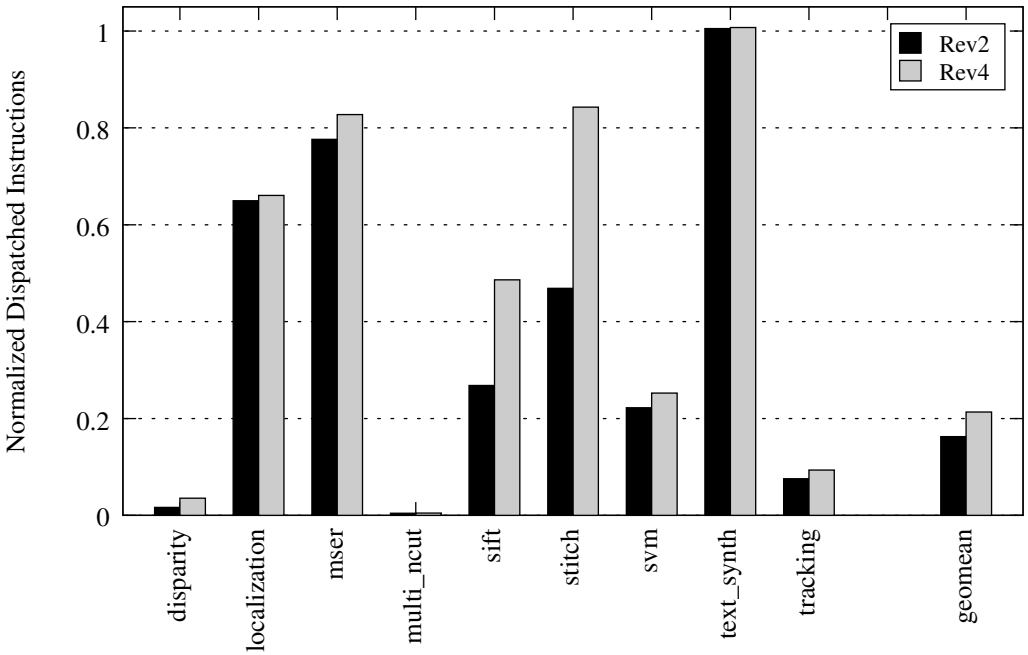


Figure 8.1: SD-VBS Dispatched Instructions

instruction dispatches through loop-mode reuse. The reduced efficiency of instruction dispatch in the *Rev4* configuration is due to additional loop unrolling allowed by the larger instruction queue. The two best performing benchmarks, *disparity* and *multi_ncut* reduce instruction dispatches by 98.4% and 99.6% as they spend almost all execution within simple loops. The only benchmark which fails to achieve benefit is *text_synth*. Revolver increases instruction dispatches by up to 0.7% above the baseline on the *text_synth* benchmark. *text_synth*'s dispatch increase is due to unstable, data-dependent control embedded within the benchmark's innermost loops. This variable control flow quickly results in the disabling of loop-mode dispatch.

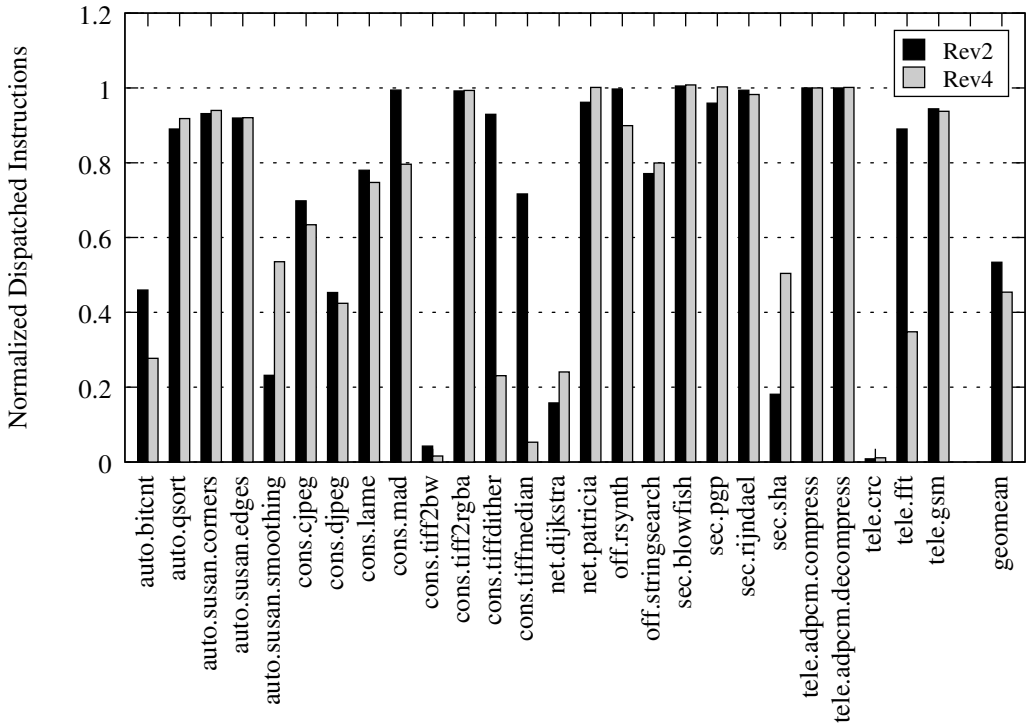


Figure 8.2: MiBench Dispatched Instructions

Instruction dispatch results for MiBench, shown in Figure 8.2, show significantly more variance than those of SD-VBS. Overall the *Rev2* and *Rev4* configurations reduce instruction dispatches by 46.6% and 54.6%. Although the *Rev4* configuration unrolls loops by larger factors than the *Rev2* processor configuration, *Rev4*'s larger instruction window allows the capturing of larger loops on multiple benchmarks. This larger loop handling capability leads to significant reductions on certain benchmarks, primarily *cons.tiffmedian* and *tele.fft*.

The SPEC CPU2006 benchmark suite observes the least benefit from the Revolver architecture's loop-mode dispatch. For the integer workloads,

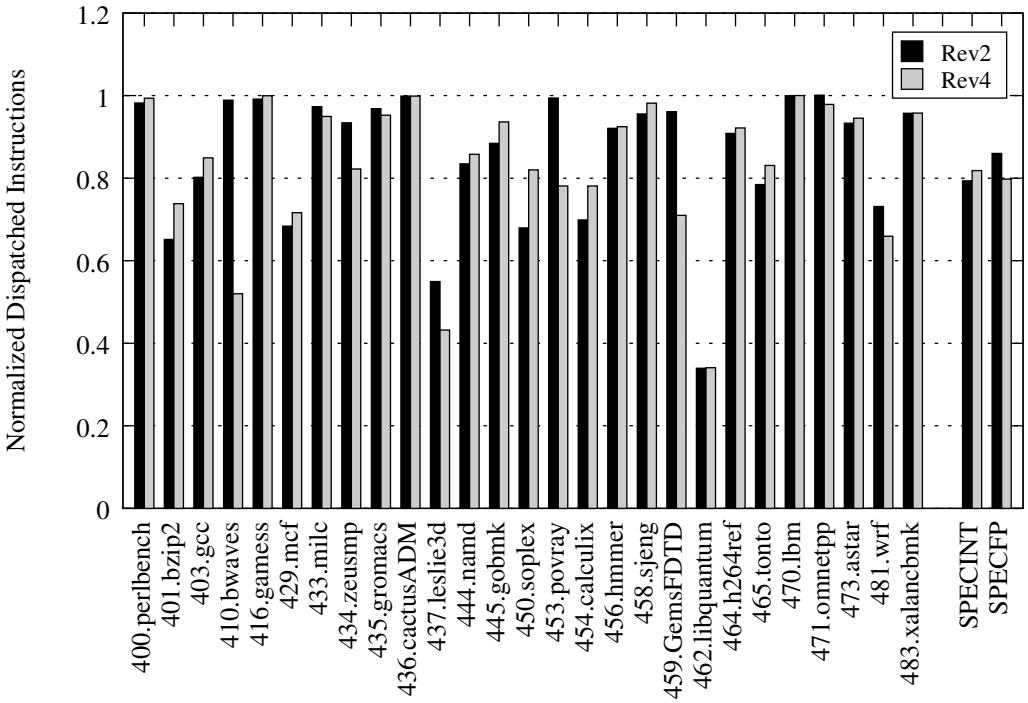


Figure 8.3: SPEC CPU2006 Dispatched Instructions

overall instruction dispatches are reduced by 20.6% and 18.2% for the *Rev2* and *Rev4* configurations. On the floating point workloads, the *Rev2* configuration observes less benefit and reduces instruction dispatches by 14.0%. Shown earlier in Figure 1.2, although most of floating-point execution is spent within simple loops, the loops are of considerable size. This results in in-place execution and traditional loop buffers capturing a small fraction of total loop opportunities due to their limited size. The *Rev4* configuration performs slightly better than *Rev2*, capturing 20.2% of all instruction dispatches.

Performance Impact

This subsection evaluates the performance impact of in-place loop execution in terms of execution latency. As loops are unrolled by even factors into the respective processor issue queues, multiple issue queue entries may go unutilized. Issue queue underutilization during loop-mode dispatch potentially results in reduced performance. Reduced performance is observed when utilizing all issue queue entries would result in additional instruction level parallelism.

In Figures 8.4, 8.5, and 8.6 the present the normalized execution for all processor configurations across the three benchmark suites. All execution latencies are normalized against the *OO2* processor baseline.

On SD-VBS, presented in Figure 8.4, moving from the *OO2* to *OO4* reduces execution time by 26.9% The *Rev2* and *Rev4* processors reduce performance on average by 1.6% and 1.0% against their equivalent width baselines. The smaller performance degradation of the *Rev4* design in comparison to *OO4* is expected due to diminishing returns on large execution windows. The *stitch* bencharm demonstrates the largest performance degradation, hurting performance by 5.3% on the *Rev2* configuration.

In Figure 8.5, the performance results for MiBench are shown. Comparing the baseline designs, the *OO4* configuration offers 28.6% better overall performance than the *OO2* configuration. Overall the *Rev2* and *Rev4* both negatively impact performance against their equivalent width baselines by 0.6%.

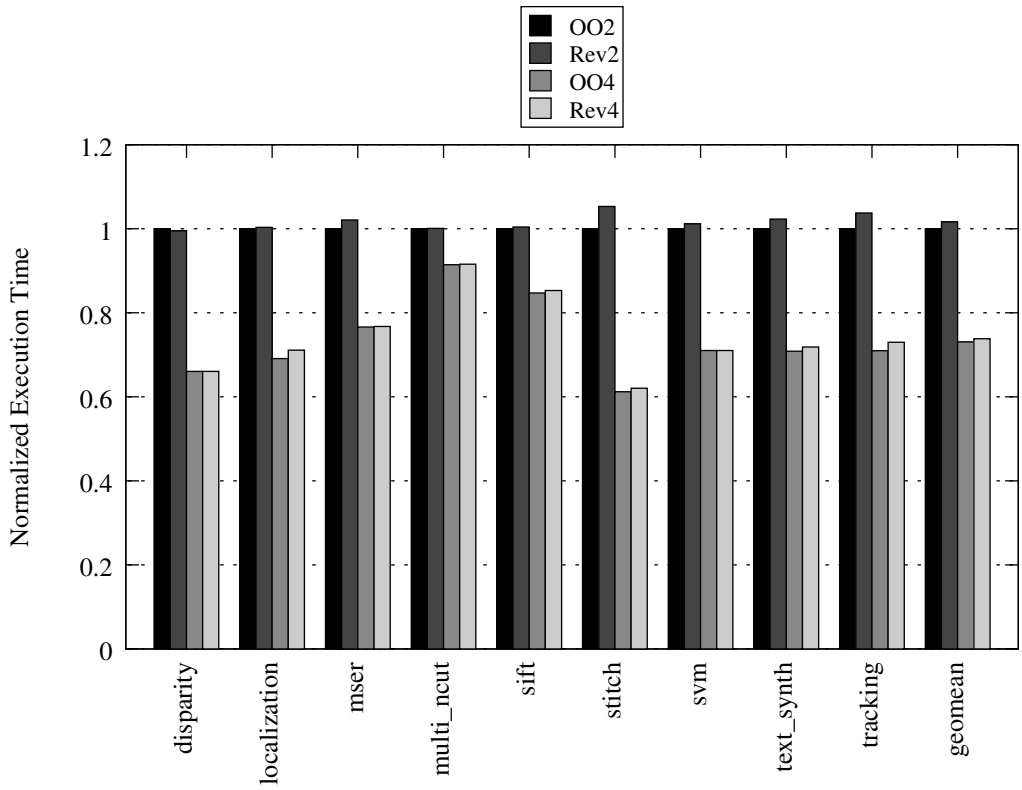


Figure 8.4: SD-VBS Normalized Execution Time

Finally, performance results for the SPEC CPU2006 benchmark suite are shown in Figure 8.6. On the integer workloads, the *OO4* baseline reduces execution time by 26.2% in comparison to *OO2*. The *Rev2* and *Rev4* designs reduce overall performance by 0.3% in comparison to their respective baselines. For the floating point workloads, *OO4* reduces execution time by 24.3% over *OO2*. The *Rev2* and *Rev4* designs hurt overall performace by 0.6% on the floating point workloads. The reduced performance degradation on SPEC is correlated with a reduction in opportunity for

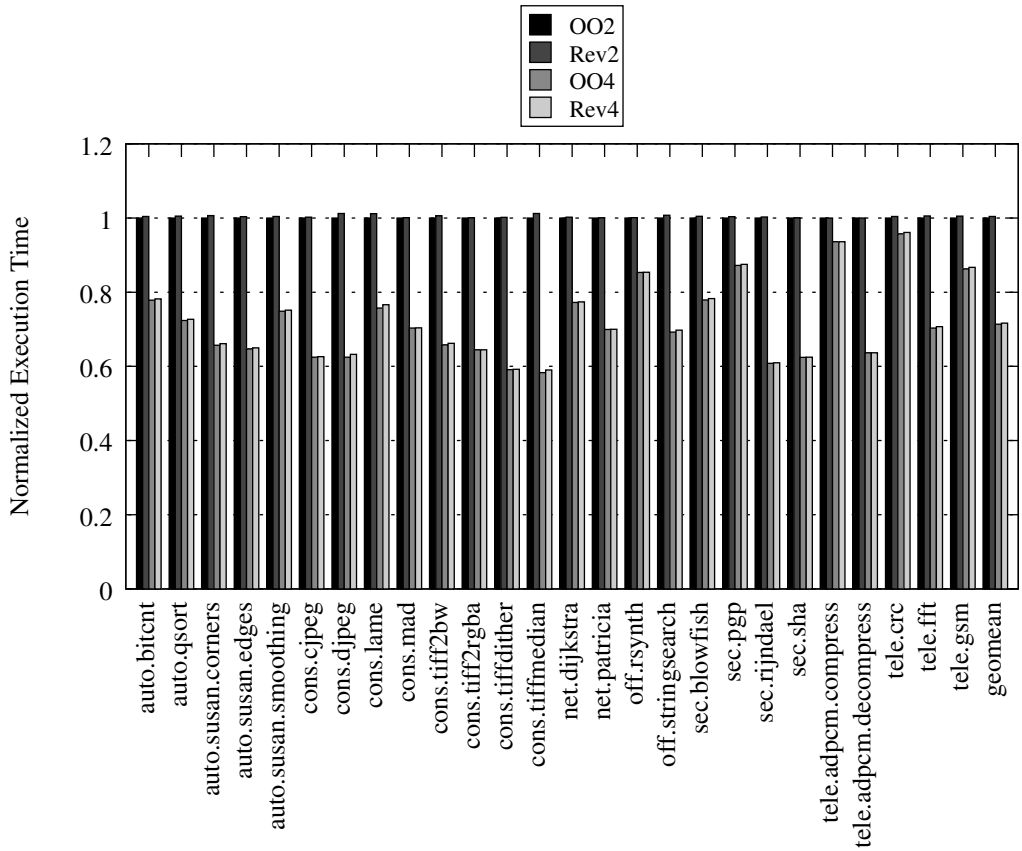


Figure 8.5: MiBench Normalized Execution Time

loop-mode dispatch. The two benchmarks with the largest performance degradation, *437.leslie3d* and *450.soplex* eliminate a significant number of instruction dispatches through loop-mode.

Overall it is observed that in-place loop execution, on average, hurts processor performance by less than 1% while significantly reducing the number of instruction dispatches.

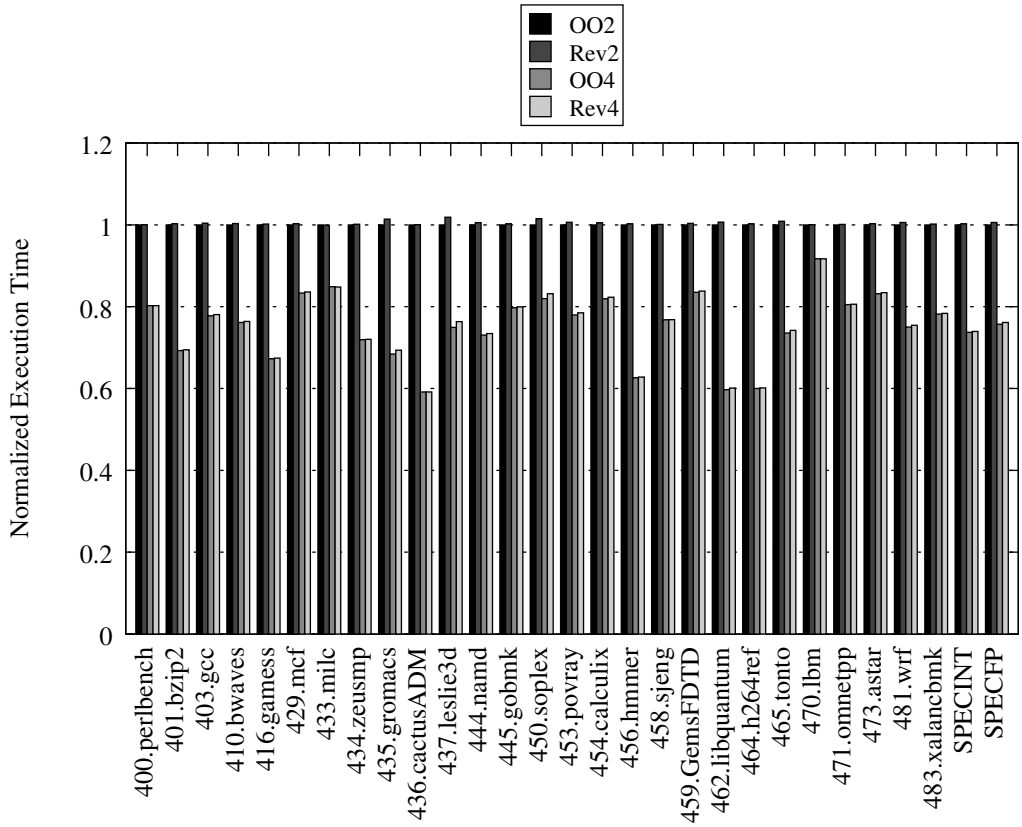


Figure 8.6: SPEC CPU2006 Normalized Execution Time

Energy Impact

This subsection presents the processor core energy estimations obtained by through cycle-accurate simulation and the McPAT power analysis tool. For comparison, configurations of the baseline *OO2* and *OO4* are evaluated with optional loop buffers and μop caches. All energy numbers are normalized against the equivalent width out-of-order baseline without a loop buffer or μop cache.

