Scrutinizing Resource Utilization for High Performance and Low Energy Computation

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In partial fulfillment

of the requirements for the degree

Doctor of Philosophy

Computer Engineering

by

Wei Zhang

December 2016

APPROVAL SHEET

This Dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Author Signature: 

This Dissertation has been read and approved by the examining committee:

Advisor: John Lach

Committee Member: Mircea Stan

Committee Member: James Aylor

Committee Member: Samira Khan

Committee Member: Jack Davidson

Committee Member:

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

December 2016

Abstract

Modern processors often suffer from inefficient resource utilization, which leads to inferior performance and energy efficiency. This dissertation scrutinizes the utilization of datapath and cache resources in superscalar processors for opportunities to improve performance and energy efficiency.

Traditional superscalar processors usually employ a one-size-fits-all design approach that allocates a fixed amount of resources for all applications at all times to deliver the best overall performance. However, the one-size-fits-all approach is not always energy efficient, because both the application behavior and the use scenario are changing all the time and the demand for processor resources is also changing accordingly.

To improve the utilization of datapath resources, this dissertation proposes an adaptive processor that dynamically allocates datapath resources based on the needs of applications and use scenarios. The adaptive processor is applied to two use cases to improve energy efficiency. In the first use case (front-end throttling (FET)), the adaptive processor dynamically throttles the front-end instruction delivery bandwidth as program behavior changes to optimize a target metric, being performance, energy, or an arbitrary trade-off between them. In the second use case (dynamic core scaling (DCS)), the adaptive processor extends performance-energy tradeoff capabilities in superscalar processors by scaling datapath resource rather than voltage. The adaptive processor ensures that programs run at a given percentage of their maximum speed and, at the same time, minimizes energy consumption by dynamically adjusting the active superscalar datapath resources. DCS is more effective in performance-energy tradeoffs than DVFS at the high performance end. When used together with DVFS, DCS significantly extends the range of performance-energy tradeoffs.

Caches also suffer from inefficient utilization in modern processors. To minimize the access

latency of set-associative caches, the data in all ways are read out in parallel with the tag lookup. However, this is energy inefficient, as only the data from the matching way is used and the others are discarded. To improve the utilization of the L1 instruction cache, this dissertation proposes an early tag lookup (ETL) technique for L1 instruction caches that determines the matching way one cycle earlier than the cache access, so that only the matching data way need to be accessed. ETL incurs no performance penalty and insignificant hardware overhead, but dramatically reduces the read energy of L1 instruction cache.

For memory intensive workloads, caches often suffer from thrashing, i.e., high-reuse blocks evicting each other from the cache due to the lack of space. To reduce thrashing, only a fraction of the working set should be kept in the cache, so that at least this fraction stays longer in the cache to enable reuse before eviction. However, prior insertion policies take an ad hoc approach to selecting that fraction, e.g., inserting blocks with high priority at fixed or randomly determined fractions, thus limiting the performance impact. This dissertation observes that the optimal fraction of the workload that should be kept in the cache is related to the cache block reuse distance distribution (RDD) of the application. Based on this observation, this dissertation provides an oracle analytical model to determine this optimal fraction assuming that