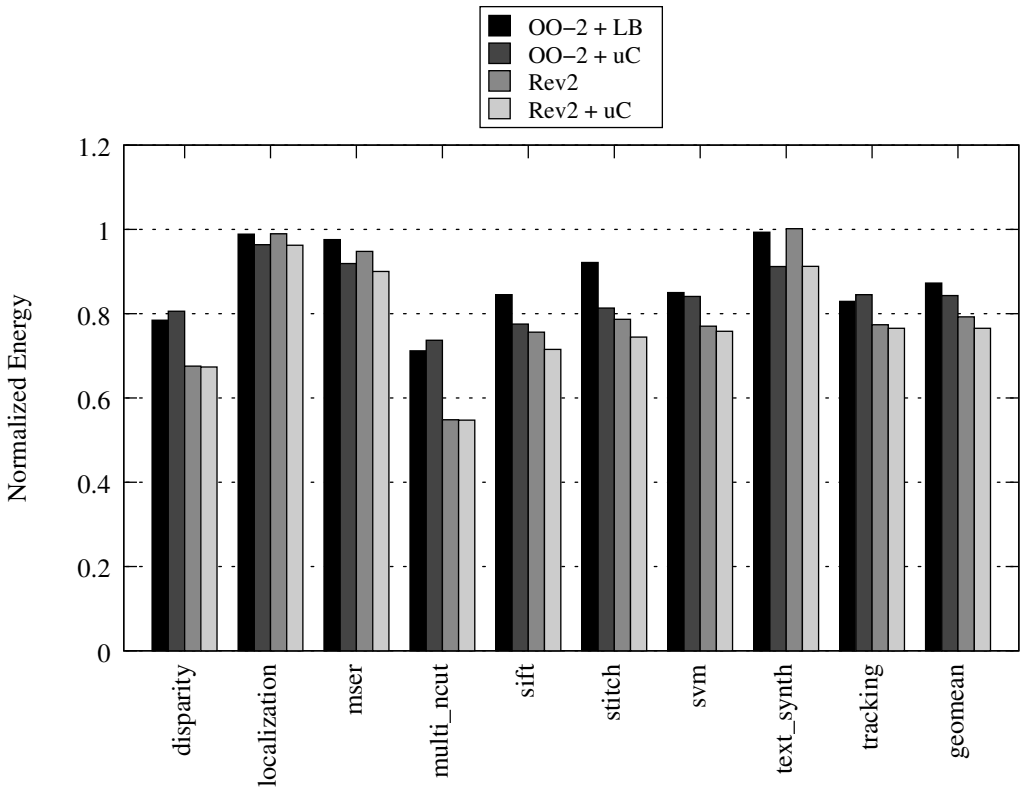


Figure 8.7: SD-VBS 2-Wide Normalized Energy

SD-VBS

Figure 8.7 presents energy consumption for the 2-wide processor configurations on the SD-VBS benchmark suite. With respect to the baselines, the loop buffer and μop cache designs offer significant energy improvements. Although the μop cache design offers the best energy consumption overall, specific benchmarks demonstrate better energy consumption with a small loop buffer. For benchmarks which expend almost all execution time within very simple loops, like *disparity* or *multi_ncut*, the loop buffer

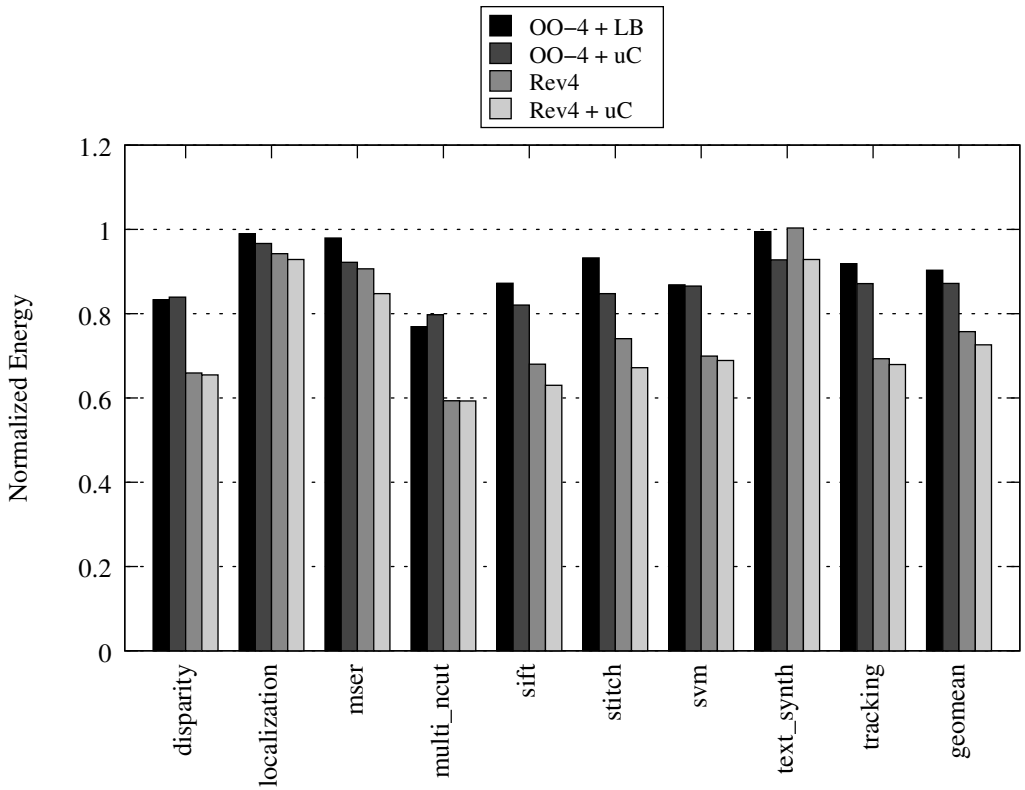


Figure 8.8: SD-VBS 4-Wide Normalized Energy

offers all of the dispatch benefits of a μop cache, but with reduced access energy requirements. The *Rev2* configuration also outperforms the loop buffer equipped out-of-order due to it's ability to handle similar sized loops with reduced energy costs. Due to the high occurrences of simple loops, the Revolver-based designs outperform the conventional out-of-order designs on all benchmarks except for *text_synth*. The *Rev4 + μop cache* configuration offers the best overall energy consumption.

Figure 8.8 shows the energy consumption for the 4-wide issue processor

configurations on SD-VBS. In general the trends from Figure 8.7 remain the same, however the overall energy benefit from front-end energy savings in the Revolver designs is greater than that of the 2-wide configuration. For the conventional loop buffer and μop cache designs, the relative benefit is approximately the same. This energy characteristic is due to the *Rev2* and *Rev4* designs eliminating allocation energy overheads. In general the pre-decode stages energy consumption scales linearly with processor width, while front-end renaming and allocation hardware scales poorly with issue width.

MiBench

Figure 8.9 presents the energy consumption for 2-wide issue processor configurations on the MiBench suite. The μop cache equipped designs, *OO2* + μ*C* and *Rev2* + μ*C*, designs offer the best overall processor energy consumption. This trend is due to the limited potential of *Rev2* and *OO2* + *LB* to capture proportionally as many dynamic instructions as the μop caches. Benchmarks such as *tele.adpcm.compress* exemplify this as loop buffering saves no dynamic energy while μop caches result in great energy benefits.

The energy results for 4-wide issue processor configurations on MiBench are show in Figure 8.10. In contrast to the 2-wide configurations, the *Rev* design outperforms the *OO* + μ*C* design overall. This is possible because the 64-entry instruction window on the *Rev4* design is large enough that

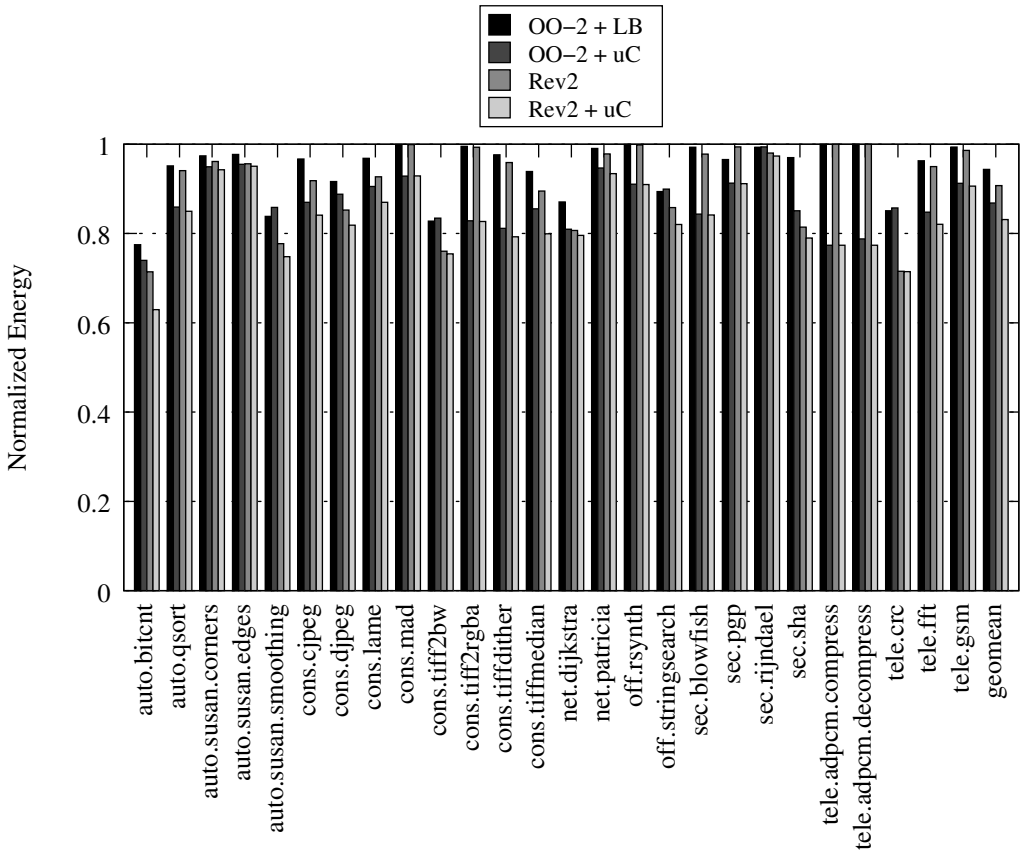


Figure 8.9: MiBench 2-Wide Normalized Energy

it captures significantly more loops than the *Rev2* design on MiBench.

SPECINT

Energy results for the 2-wide and 4-wide issue processor configurations for the SPECINT 2006 benchmarks are shown in Figures 8.11 and 8.12 respectively.

For the 2-wide configuration, *OO2* + μ*C* again outperforms the *Rev2*

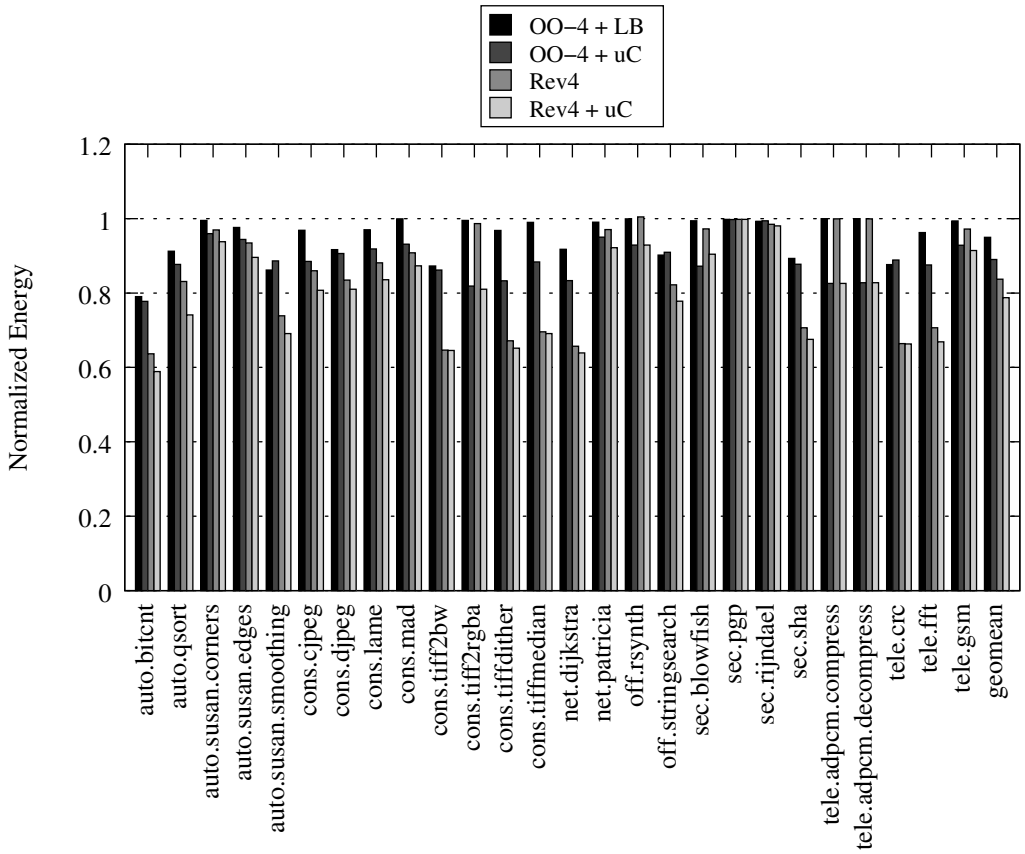


Figure 8.10: MiBench 4-Wide Normalized Energy

configuration overall. However this trend reverses in the 4-wide issue configurations with the *Rev4* design offering greater energy benefits than the corresponding *OO4* + μ*C* design. The *462.libquantum* benchmark results in the greatest overall energy savings from all instruction buffering techniques since the benchmark spends almost all execution within loop bodies with fewer than 10 instructions.

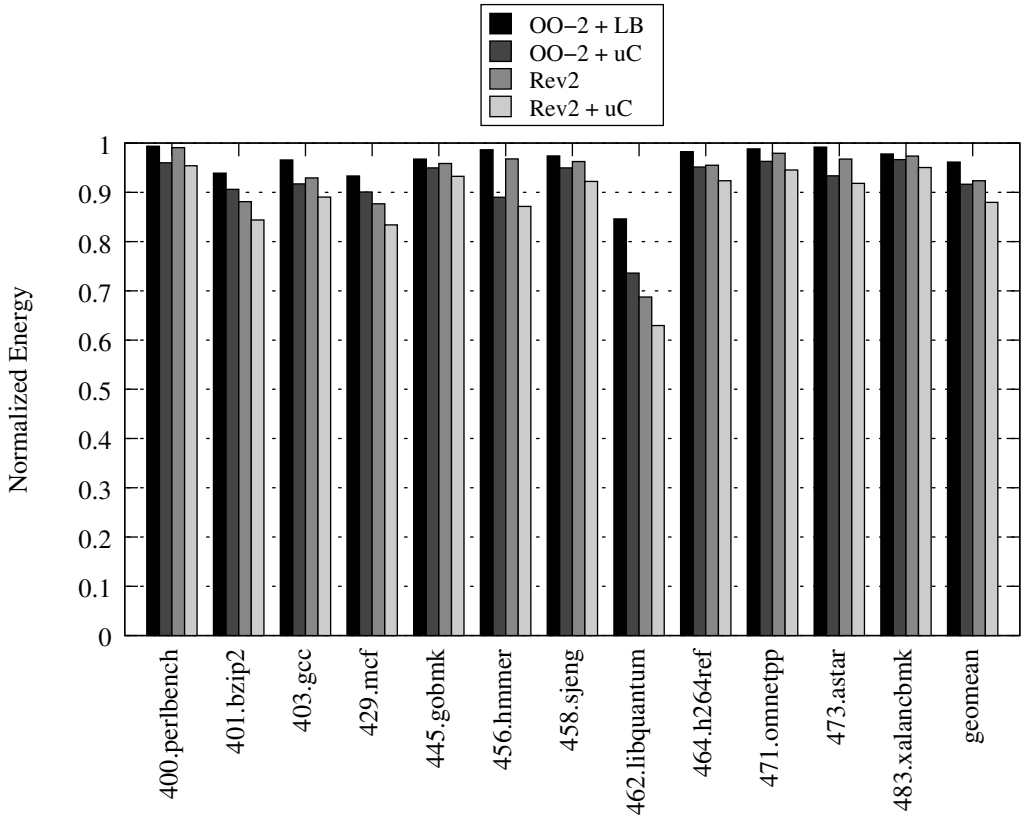


Figure 8.11: SPECINT CPU2006 2-Wide Normalized Energy

SPECFP

Figures 8.13 and 8.14 show the normalized energy consumption of the SPECFP 2006 benchmarks across the different processor configurations. In general, all instruction buffering techniques offer comparatively less energy benefits in comparison to the SPECINT benchmarks. This is due to higher execution unit energy consumption consumed on floating point benchmarks, resulting in less energy reduction opportunity [78, 55].

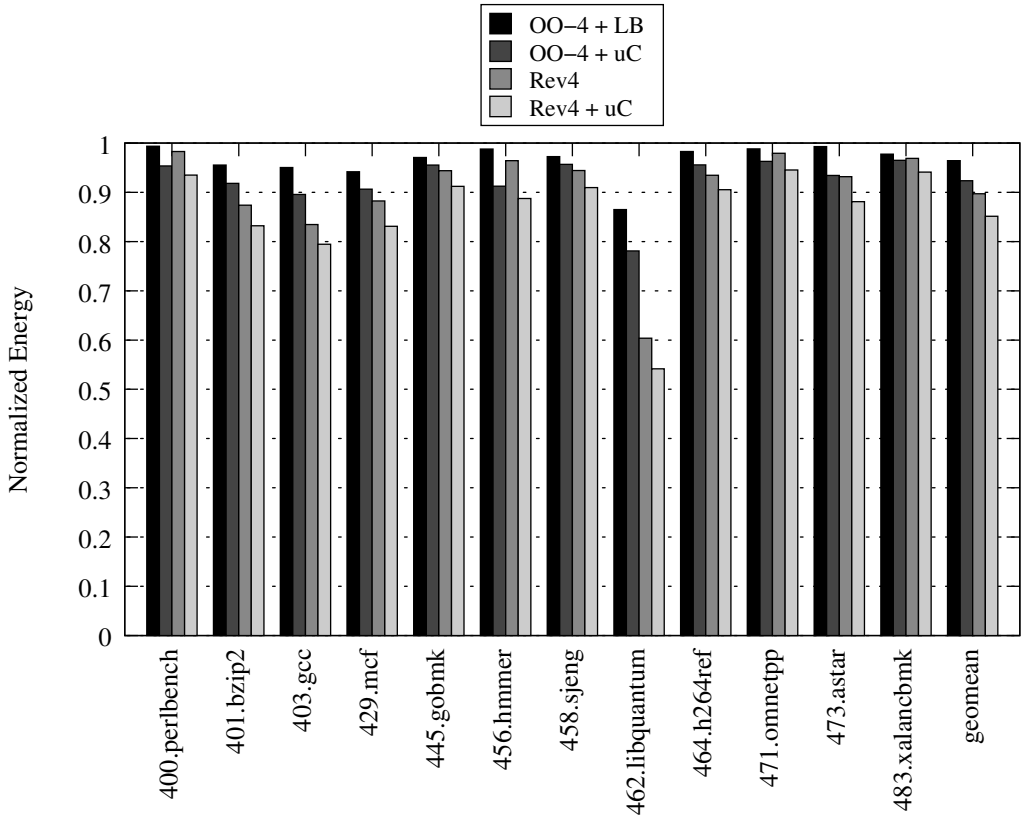


Figure 8.12: SPECINT CPU2006 4-Wide Normalized Energy

For the 2-wide configurations, µop cache equipped designs outperform the *OO2 + LB* and *Rev2* designs. However the both Revolver desings, *Rev4* and *Rev4 + µC* designs offers the best energy efficiency for 4-wide issue machines. The 64-instruction window of Revolver-based designs leads to the most significant energy savings on the *436.leslie3d* and *465.tonto* benchmarks.

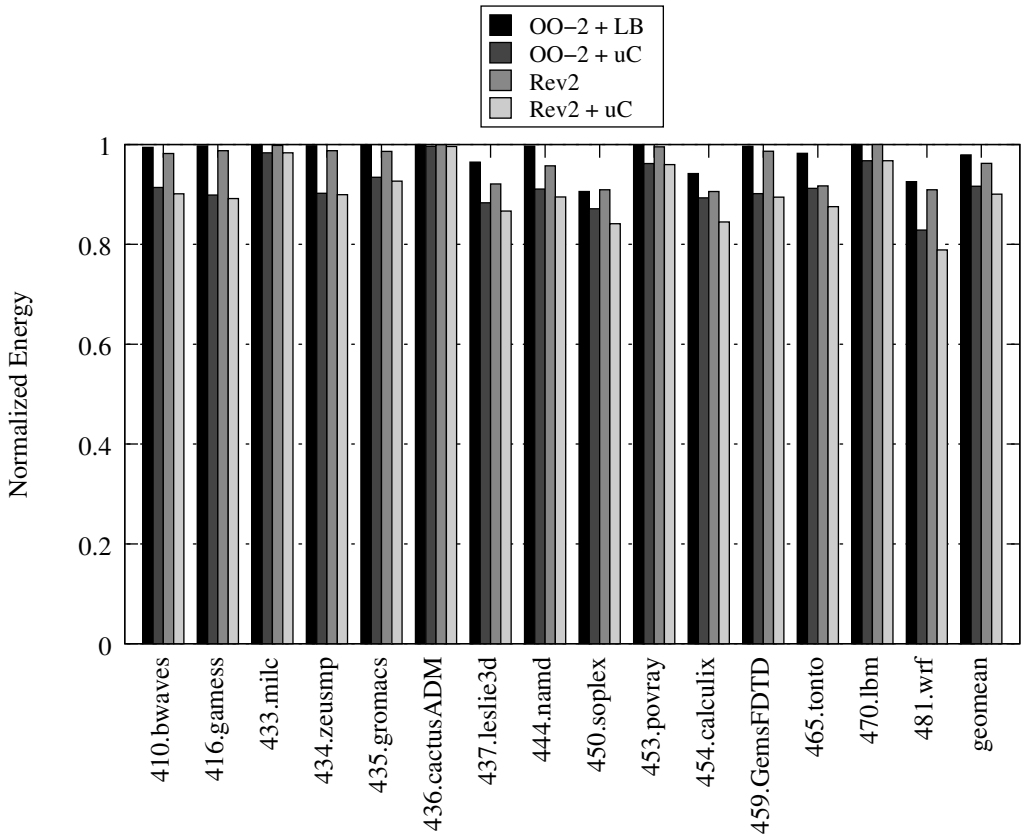


Figure 8.13: SPECFP CPU2006 2-Wide Normalized Energy

Overall Energy-Delay

Although piecewise evaluations of energy and delay highlight specific impacts of in-place loop execution, looking at each characteristic in fails to show the true efficiency of in-place loop execution. In this subsection we present the overall energy-delay picture for in place execution for the conventional out-of-order based Revolver architectures.

Figures 8.15 and 8.16 show the overall energy-delay product results

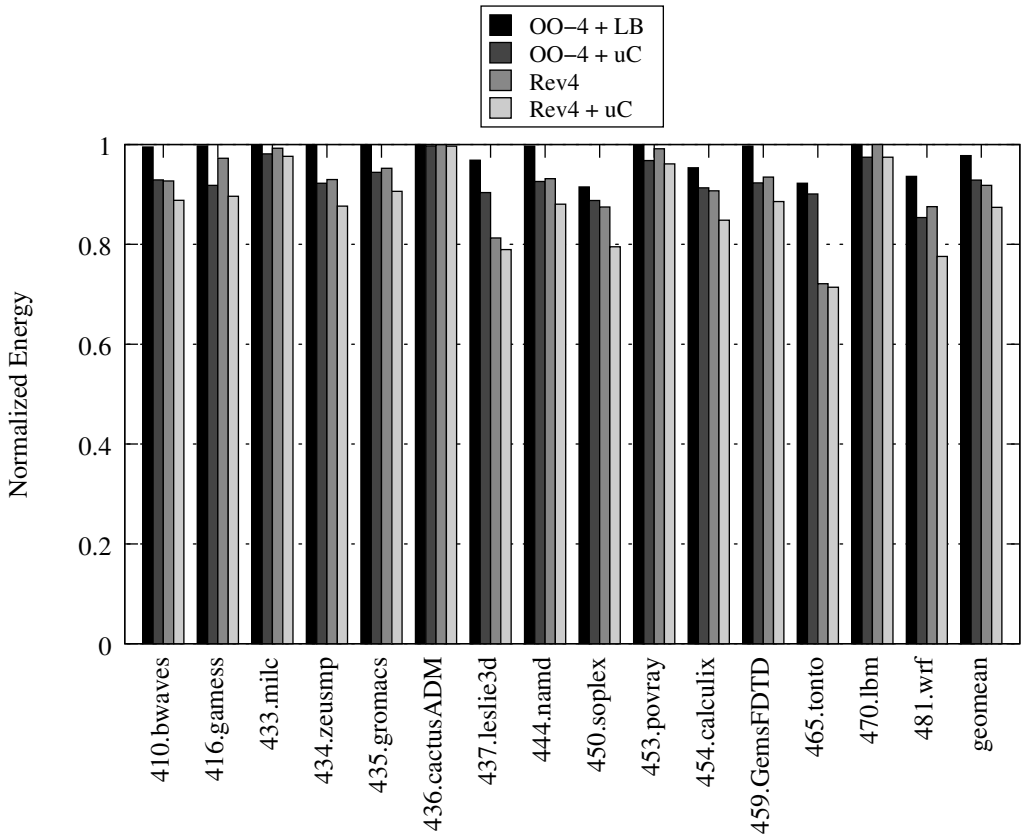


Figure 8.14: SPECFP CPU2006 4-Wide Normalized Energy

for each configuration normalized against a conventional out-of-order withoutany loop buffer or μop cache capabilities. Figure 8.15 presents results for the smaller out-of-order configurations while Figure 8.16 presents results for the large out-of-order designs.

For the small out-of-order designs shown in Figure 8.15, multiple trends are observed. First the benchmark suites perform as expected with the Revolver architectures extracting the most energy benefit from SD-VBS, followed by MiBench and SPEC. Secondly, the Revolver architecture always

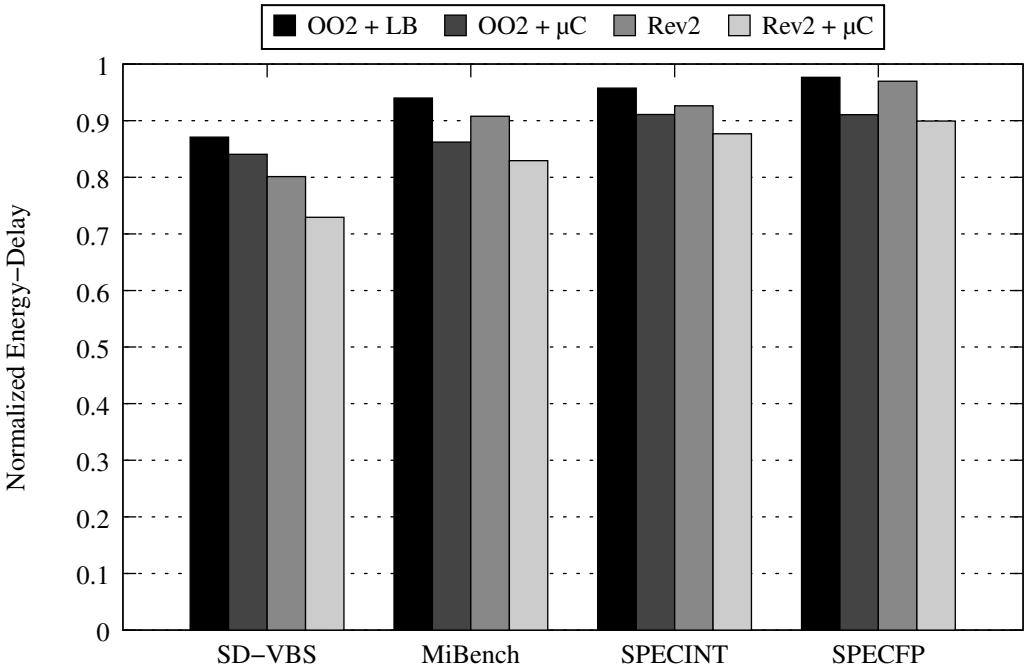


Figure 8.15: Overall 2-Wide Energy-Delay

outperforms the loop buffer equipped out-of-order, due to the reduced energy costs while capturing similarly sized loops. Third, the µop cache always outperforms the loop buffer with respect to entire benchmark suites. Fourth,, for the benchmark suites with fewer capturable loops (MiBench and SPEC), the µop cache outperforms the *Rev2* configuration. Finally, *Rev2* with a µop cache exhibits the best energy-delay performance across all benchmark suites.

On the large out-of-order designs shown in Figure 8.16, Revolver demonstrates even greater energy-delay benefit due to its ability to capture larger loops. Across the benchmark suites, *Rev4* outperforms traditional

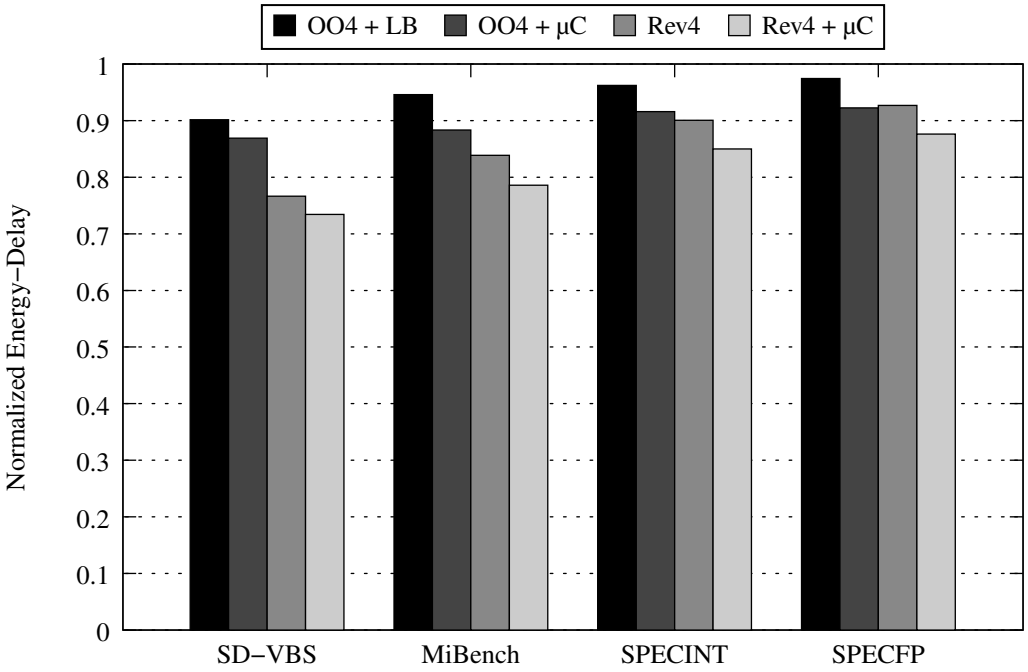


Figure 8.16: Overall 4-Wide Energy-delay

	SD-VBS	**MiBench**	**SPECINT**	**SPECFP**
Rev2 vs. *OO2+LB*	8.7%	3.5%	3.4%	0.7%
Rev2+µ*C* vs. *OO2*+µ*C*	15.3%	4.0%	3.9%	1.3%
Rev4 vs. *OO4+LB*	17.6%	12.8%	6.9%	5.1%
Rev4+µ*C* vs. *OO4*+µ*C*	18.3%	12.4%	7.8%	5.3%

Table 8.6: Energy-Delay Improvement.

out-of-orders on all benchmarks except SPECFP. Again, this is expcted due to the numerous large loops that cannot be captured by Revolver in SPECFP. However, when combined with a µop cache, the Revolver architecture outperforms all other configurations.

In Table 8.6 we show Revolver's relative energy-delay improvement for multiple configurations. With respect to loop buffers, *Rev2* obtains

0.7%-8.7% energy delay improvement against a loop-buffer equipped out-of-order. While *Rev4* observes an even greater 1.3%-17.6% benefit over loop-buffer equipped designs. When all designs are equipped with μop caches, Revolver remains more energy efficient. The two-wide configuration enjoys 5.1%-17.6% benefit, while the 4-wide Revolver configuration observes 5.3%-18.3% benefit over a traditional out-of-order design.

8.3 Loop Design Tradeoffs

The design of loop-mode execution in the Revolver architectures involved many tradeoffs with respect to performance, energy savings, and design simplicity. In this section the interactions and design alternatives relating to loop unrolling, back-end feedback, and branch prediction are examined.

Loop Unrolling

Within the Revolver architectures, although loops are executed in-place within the processor out-of-order back-end, they are unrolled multiple times in order to extract additional instruction level parallelism. This unrolling tradeoff has been made in the previous processor back-end loop buffering proposals. As noted earlier, [40] also unrolls loops into the issue queue, while [63] does not. This subsection investigates the impacts of this design decision.

Performance Impact

The act of unrolling loops in order to extract additional performance is a widely known and longstanding technique used in both hardware and software [23]. Unrolling loops in software comes at the expense of additional instruction cache requirements, although some dynamic operations may be eliminated. Dynamic unrolling of loops by hardware alternatively has no instruction cache impact, however may execute more operations than the corresponding loop unrolled by software. In modern out-of-orders program based loops are dynamically unrolled across all iterations and extract the maximum available instruction level parallelism.

The use of loop-mode execution in the Revolver architectures places additional constraints by limiting the maximum amount of loop unrolling possible. Although much work greatly suggests loop unrolling greatly impacts performance, [63] suggests that loop unrolling has minimal impact on performance when buffering loops in the processor's back-end.

Figures 8.17, 8.18, and 8.19 show the performance degradation realized if loop unrolling were disabled and only a single iteration of a loop were allowed within the out-of-order processor back-end at any time. All designs are normalized against their equivalent design with loop unrolling.

For SD-VBS, shown in Figure 8.17, the lack of loop unrolling hurts performance by 16.5% and 38.1% for the *Rev2* and *Rev4* designs respectively. The larger performance degradation on the *Rev4* design in comparison to the *Rev2* design is due to the larger unrolling factors typically afforded by

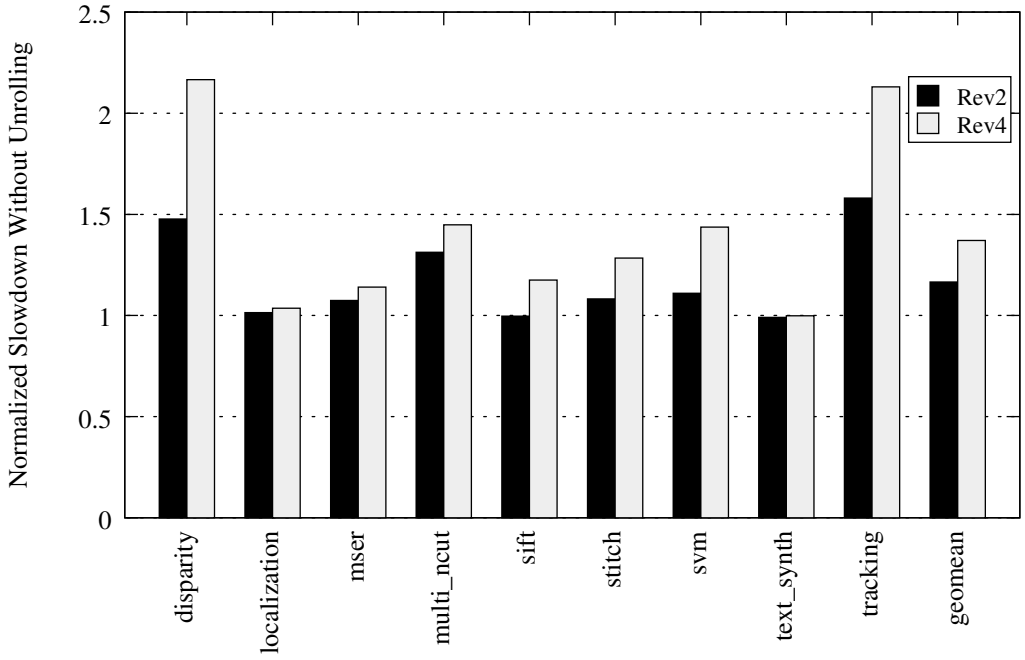


Figure 8.17: SD-VBS Execution Slowdown Without Loop Unrolling

the *Rev4* processor's instruction window. Effectively, during loop mode execution, the instrution windows of the *Rev2* and *Rev4* designs are the same size if unrolling is disabled. Benchmarks which do not benefit from our in-place execution technique, like *text_synth* are unaffected by this design tradeoff.

For MiBench, shown in Figure 8.18, the lack of loop unrolling worsens performance on the *Rev2* and *Rev4* designs by 6.0% and 15.0% respectively. In Figure 8.19 results are presented for the SPEC CPU2006 benchmark suite. Overall, the SPEC benchmark suite is hurt the least by disallowing loop unrolling. No unrolling hurts SPECINT performance by 5.5% and

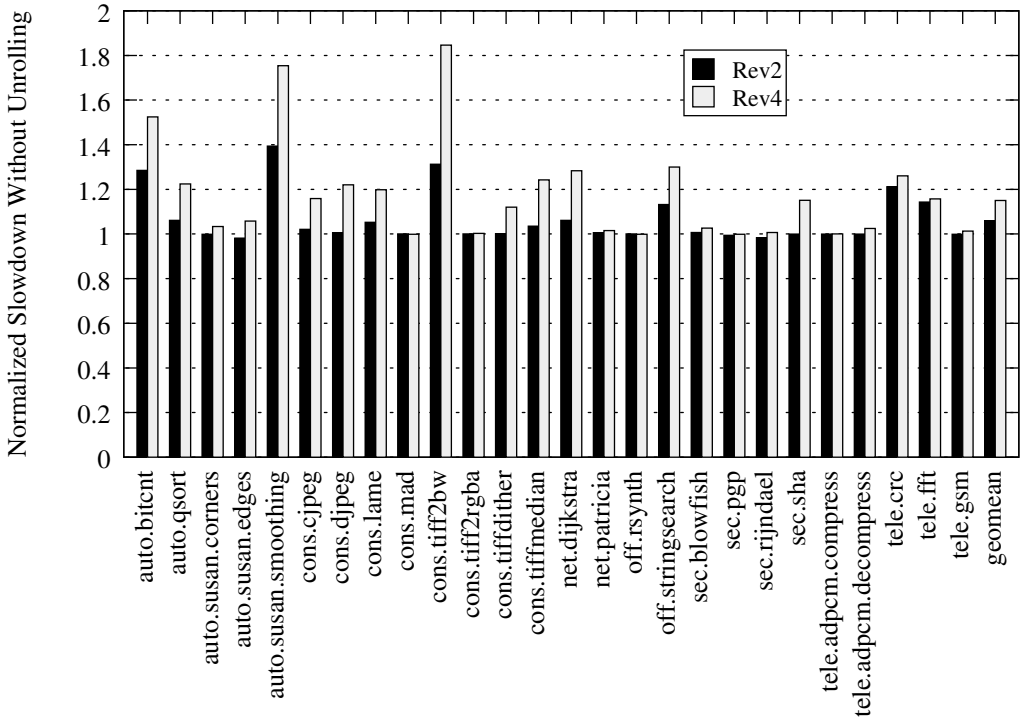


Figure 8.18: MiBench Execution Slowdown Without Loop Unrolling

10.4% for the *Rev2* and *Rev4* designs respectively. The impact on the floating point benchmarks is less with *Rev2* and *Rev4* losing 3.0% and 7.2% performance respectively on SPECFP.

Overall it is observed that loop unrolling is highly beneficial to performance during loop-mode operation.

Instruction Dispatch Impact

Although limiting loop unrolling negatively impacts performance it does provide energy consumption benefits. To evaluate the potential savings

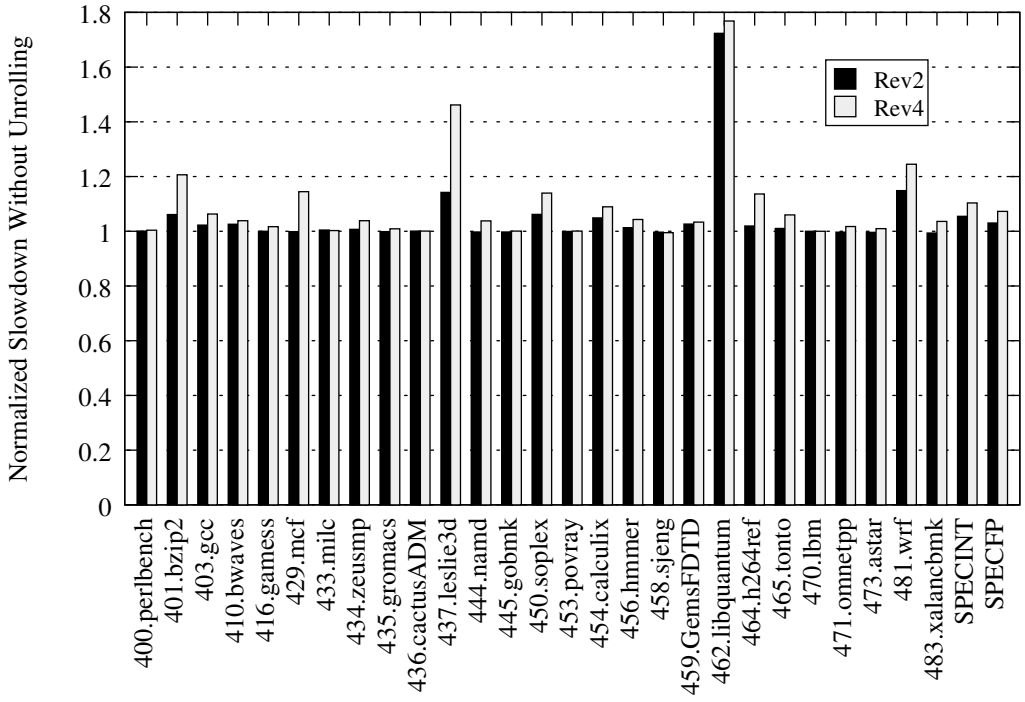


Figure 8.19: SPEC CPU2006 Execution Slowdown Without Loop Unrolling

from eliminating unrolling, Figures 8.20, 8.21, and 8.22 present the number of dispatched instructions without loop unrolling across the benchmark suites. All designs are normalized against their respective baseline with in-place execution and loop unrolling.

Shown in Figure 8.20, eliminating loop unrolling reduces instruction dispatches by 22.0% and 42.0% for the *Rev2* and *Rev4* designs respectively. Although these are significant dispatch reductions, they were accompanied by 17%-38% performance reductions in the previous study. As front-end energy is already reduced through the use of the Revolver architectures, eliminating loop unrolling is not a sensible energy-delay based

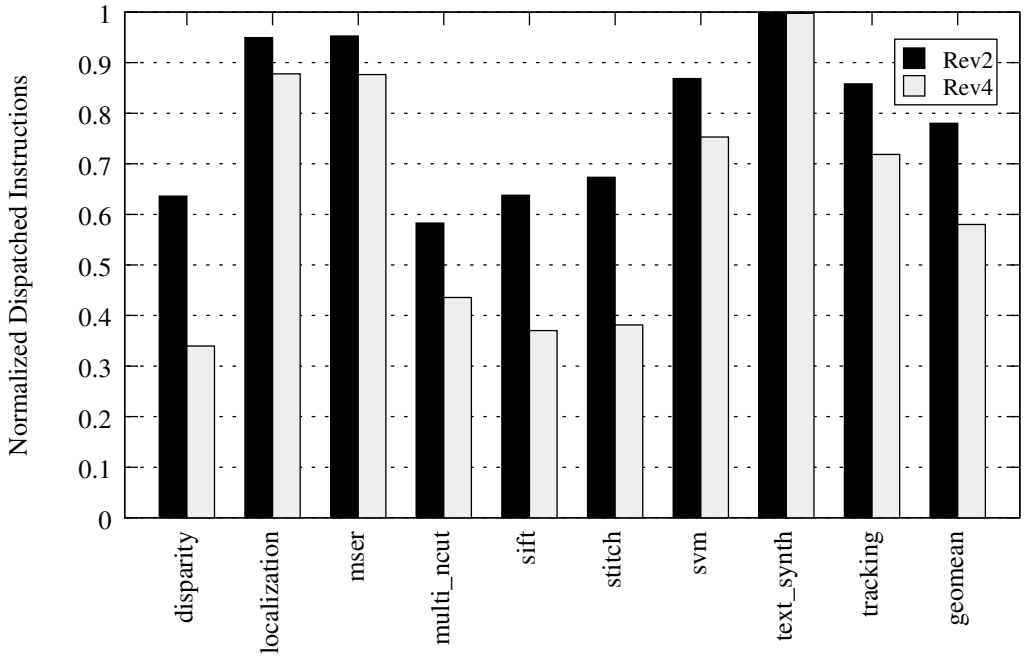


Figure 8.20: SD-VBS Normalized Dispatched Instructions Without Loop Unrolling

tradeoff.

Figure 8.21 shows that eliminating loop unrolling can save 10.4% and 21.8% of all instruction dispatches on the MiBench suite for the *Rev2* and *Rev4* designs.

Finally, instruction dispatch results for the SPEC CPU2006 benchmark suite are shown in Figure 8.22. On the SPECINT subset, no unrolling results in a 6.3% and 7.7% reduction in instruction dispatches for the 2-wide and 4-wide Revolver architectures. SPECFP results in 6.4% savings for the 2-wide configuration, while the 4-wide confuration eliminates

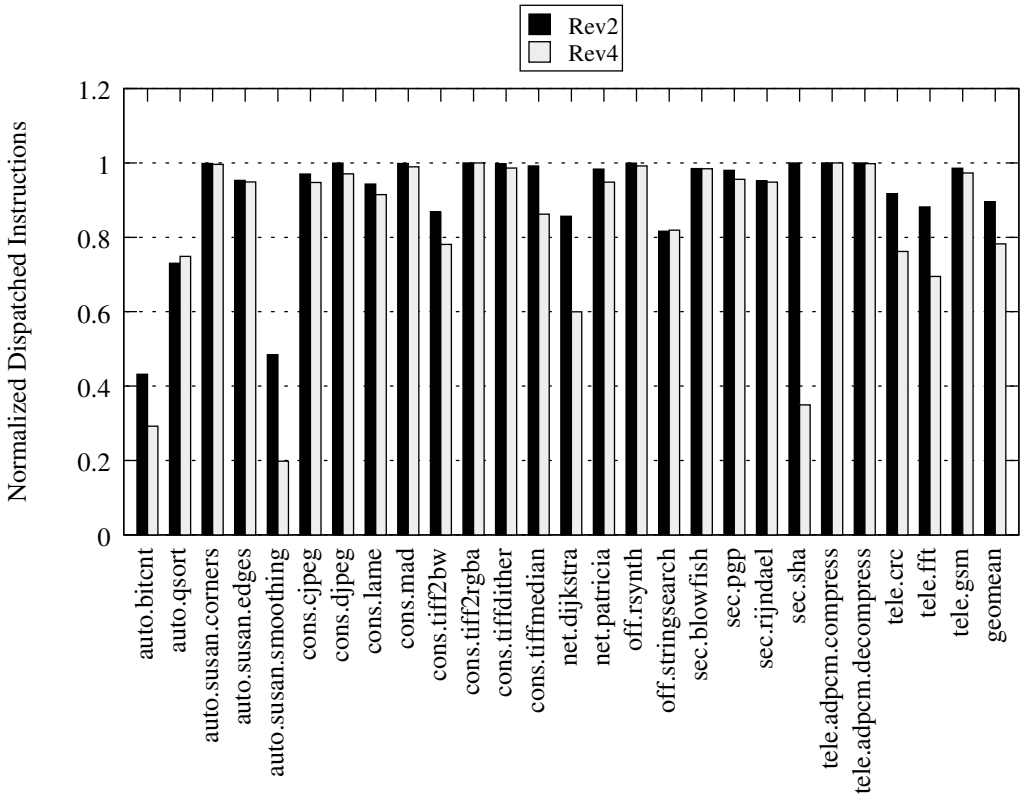


Figure 8.21: MiBench Normalized Dispatched Instructions Without Loop Unrolling

10.2% of all instruction dispatches.

Due to the large performance regressions and limited dispatch benefits resulting from limited loop dispatch, loop unrolling is utilized within the Revolver architectures.

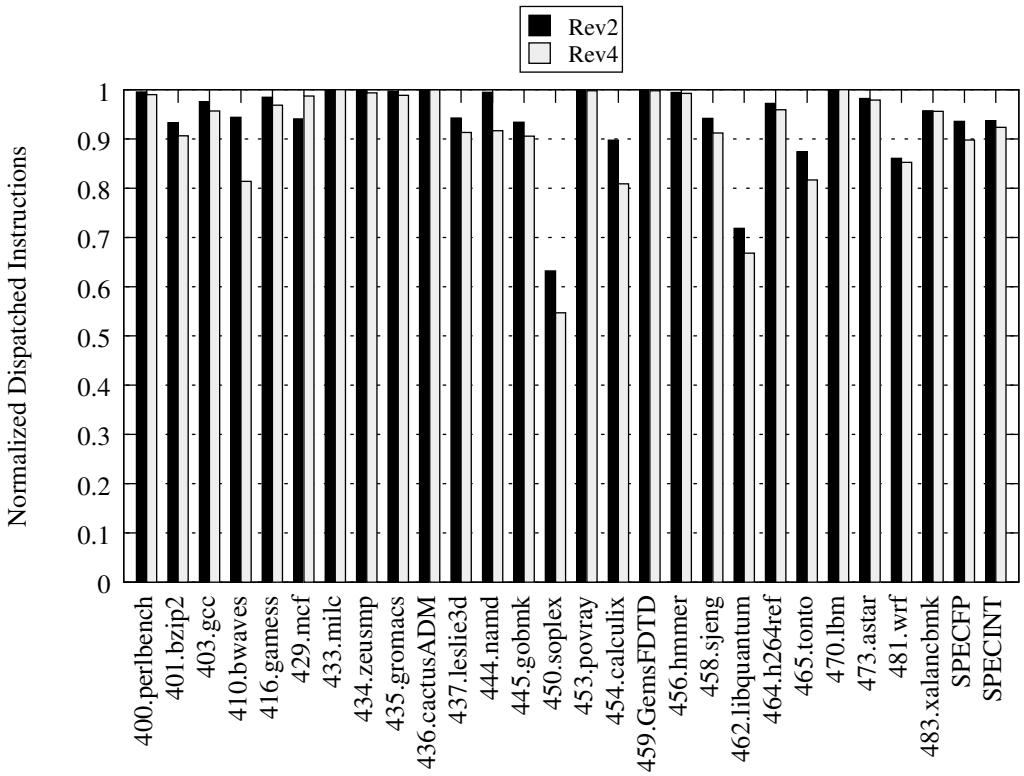


Figure 8.22: SPEC CPU2006 Normalized Dispatched Instructions Without Loop Unrolling

Loop Training Feedback

Described earlier in Section 3.4, feedback from the out-of-order processor back-end provides the function of disabling loop-mode dispatch for loops with unstable control flow or do not iterate a sufficient number of times. In this subsection the impact and effectiveness of the loop profitability prediction is evaluated.

To evaluate the impact of back-end feedback, three alternative loop feedback mechanisms are evaluated. In addition to the earlier proposed

feedback policy, the impact of no back-end feedback and feedback without a positive reinforcement mechanism are considered. The first alternative policy, no training (NT), simply performs loop-dispatch without feedback from the processor back-end. With this policy loop-mode dispatch is never impacted by the number of executed loop iterations. Additionally, in the event of a branch misprediction, the executed loop is simply removed from the LAT. The front-end however is free to immediately retrain and attempt loop-mode dispatch again. In the second policy, abort (ABT), any branch mispredictions lead to loop-mode dispatch being permanently disabled. To evaluate the impact of these policies, the remainder of this section evaluates their impact on instruction dispatch and performance.

Figure 8.23 shows the effect of back-end feedback on the number of instructions dispatched for SD-VBS. In general the *Rev2* and *Rev2-NT* configurations result in similar instruction dispatch characteristics with the NT policy resulting in marginally more instruction dispatches. The ABT policy drastically reduces the effectiveness of loop-mode dispatch. Overall, the *Rev2* and *Rev2-NT* configurations result in 84% reductions in dispatched instructions compared to the baseline *Rev2* architecture. Due to the lack of positive reinforcement, the *Rev2-ABT* configuration only results in 36% savings in instruction dispatches.

Figure 8.25 shows the impact of the three training policies in terms of execution time. Overall the designs perform similarly, although the *Rev2* and *Rev2-NT* configurations degrade performance on some benchmarks.

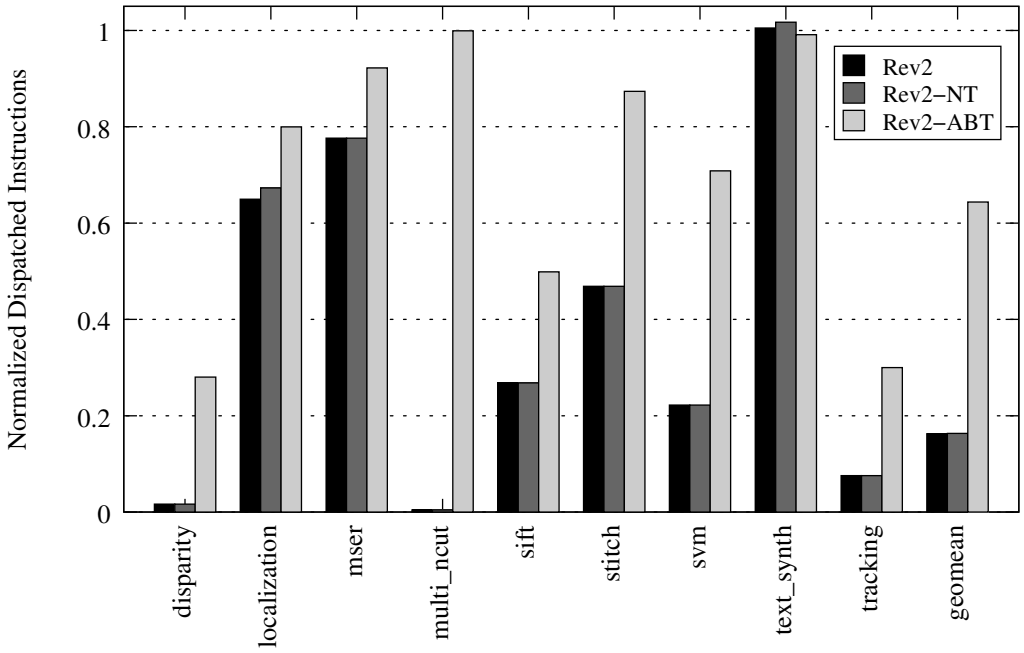


Figure 8.23: SD-VBS Loop Profitability Feedback - Dispatched Instructions.

The *Rev2-ABT* policy offers the best performance of all designs, bettering overall performance over the baseline *OO2* design by 0.1%.

Figure 8.26 shows the instruction dispatch impact of the policies on the MiBench suite. On most benchmarks *Rev2* and *Rev2-NT* perform similarly, however for benchmarks like *auto.bitcnt* the impact of training results in the disabling of loop-mode dispatch more frequently on the *Rev2* design. Overall, the *Rev2-ABT* policy increases the number of dispatched instructions by 36% over the *Rev2* configuration.

Next, Figure 8.27 presents the impact of feedback policy on execution time for the MiBench suite. Although performance is similar on many

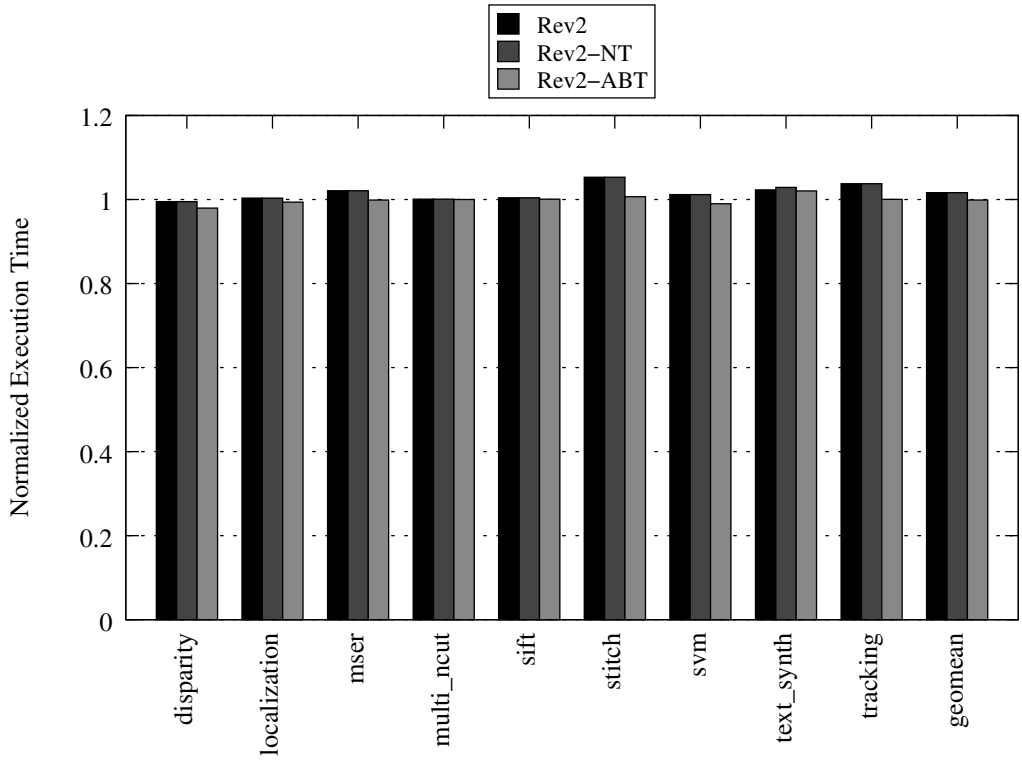


Figure 8.24: SD-VBS Loop Profitability Feedback - Execution Time.

benchmarks, significant performance regressions are observed on some benchmarks. For the *auto.bitcnt* and *auto.qsort* benchmarks, the lack of feedback in the NT policy significantly degrades performance. For these benchmarks the performance degradation is due to the presence of unstable control flow.

Figure 8.28 details the number of dispatched instructions for the SPEC CPU2006 benchmark suite. Again, in comparison to the other policies, the *Rev2-ABT* configuration significantly increases the number of dispatched

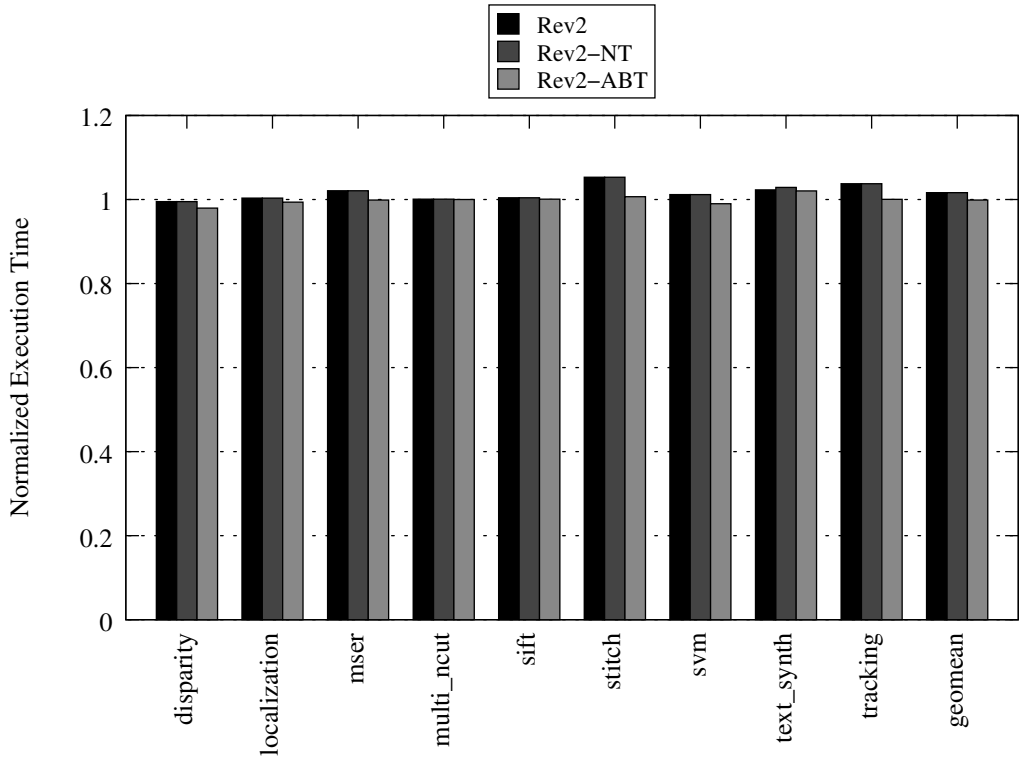


Figure 8.25: SD-VBS Loop Profitability Feedback - Execution Time.

instructions.

Finally, Figure 8.29 shows the impact on profitability training on the SPEC CPU2006 benchmark suite. Both the *Rev2* and *Rev2-ABT* policies negligibly impact execution. Improper loop dispatch on *Rev2-NT* increases execution time by 2.9% and 1.8% on the integer and floating point benchmark subsets respectively.

Overall, it is observed that the existing policy effectively reduces the number of required instruction dispatches while successfully mitigating

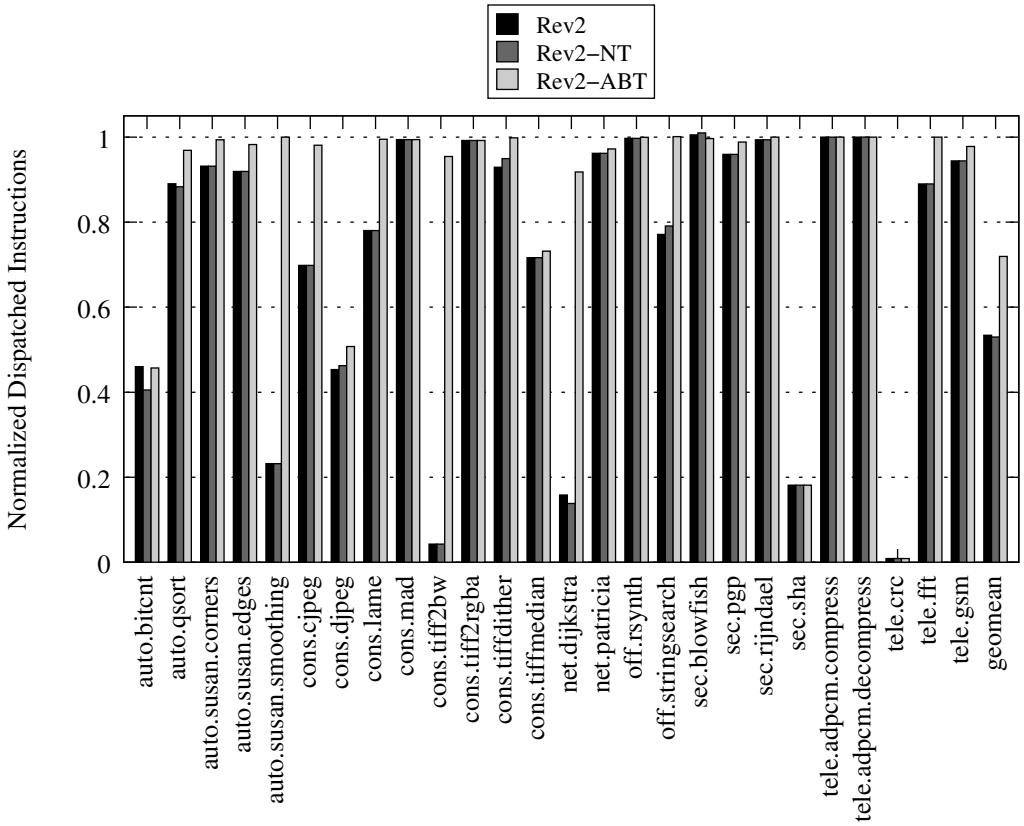


Figure 8.26: MiBench Loop Profitability Feedback - Dispatched Instructions.

potential performance regressions due to unstable control flow.

Branch Prediction Impact

The process of performing loop-mode execution in the Revolver architectures impacts most microarchitectural structures. In the evaluated microarchitectures, a traditional tournament-based branch predictor is

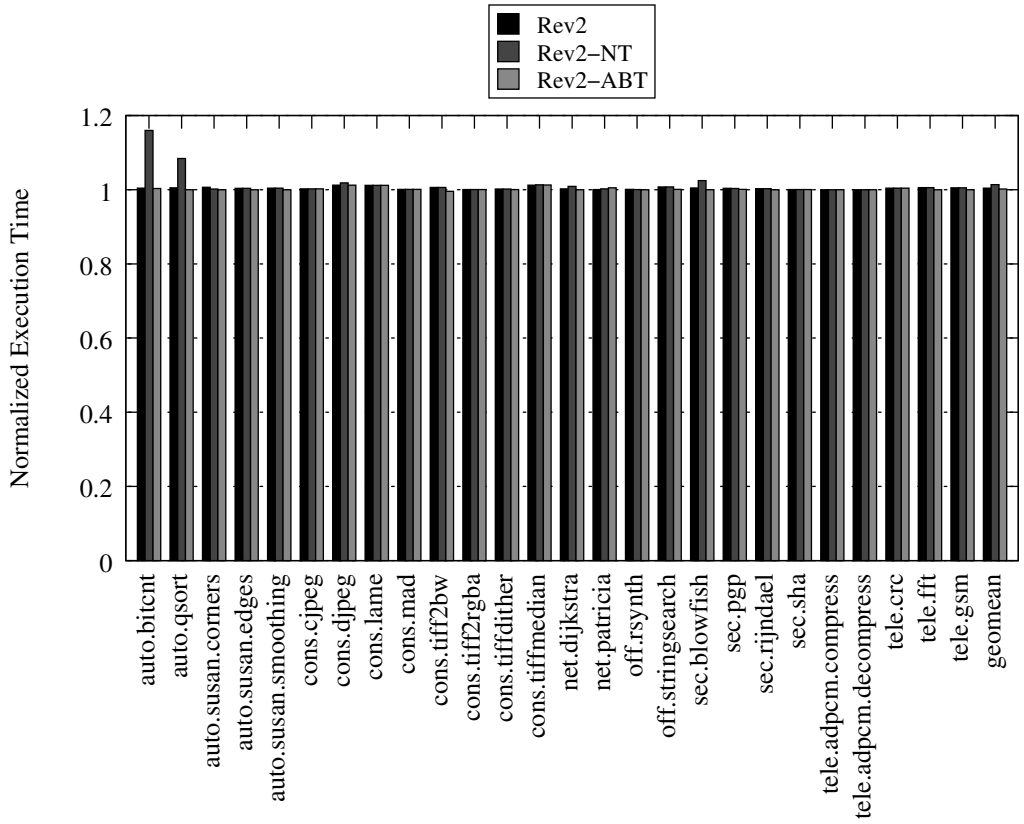


Figure 8.27: MiBench Loop Profitability Feedback - Execution Time.

utilized [57]. In recently proposed branch predictors, filtering branch predictor lookups and updates is used to better branch prediction accuracy and power [70]. Since during loop-mode execution branches within loop bodies are statically predicted, many lookups and updates are also filtered. Additionally, the number of times a loop executes is no longer predicted, thus a potential source of branch mispredicts is directly eliminated by in-place loop execution. This subsection investigates the impact of loop-mode

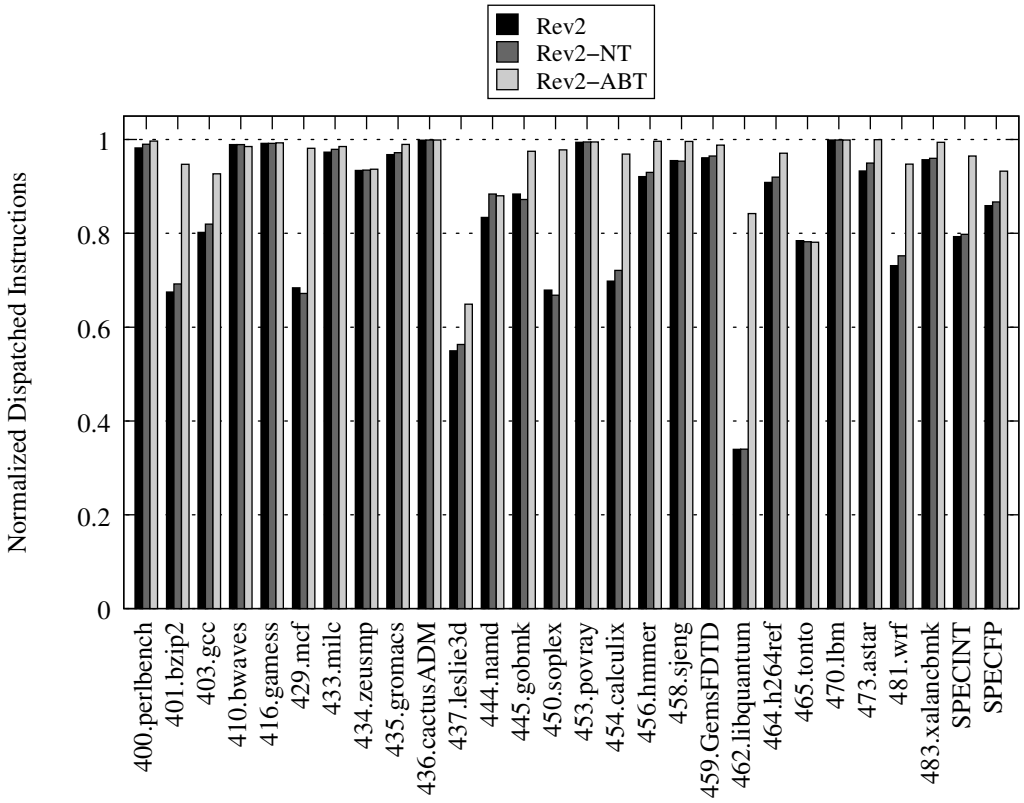


Figure 8.28: SPEC CPU2006 Loop Profitability Feedback - Dispatched Instructions.

execution on branch prediction.

Figures 8.30 and 8.31 show the impact of loop-mode execution on the 2-wide conventional out-of-order based designs for SD-VBS. Shown in Figure 8.30, the percentage of branches mispredicted greatly increase. However, this is expected as loop-mode execution effectively filters the easy to predict branches, resulting in the branch predictor no longer predicting them. Thus the subset of branches predicted are less stable and predictable.

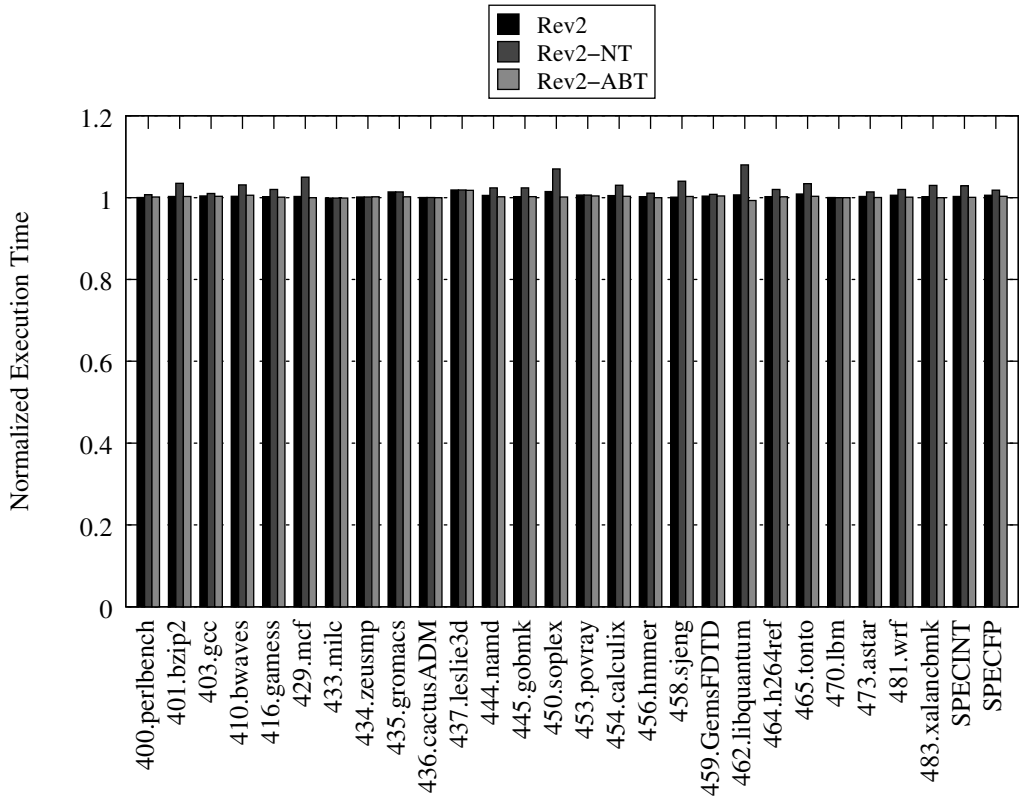


Figure 8.29: SPEC CPU2006 Loop Profitability Feedback - Execution Time.

Figure 8.31 shows the total reduction in branch mispredictions due to loop execution on the *Rev2* architecture. The reduction in branch prediction mispredicts originates from loop-mode ability to handle variable iteration or unpredictable loop iteration counts without pipeline flushes. On a conventional architecture, even on very predictable loops prediction tables require training periods during which loop iteration counts may be improperly predicted. This is visible in 8.31 as the misprediction rate for SD-VBS is quite low for most benchmarks, but great reductions in the

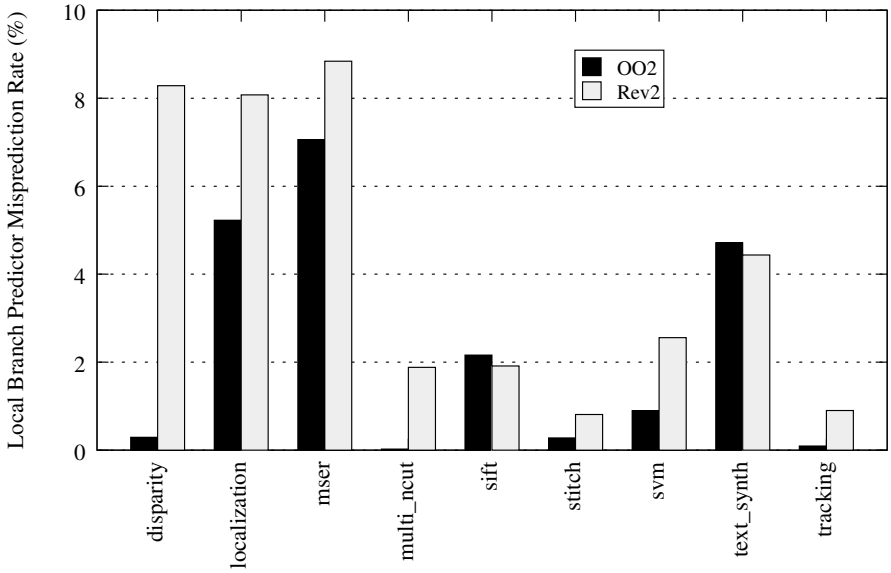


Figure 8.30: SD-VBS Local Branch Misprediction Rate.

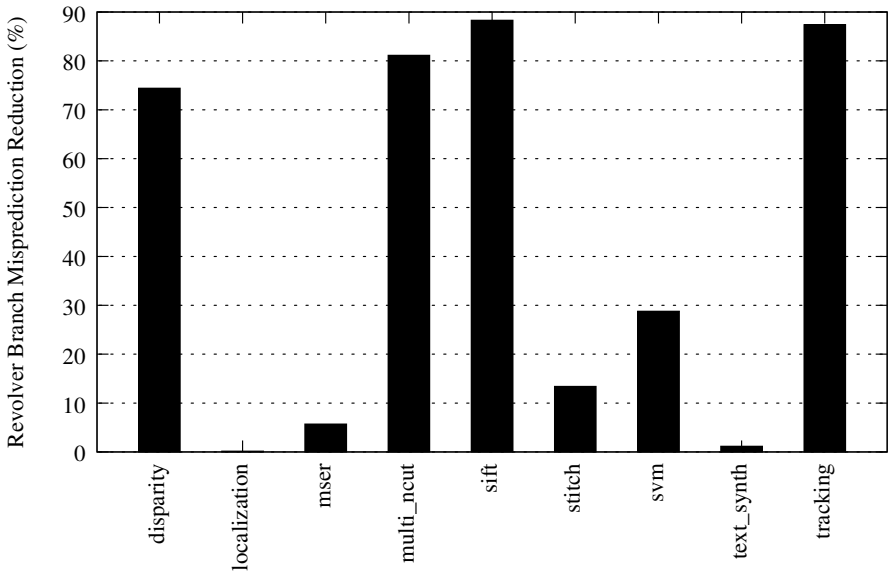


Figure 8.31: SD-VBS Reduction in Branch Mispredicts.

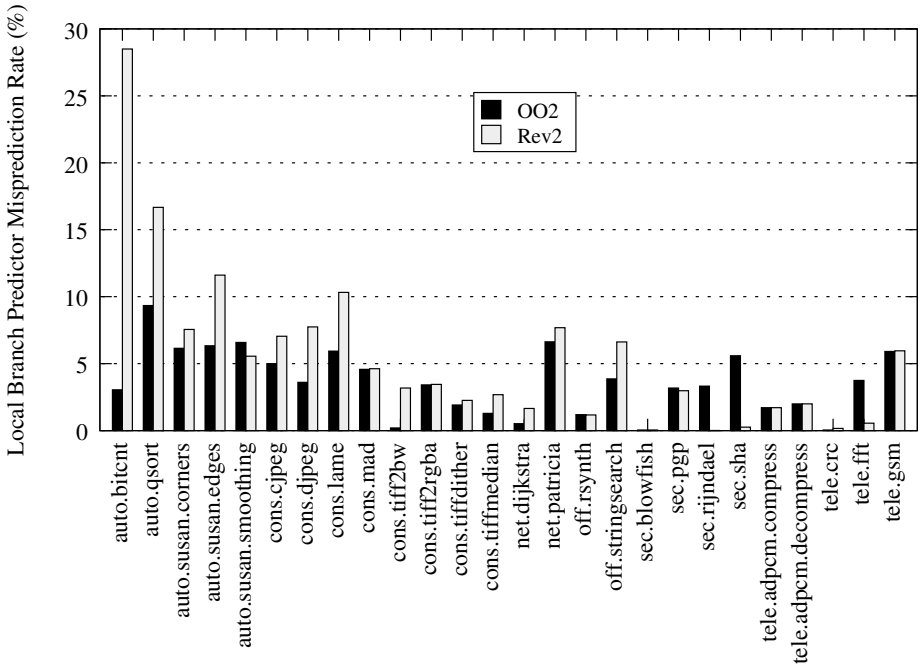


Figure 8.32: MiBench Local Branch Misprediction Rate.

total percentage of mispredictions are possible.

Figures 8.32 and 8.33 show the branch prediction impacts for the MiBench suite. The trend of increased local misprediction rates, but fewer total mispredictions in general is maintained. Only two benchmarks, *cons.jpeg* and *net.patricia* show an increase in total branch mispredictions.

Finally, Figures 8.34 and 8.35 show loop-mode executions impact on the SPEC CPU2006 branch prediction. In general the SPECFP benchmarks which make use of loop-mode execution show the greatest reduction in total mispredicts. This is mostly attributable to their low misprediction

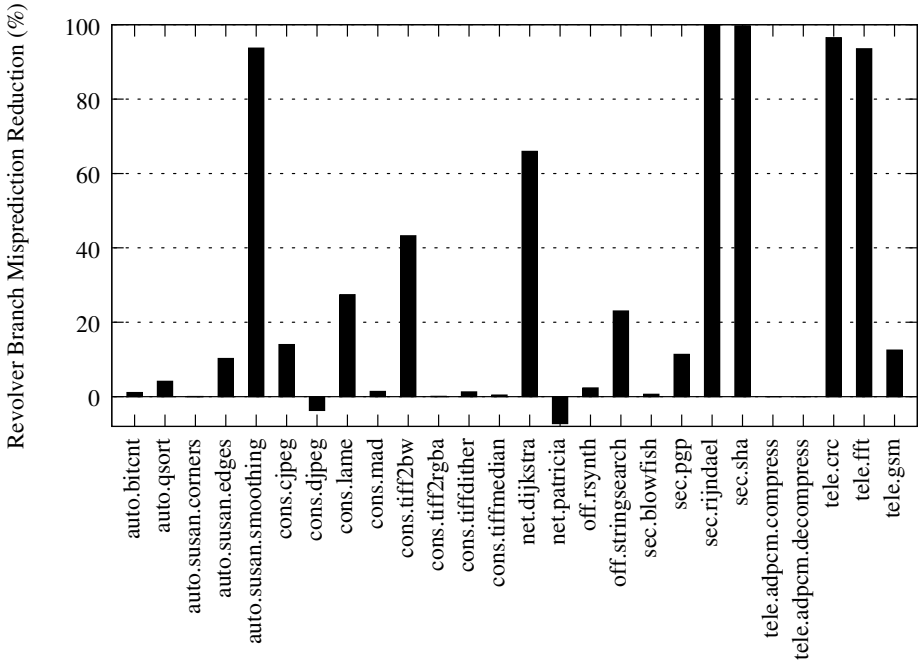


Figure 8.33: MiBench Reduction in Branch Mispredicts.

rates and the lack of predictor training overheads for such loops on the Revolver architectures.

Overall we find loop-mode execution has a minor positive impact on processor performance due to the elimination of training periods to warmup predictor structures during loops.

8.4 Load Pre-Execution

Although the Revolver architectures are targeted primarily towards energy conservation, load pre-execution enables Revolver to extract memory level

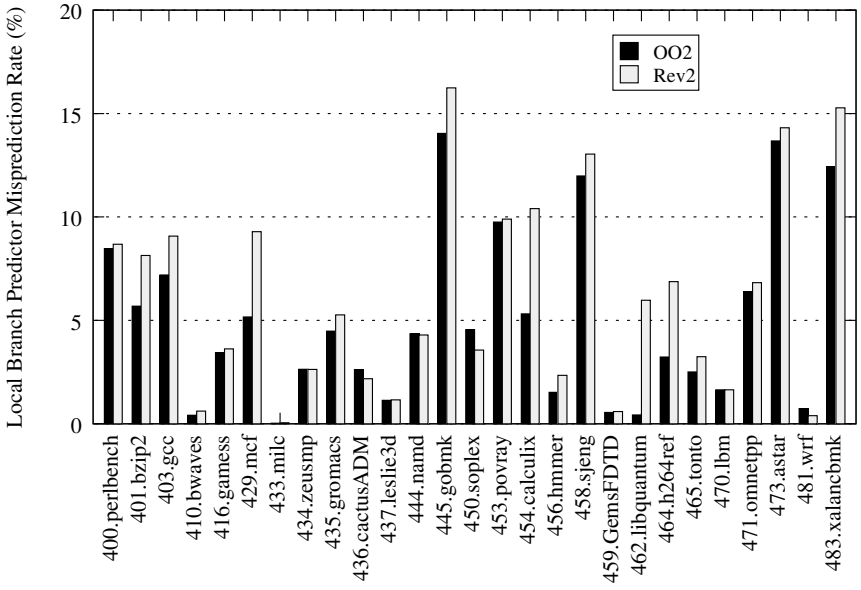


Figure 8.34: SPEC CPU2006 Local Branch Misprediction Rate.

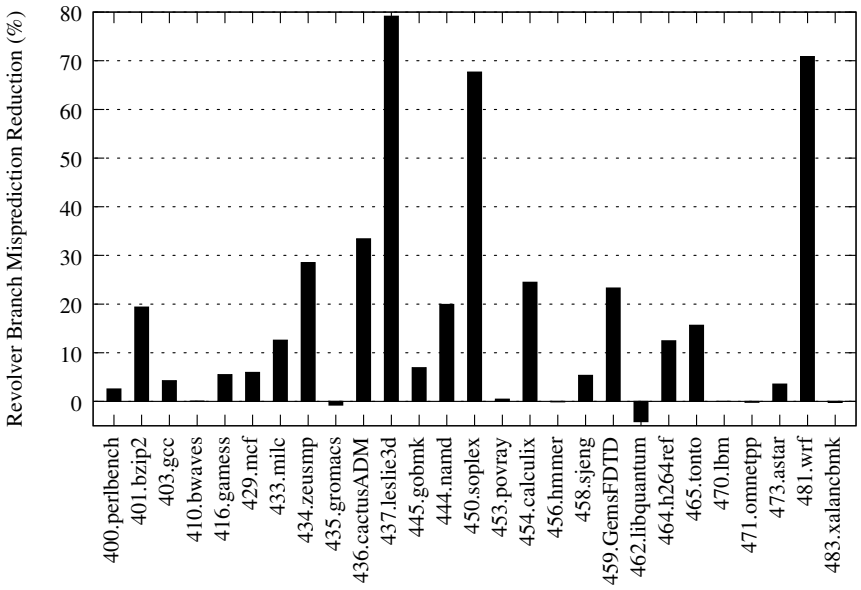


Figure 8.35: SPEC CPU2006 Reduction in Branch Mispredicts.

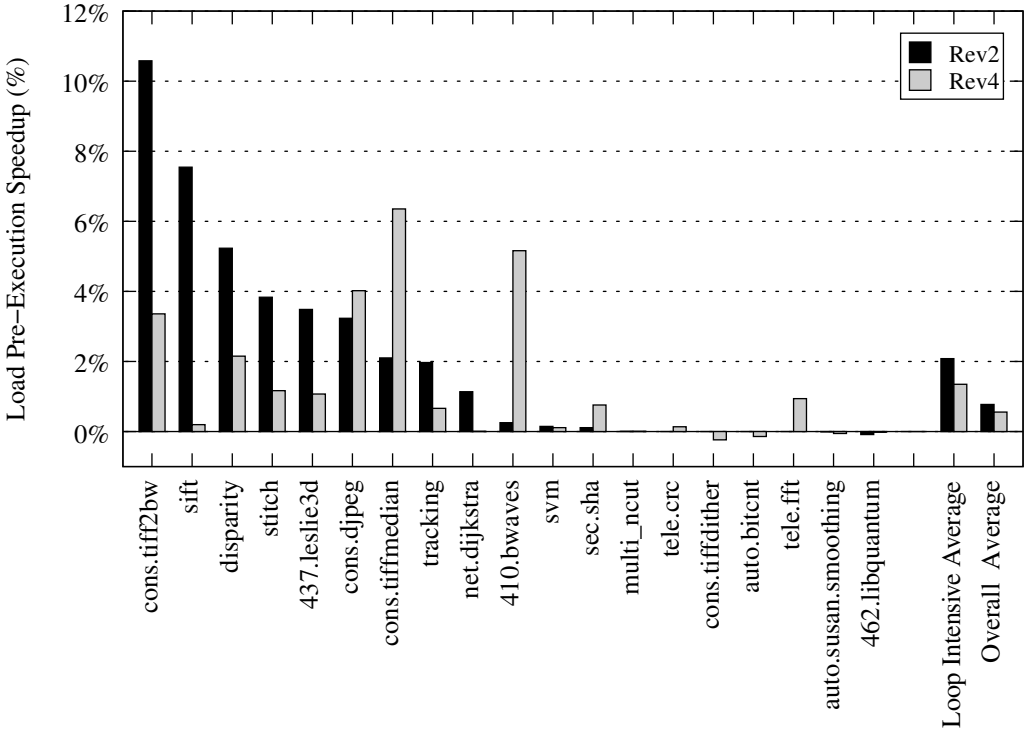


Figure 8.36: Load Pre-Execution Speedup.

parallelism beyond the currently active instruction window. In Figure 8.36, we show the prefetch benefit from load pre-execution obtained by the *Rev2* and *Rev4* configurations on loop intensive benchmarks. Loop intensive benchmarks are defined as those which execute more than 50% of all instructions in loop-mode. On loop intensive code, load pre-execution benefits *Rev2* and *Rev4* by 2.1% and 1.4% respectively. Across all benchmarks, including non-loop intensive codes, the overall benefit is 0.8% and 0.6% for *Rev2* and *Rev4*.

Figures 8.37, 8.38, and 8.39 show the load pre-execution breakdown for

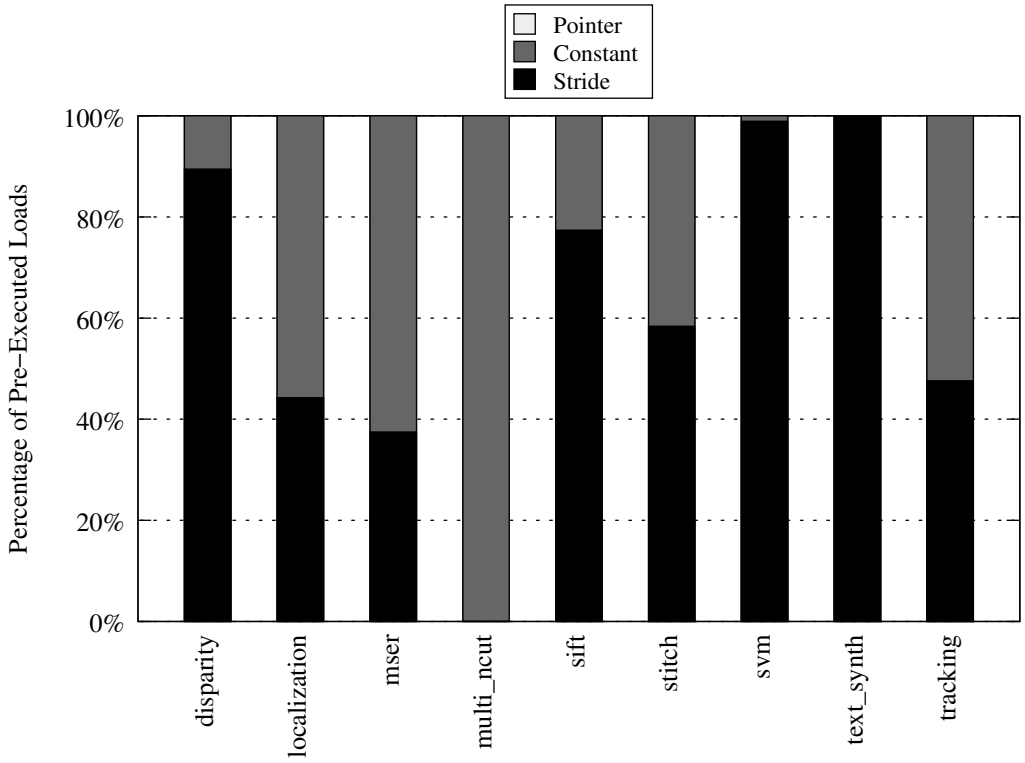


Figure 8.37: SD-VBS Load Pre-Execution Breakdown.

each benchmark suite. Pre-executed loads are classified as being stride-based, constant address, or pointer-based.

As seen in Figure 8.37, virtually all pre-executed loads on SD-VBS are stride-based or constant address. Large fluctuations in the ratios of pre-executed load types exist across the benchmarks.

Figure 8.38 shows the load pre-execution breakdown for the MiBench suite. Again the majority of pre-executed loads are stride-based or constant. Within the networking subset, the *net.dijkstra* and *net.patricia* consist of

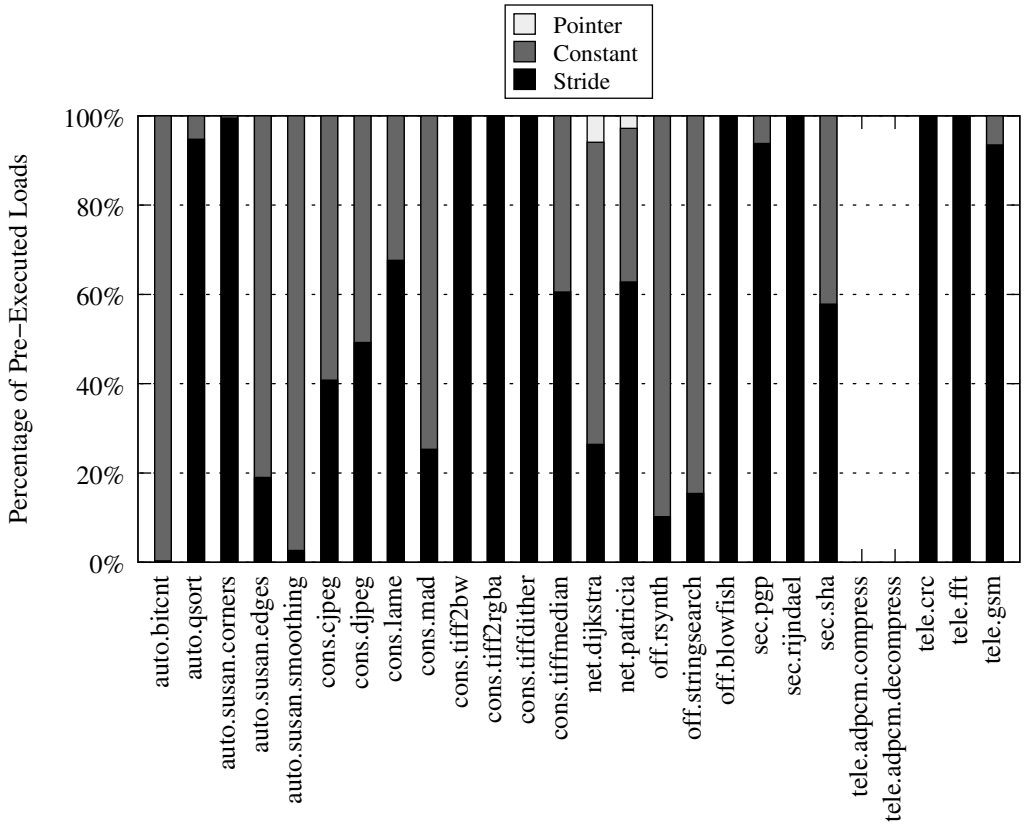


Figure 8.38: MiBench Load Pre-Execution Breakdown.

5.9% and 2.8% pointer-based pre-executed loads. Pointer-based load pre-execution benefit exists on these two benchmarks as dijkstra's algorithm and patricia trees each deal with pointer heavy structures. However the overall benefit of supporting pointer-based load pre-execution is quite low on these benchmark suites. The two missing datapoints for *tele.adpcm* exist as no pre-executed loads are generated for these benchmarks.

Finally, Figure 8.39 shows the pre-execution breakdown for SPEC

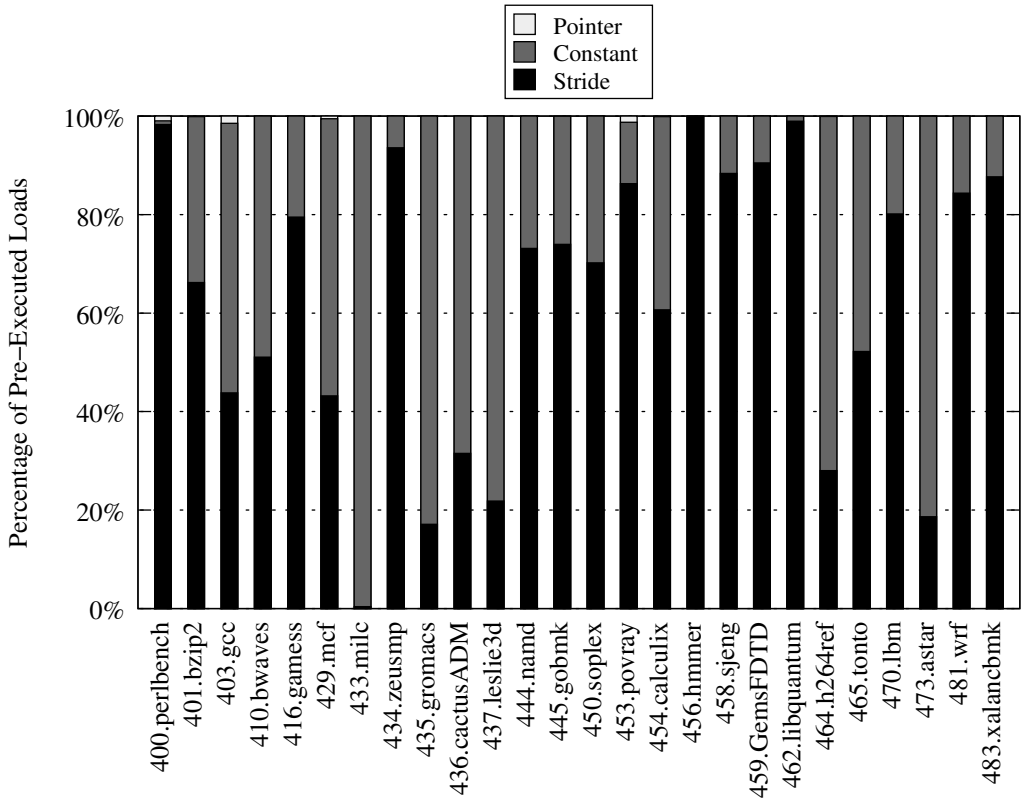


Figure 8.39: SPEC CPU2006 Load Pre-Execution Breakdown.

CPU2006. Pointer-based pre-executed loads are also rare on this benchmark suite. Additionally, no constant and stride based pre-executed loads are quite varied.

8.5 CRIB Out-of-Order

In this section the CRIB-based Revolver architecture is evaluated in terms of performance and energy efficiency.

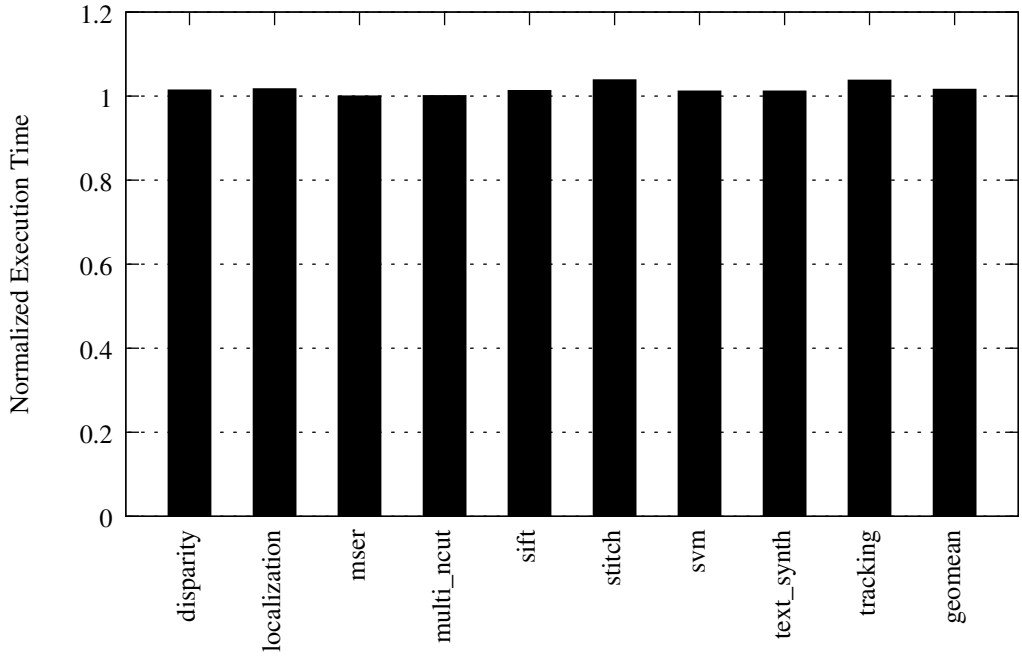


Figure 8.40: CRIB SD-VBS Normalized Execution Time

Performance Impact

Figures 8.40, 8.41,and 8.42, present the normalized execution time of the CRIB-based revolver against a baseline CRIB design without in-place loop execution. Overall, the performance impact of the CRIB-based Revolver is also small, although larger performance deltas relating to in-place execution are realized in comparison to the PRF-based Revolver designs. This performance delta is because, as observed within [32], CRIB makes very effective use of it's provided window size. By performing loop unrolling similarly to our PRF-based designs, possible underutilization of CRIB entries is more likely to lead to performance degradations.

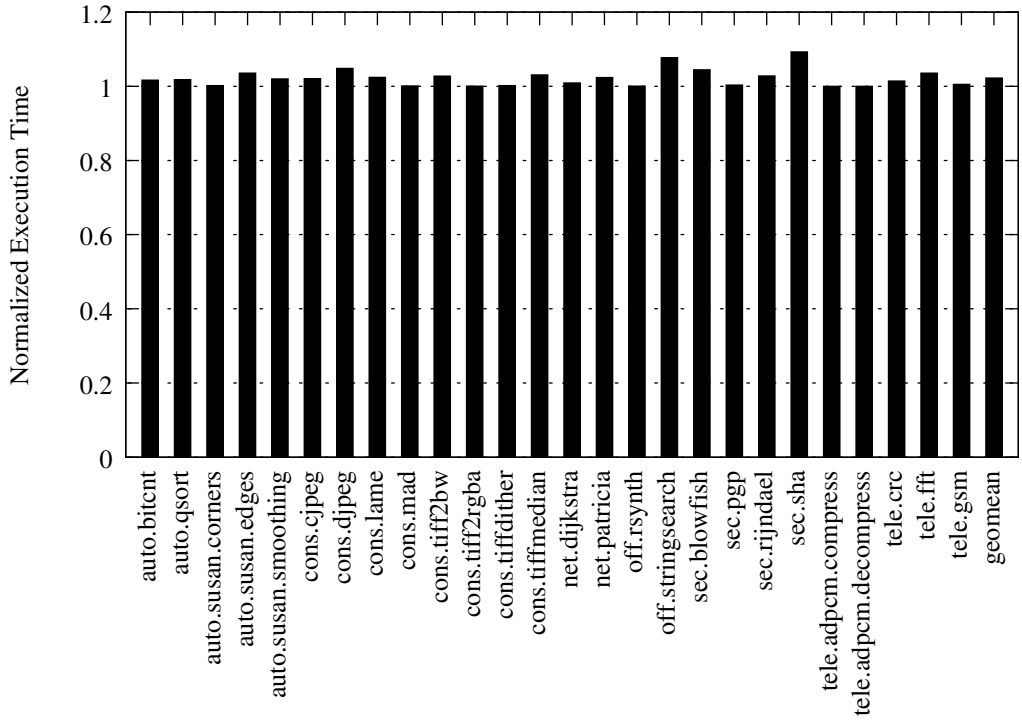


Figure 8.41: CRIB MiBench Normalized Execution Time

Overall on the SD-VBS, MiBench, SPECINT, and SPECFP workloads performance regressions of 1.6%, 2.2%, 0.8%, and 1.7% respectively are observed.

Energy Impact

Although the percentage of eliminated instruction dispatches for the CRIB-based Revolver architecture are similar to *Rev4*, eliminating an equivalent number of instruction dispatches will benefit a CRIB-based design proportionally more than a conventional out-of-order design. The CRIB

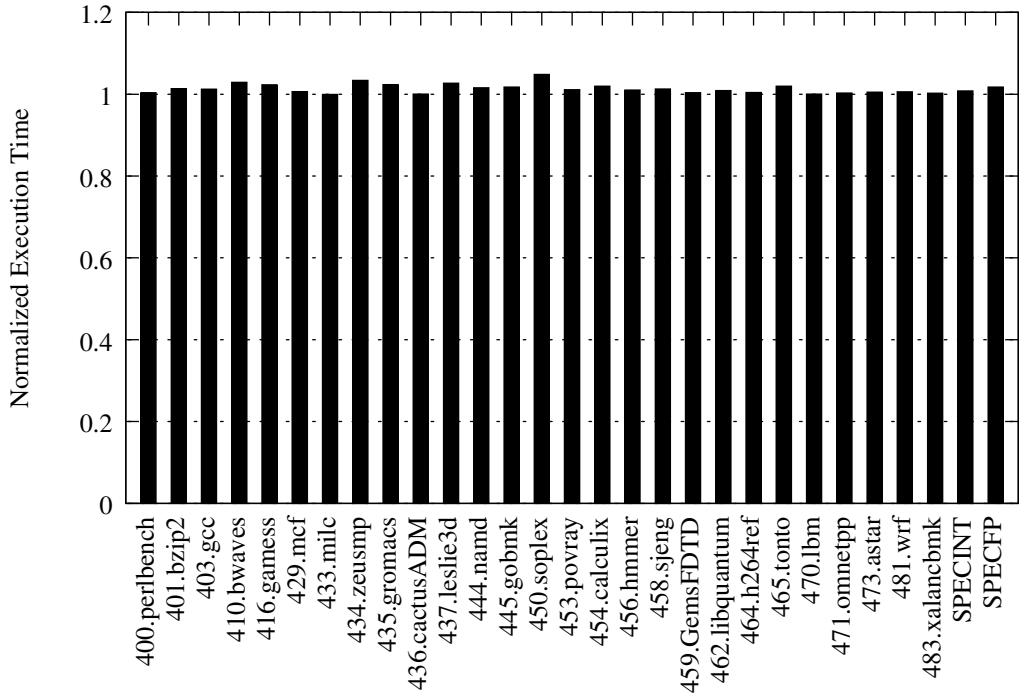


Figure 8.42: CRIB SPEC CPU2006 Normalized Execution Time

architecture minimizes execution energy within the out-of-order back-end, correspondingly the unaffected front-end energy represents a higher overall percentage of total core power.

Figures 8.43, 8.44, and 8.45 present the energy consumed for CRIB-based Revolver designs against an equivalent CRIB design without in-place loop execution. The designs are also compared with and without the presence of μop caches and loop buffers. All designs are normalized against CRIB without any loop buffers or μop caches.

Figure 8.44, shows the energy consumed on the MiBench suite. Overall, The *CRIB+Rev* configuration achieves 17.2% energy savings over the base-

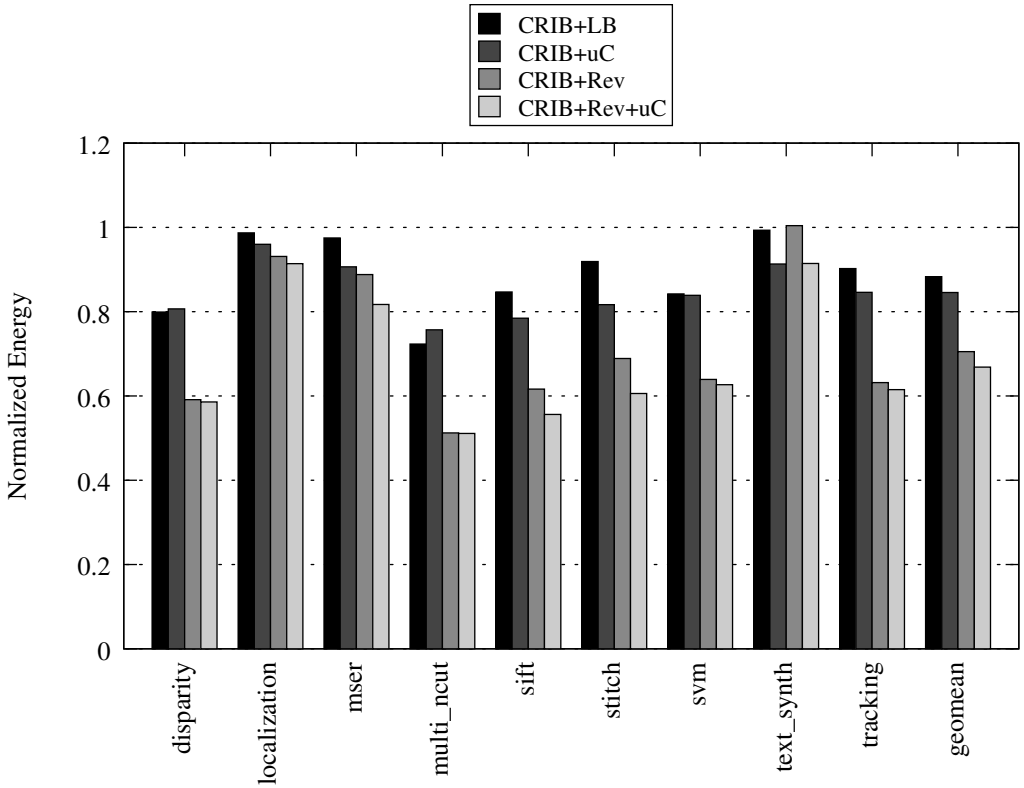


Figure 8.43: CRIB SD-VBS Normalized Energy.

line *CRIB+LB* configuration. With μop caches, the CRIB-based Revolver architecture achieves 17.0% energy savings.

Figure 8.45, shows the energy consumed on the SPEC CPU2006 suite. Overall, The *CRIB+Rev* configuration achieves 9.5% and 8.0% energy savings over the baseline *CRIB+LB* configuration for the SPECINT and SPECFP subsets respectively. With μop caches, the CRIB-based Revolver architecture achieves 10.8% and 7.9% energy savings for the SPECINT and SPECFP benchmarks respectively.

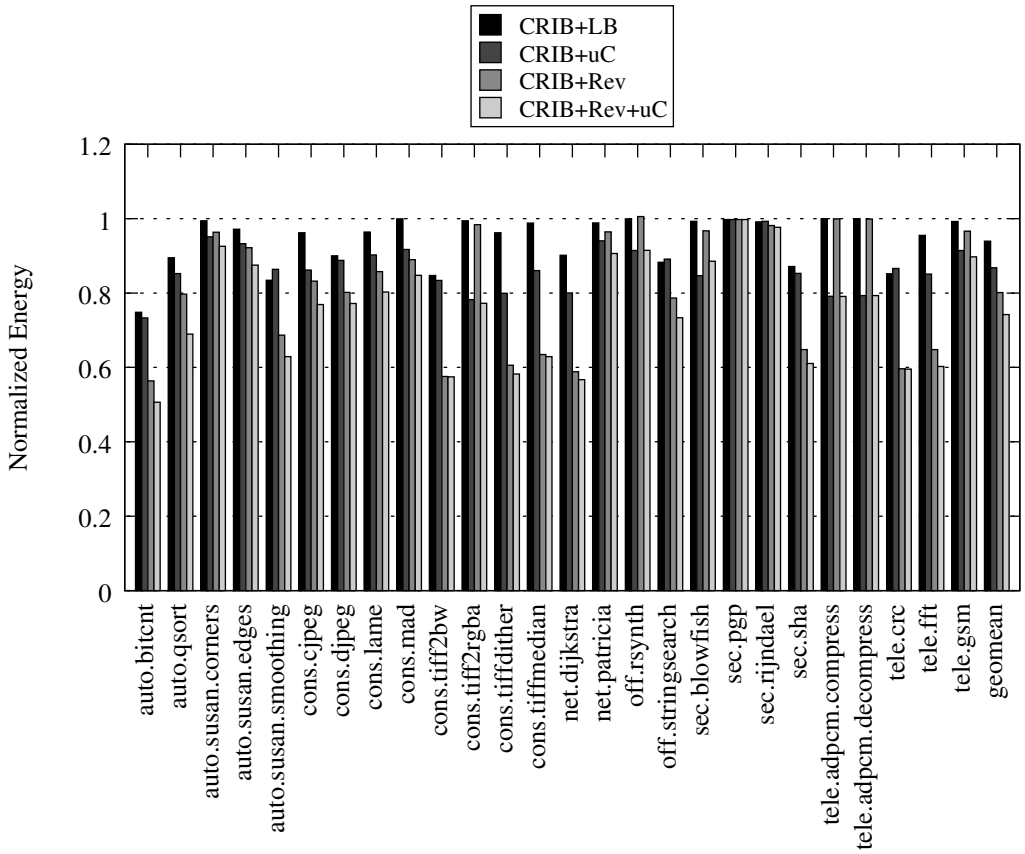


Figure 8.44: CRIB MiBench Normalized Energy.

Overall, the CRIB-based Revolver architecture energy consumption is similar but significantly lower than the PRF-based Revolver architecture.

8.6 Summary

In this chapter performance and physical simulation was performed to evaluate the effectiveness of the Revolver architecture on PRF-based and

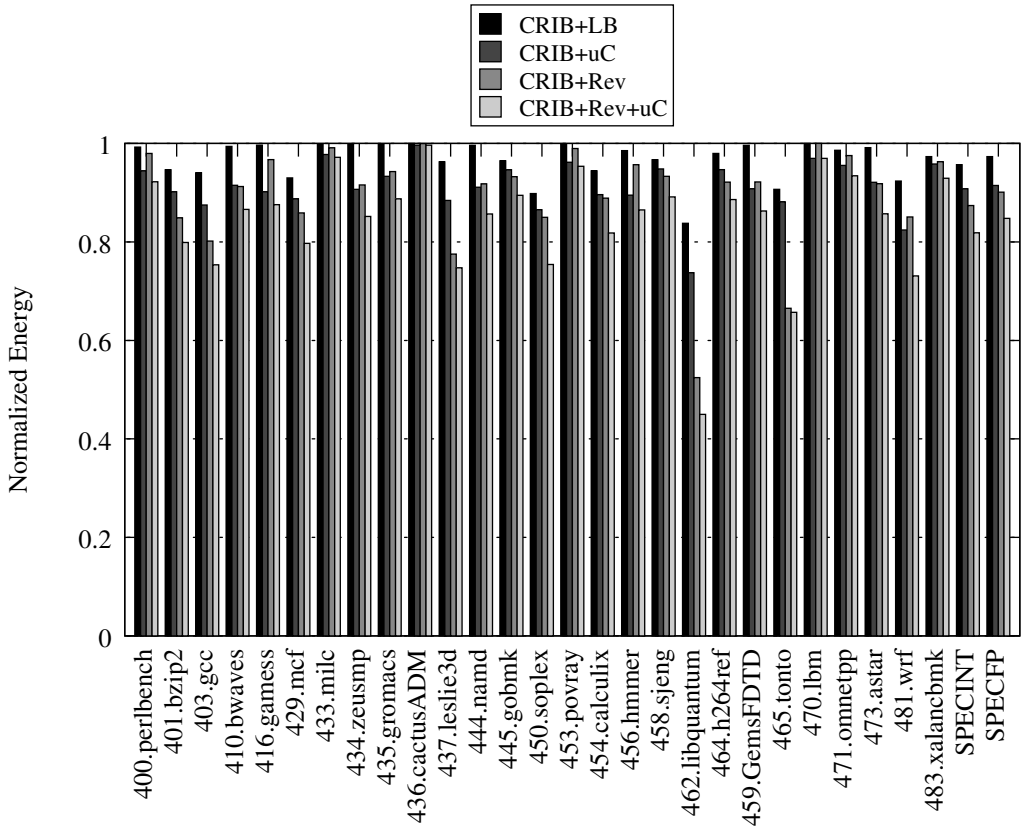


Figure 8.45: CRIB SPEC CPU2006 Normalized Energy.

CRIB-based out-of-order processors. Additionally the impacts of loop unrolling, branch prediction, and load pre-execution on loop-mode execution were evaluated. Overall large energy reductions in front-end power were observed.

9 CONCLUSION

Historically, advances in process technology have provided computer architects with additional transistor resources and energy savings. However, with the end of Dennard scaling, the associated per-transistor energy savings from successive technology nodes has been curtailed. To overcome the stringent energy limitations brought about by this trend, future out-of-order processors must further streamline common execution patterns, thereby eliminating unnecessary pipeline activity.

Prior work has focused on reducing front-end energy overheads by means of pipeline-centric instruction caching mechanisms, like loop buffers, aimed at capitalizing on spatial and temporal instruction access patterns. With time these instruction reuse mechanisms have become progressively more complex and deeply embedded in modern processor pipelines. This thesis represents another step in this logical progression of embedding instruction reuse techniques deep within a processor pipeline.

This thesis presented the Revolver architectures, two processor designs that make use of in-place loop execution within the out-of-order processor's back-end. Through in-place execution, a few static instances of each instruction from a loop body are dispatched to the out-of-order execution core by the processor front-end. After dispatch, each static instruction instance may be executed multiple times in order to complete all necessary loop iterations. During in-place loop execution the processor front-end,

including fetch, branch prediction, decode, allocation, and dispatch logic may be completely clock gated in order to save energy.

To realize the in-place execution, a traditional PRF-based out-of-order and CRIB were utilized as baseline architectures for modification. For the conventional out-of-order, operand dependence linking was moved from the traditional front-end RAT into the processor's out-of-order execution core. Additionally modifications were made to the conventional scheduler and LSQ operation in order to enable in-place execution. For the CRIB architecture the a modified LSQ structure, sufficient for in-place execution was proposed. Additionally, a study to investigate the performance impact of a more ideal CRIB operand network was conducted.

Additionally, to accelerate in-place loop execution, load pre-execution was proposed. Load pre-execution improves processor performance by dynamically detecting common load access patterns and speculatively executing loads from future loop iterations. Through load pre-execution, single-cycle loads are enabled.

Finally, the efficacy of in-place execution and load pre-execution were evaluated across three benchmark suites. In addition to the final configurations many design alternatives were studied. Overall, in-place loop execution was found able to eliminate 20%, 55%, and 84% of all front-end dispatches on the SPEC CPU2006, MiBench, and SD-VBS benchmark suites. Through these large reductions in instruction dispatches, great energy gains were observed for the conventional out-of-order and CRIB-based

Revolver architectures.

9.1 Future Work

During the conduction of this thesis research, multiple avenues of potential future research were discovered. These research areas consist of an extension of our load pre-execution mechanism, an interesting in-depth power study of the CRIB architecture, and an investigation of tightly integrated prefetchers.

Load Pre-Execution Extensions

In Section 8.4, it was observed that a large quantity of pre-executed loads were to constant addresses. Dynamically re-executing constant loads each loop iteration, even though data may never change, seems wasteful with respect to processor energy. An alternative implementation could save constant data alongside the load queue and maintain a constant entry across loop iterations, eliminating energy from repetitive cache accesses.

Additionally, as noted earlier in Section 5.3, it is possible an architecture could allow speculatively pre-executed load data propagate to dependent operations before validating the load's effective address. Although this operation is difficult to support in conventional out-of-orders, CRIB's simple re-execution mechanism could enable this operation. A study investigating the relative benefit could be performed.

Bit-Level CRIB Power Study

Within the CRIB architecture, logical register operands are passed along a specific column. This differs significantly from traditional out-of-order architectures that multiplex many differing logical registers on the same bypass paths. It is speculated that the operational energy of CRIB's operand network is significantly lower than was estimated in [32]. This may be the case because logical registers may be more likely to contain similar values over time, resulting in fewer switching events along the operand network.

Prefetcher and Processor Core Interaction

Finally, much effort went into verifying the operation and effectiveness of L1 cache prefetching for our respective architectures. During this effort it became clear that many processor core interactions can easily confuse L1 cache prefetchers. For example, in an out-of-order processor, although a code sequence may be regular and repetitive, the loads from within may be presented to the memory system in different orders. This is the case when absolute age-based instruction issue is not enforced. Although such re-orderings have no impact on functional correctness, potential confusion of the L1 cache prefetcher results from such nondeterminism. This impact was observed and corrected for during the early implementation of in-place load execution, however it seems largely unstudied within academia. To the best of our knowledge the only investigation of processor core and

prefetcher interactions has been in relation to instruction caching [28].

BIBLIOGRAPHY

[1] S.V. Adve and K. Gharachorloo. "Shared memory consistency models: a tutorial". In: *Computer* 29.12 (1996), pp. 66–76.

[2] Pritpal S. Ahuja, Douglas W. Clark, and Anne Rogers. "The performance impact of incomplete bypassing in processor pipelines". In: *Proceedings of the 28th annual international symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, United States: IEEE Computer Society Press, 1995, pp. 36–45.

[3] M. Alidina et al. "DSP16000: A High Performance, Low-Power Dual-MAC DSP Core for Communications Applications". In: *CICC'98*. 1998, pp. 119–122.

[4] *AMD Jaguar Software Optimization Guide*. URL: `http://support.amd.com/us/Processor_TechDocs/52128_16h_Software_Opt_Guide.zip`.

[5] Wilhelm Anacker and Chu Ping Wang. "Performance Evaluation of Computing Systems with Memory Hierarchies". In: *Electronic Computers, IEEE Transactions on* EC-16.6 (1967), pp. 764–773.

[6] *ARM Cortex A9*. URL: `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`.

[7] T.M. Austin and G.S. Sohi. "Zero-cycle loads: microarchitecture support for reducing load latency". In: *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*. 1995, pp. 82–92.

[8] John Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641.

[9] Jean-Loup Baer and Tien-Fu Chen. "An effective on-chip preloading scheme to reduce data access penalty". In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 176–186.

[10] R.S. Bajwa et al. "Instruction Buffering to Reduce Power in Processors for Signal Processing". In: *VLSI Systems, IEEE Transactions on* 5.4 (1997), pp. 417–424.

[11] N. Bellas et al. “Energy and Performance Improvements in Microprocessor Design Using a Loop Cache”. In: *ICCD-17*. 1999, pp. 378–383.

[12] David Bernstein, Doron Cohen, and Ari Freund. “Compiler techniques for data prefetching on the PowerPC”. In: *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*. PACT ’95. Limassol, Cyprus: IFIP Working Group on Algol, 1995, pp. 19–26.

[13] Nathan Binkert et al. “The gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7.

[14] Eric Borch et al. “Loose Loops Sink Chips”. In: *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 299–. URL: http://dl.acm.org/citation.cfm?id=874076.876467.

[15] Alper Buyuktosunoglu, Ali El-Moursy, and David H. Albonesi. “An Oldest-First Selection Logic Implementation for Non-Compacting Issue Queues”. In: *15TH INTERNATIONAL ASIC/SOC CONFERENCE*. 2002, pp. 31–35.

[16] H.W. Cain and P. Nagpurkar. “Runahead execution vs. conventional data prefetching in the IBM POWER6 microprocessor”. In: *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*. 2010, pp. 203–212.

[17] David Callahan, Ken Kennedy, and Allan Porterfield. “Software prefetching”. In: *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 40–52.

[18] R. Canal, J.-M. Parcerisa, and A. Gonzalez. “A cost-effective clustered architecture”. In: *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. 1999, pp. 160 – 168.

[19] George Z. Chrysos and Joel S. Emer. “Memory dependence prediction using store sets”. In: *Proceedings of the 25th annual international symposium on Computer architecture*. ISCA ’98. Barcelona, Spain: IEEE Computer Society, 1998, pp. 142–153.

[20] N. Clark, A. Hormati, and S. Mahlke. "VEAL: Virtualized Execution Accelerator for Loops". In: *ISCA-35*. 2008, pp. 389–400.

[21] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors". In: *Proceedings of the 1993 International Conference on Parallel Processing - Volume 01*. ICPP '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 56–63.

[22] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *Solid-State Circuits, IEEE Journal of* 9.5 (1974), pp. 256–268.

[23] J. J. Dongarra and A. R. Hinds. "Unrolling loops in fortran". In: *Software: Practice and Experience* 9.3 (1979), pp. 219–226. ISSN: 1097-024X.

[24] James Dundas and Trevor Mudge. "Improving data cache performance by pre-executing instructions under a cache miss". In: *Proceedings of the 11th international conference on Supercomputing*. ICS '97. Vienna, Austria: ACM, 1997, pp. 68–75.

[25] H. Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *ISCA-38*. 2011, pp. 365–376.

[26] Dave J Everitt. "Inexpensive Performance Using the Am29000". In: *Microprocessors and Microsystems* 14.6 (1990), pp. 397 –406.

[27] Keith I. Farkas et al. "The multicluster architecture: reducing cycle time through partitioning". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, United States: IEEE Computer Society, 1997, pp. 149–159.

[28] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. "Proactive instruction fetch". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44 '11. Porto Alegre, Brazil: ACM, 2011, pp. 152–162. ISBN: 978-1-4503-1053-6.

[29] Manoj Franklin and Gurindar S. Sohi. "Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors". In: *Proceedings of the 25th annual international symposium on Microarchitecture*. MICRO 25. Portland, Oregon, United States: IEEE Computer Society Press, 1992, pp. 236–245.

[30] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. "Compiler-directed data prefetching in multiprocessors with memory hierarchies". In: *Proceedings of the 4th international conference on Supercomputing*. ICS '90. Amsterdam, The Netherlands: ACM, 1990, pp. 354–368.

[31] E. Gunadi and M.H. Lipasti. "A position-insensitive finished store buffer". In: *Computer Design, 2007. ICCD 2007. 25th International Conference on*. 2007, pp. 105–112.

[32] Erika Gunadi and Mikko H. Lipasti. "CRIB: consolidated rename, issue, and bypass". In: *Proceedings of the 38th annual international symposium on Computer architecture*. ISCA '11. San Jose, California, USA: ACM, 2011, pp. 23–32.

[33] Matthew R Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE. 2001, pp. 3–14.

[34] Linley Gwennap. *Speed Kills? Not for RISC Processors*. 1993.

[35] Greg Hamerly et al. "Simpoint 3.0: Faster and More Flexible Program Phase Analysis". In: *Journal of Instruction Level Parallelism*. 2005.

[36] John L Henning. "SPEC CPU2006 Benchmark Descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[37] M.D. Hill and A.J. Smith. "Evaluating associativity in CPU caches". In: *Computers, IEEE Transactions on* 38.12 (1989), pp. 1612–1630.

[38] M. Hiraki et al. "Stage-Skip Pipeline: A Low Power Processor Architecture Using a Decoded Instruction Buffer". In: *ISLPED'96*. 1996, pp. 353–358.

[39] R. Ho, K.W. Mai, and M.A. Horowitz. "The future of wires". In: *Proceedings of the IEEE* 89.4 (2001), pp. 490 –504.

[40] J.S. Hu et al. "Scheduling Reusable Instructions for Power Reduction". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. Vol. 1. 2004, 148–153 Vol.1.

[41] Doug Joseph and Dirk Grunwald. "Prefetching using Markov predictors". In: *Proceedings of the 24th annual international symposium on Computer architecture*. ISCA '97. Denver, Colorado, USA: ACM, 1997, pp. 252–263.

[42] Norman P. Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”. In: *ISCA-17*. Seattle, Washington, USA: ACM, 1990, pp. 364–373.

[43] Gokul B. Kandiraju and Anand Sivasubramaniam. “Going the distance for TLB prefetching: an application-driven study”. In: *Proceedings of the 29th annual international symposium on Computer architecture*. ISCA ’02. Anchorage, Alaska: IEEE Computer Society, 2002, pp. 195–206.

[44] David Kanter. *IntelâŁ™s Haswell CPU Microarchitecture*. 2012. URL: http://www.realworldtech.com/haswell-cpu.

[45] David Kanter. *IntelâŁ™s Sandy Bridge Microarchitecture*. 2010. URL: http://www.realworldtech.com/sandy-bridge.

[46] David Kanter. *Silvermont, Intel’s Low Power Architecture*. 2013. URL: http://www.realworldtech.com/silvermont.

[47] R.E. Kessler, E.J. McLellan, and D.A. Webb. “The Alpha 21264 microprocessor architecture”. In: *Computer Design: VLSI in Computers and Processors, 1998. ICCD ’98. Proceedings. International Conference on*. 1998, pp. 90 –95.

[48] Hyesoon Kim et al. “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution”. In: *MICRO-38*. 2005, 12 pp.–54.

[49] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. “The Filter Cache: An Energy Efficient Memory Structure”. In: *MICRO-30*. Research Triangle Park, North Carolina, USA, 1997, pp. 184–193.

[50] Alexander C. Klaiber and Henry M. Levy. “An architecture for software-controlled data prefetching”. In: *Proceedings of the 18th annual international symposium on Computer architecture*. ISCA ’91. Toronto, Ontario, Canada: ACM, 1991, pp. 43–53.

[51] L. Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *Computers, IEEE Transactions on* C-28.9 (1979), pp. 690–691.

[52] Travis Lanier. *Exploring the Design of the Cortex-A15 Processor*. Tech. rep. ARM, 2011.

[53] J.K.F. Lee and A.J. Smith. “Branch Prediction Strategies and Branch Target Buffer Design”. In: *Computer* 17.1 (1984), pp. 6–22.

[54] Lea Hwang Lee, B. Moyer, and J. Arends. “Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops”. In: *ISLPED’99*. 1999, pp. 267–269.

[55] Sheng Li et al. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures”. In: *MICRO-42*. 2009, pp. 469–480.

[56] Changhui Lin et al. “Efficient sequential consistency via conflict ordering”. In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 273–286. ISBN: 978-1-4503-0759-8.

[57] Scott McFarling. *Combining Branch Predictors*. Tech. rep. HPL, 1993.

[58] Andreas Moshovos and Gurindar S. Sohi. “Streamlining inter-operation memory communication via data dependence prediction”. In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 235–245.

[59] K.J. Nesbit and J.E. Smith. “Data Cache Prefetching Using a Global History Buffer”. In: *Software, IEE Proceedings-*. 2004, pp. 96–96.

[60] *NVIDIA Tegra 4 Family CPU Architecture*. Tech. rep. NVIDIA, 2013. URL: `http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf`.

[61] Kunle Olukotun et al. “The case for a single-chip multiprocessor”. In: *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. ASPLOS VII. Cambridge, Massachusetts, USA: ACM, 1996, pp. 2–11.

[62] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. “Complexity-effective superscalar processors”. In: *Proceedings of the 24th annual international symposium on Computer architecture*. ISCA ’97. Denver, Colorado, United States: ACM, 1997, pp. 206–218.

[63] Frederico Pratas et al. “Low Power Microarchitecture with Instruction Reuse”. In: *CF-5*. Ischia, Italy: ACM, 2008, pp. 149–158.

[64] Steven Przybylski. “The performance impact of block sizes and fetch strategies”. In: *Proceedings of the 17th annual international symposium on Computer Architecture*. ISCA ’90. Seattle, Washington, USA: ACM, 1990, pp. 160–169.

[65] *Qualcomm’s New Snapdragon S4: MSM8960 & Krait Architecture Explored*. URL: http://www.anandtech.com/show/4940/qualcomm-new-snapdragon-s4-msm8960-krait-architecture.

[66] E. Rotenberg, Q. Jacobson, and J. Smith. “A Study of Control Independence in Superscalar Processors”. In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. HPCA ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 115–.

[67] E. Safi, A. Moshovos, and A. Veneris. “A physical level study and optimization of CAM-based checkpointed register alias table”. In: *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*. 2008, pp. 233–236.

[68] K. Sankaralingam et al. “Distributed Microarchitectural Protocols in the TRIPS Prototype Processor”. In: *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. 2006, pp. 480 –491.

[69] Peter G. Sassone et al. “Matrix Scheduler Reloaded”. In: *ISCA-34*. ISCA ’07. San Diego, California, USA: ACM, 2007, pp. 335–346.

[70] André Seznec. “A 256 kbits l-tage branch predictor”. In: *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007).

[71] Tingting Sha, Milo M. K. Martin, and Amir Roth. “NoSQ: Store-Load Communication without a Store Queue”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 285–296.

[72] Ronak Singhal. “Inside Intel Next Generation Nehalem Microarchitecture”. In: *Hot Chips*. Vol. 20. 2008.

[73] B. Sinharoy et al. “POWER5 System microarchitecture”. In: *IBM J. Res. Dev.* 49.4/5 (July 2005), pp. 505–521.

[74] A. J. Smith. “Sequential Program Prefetching in Memory Hierarchies”. In: *Computer* 11.12 (Dec. 1978), pp. 7–21.

[75] Alan Jay Smith. “Cache Memories”. In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530.

[76] J.E. Smith and A.R. Pleszkun. “Implementing precise interrupts in pipelined processors”. In: *Computers, IEEE Transactions on* 37.5 (1988), pp. 562–573.

[77] J.E. Smith et al. “The Astronautics ZS-1 processor”. In: *Computer Design: VLSI in Computers and Processors, 1988. ICCD ’88., Proceedings of the 1988 IEEE International Conference on*. 1988, pp. 307–310.

[78] Avinash Sodani. “Race to Exascale: Challenges and Opportunities”. In: *MICRO-44*. Presented at the 44th International Symposium on Microarchitecture, Porto Alegre, Brazil, 2011.

[79] G. S. Sohi and S. Vajapeyam. “Instruction issue logic for high-performance, interruptable pipelined processors”. In: *Proceedings of the 14th annual international symposium on Computer architecture*. ISCA ’87. Pittsburgh, Pennsylvania, USA: ACM, 1987, pp. 27–34.

[80] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. “Multiscalar processors”. In: *Proceedings of the 22nd annual international symposium on Computer architecture*. ISCA ’95. S. Margherita Ligure, Italy: ACM, 1995, pp. 414–425.

[81] Gurindar S. Sohi and Amir Roth. “Speculative Multithreaded Processors”. In: *Computer* 34.4 (Apr. 2001), pp. 66–73. ISSN: 0018-9162.

[82] Sam S. Stone, Kevin M. Woley, and Matthew I. Frank. “Address-Indexed Memory Disambiguation and Store-to-Load Forwarding”. In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 38. Barcelona, Spain: IEEE Computer Society, 2005, pp. 171–182.

[83] W. Tang, R. Gupta, and A. Nicolau. “Design of a predictive filter cache for energy savings in high performance processor architectures”. In: *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*. 2001, pp. 68–73.

[84] Michael Bedford Taylor et al. “The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs”. In: *IEEE Micro* 22.2 (Mar. 2002), pp. 25–35.

[85] J. M. Tendler et al. “POWER4 system microarchitecture”. In: *IBM Journal of Research and Development* 46.1 (2002), pp. 5 –25.

[86] R. M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33.

[87] Sravanthi Kota Venkata et al. “SD-VBS: The San Diego Vision Benchmark Suite”. In: *IISWC'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 55–64.

[88] Wikipedia. *Pareto principle — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-July-2013]. 2013. URL: \url{http://en.wikipedia.org/w/index.php?title=Pareto_principle&oldid=564323946}.

[89] Wm. A. Wulf and Sally A. McKee. “Hitting the memory wall: implications of the obvious”. In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24.

[90] Chengmo Yang and Alex Orailoglu. “Power-efficient instruction delivery through trace reuse”. In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. PACT '06. Seattle, Washington, USA: ACM, 2006, pp. 192–201.

[91] K.C. Yeager. “The Mips R10000 Superscalar Microprocessor”. In: *Micro, IEEE* 16.2 (1996), pp. 28–41.

[92] Victor V. Zyuban and Peter M. Kogge. “Inherently Lower-Power High-Performance Superscalar Architectures”. In: *IEEE Trans. Comput.* 50.3 (Mar. 2001), pp. 268–285.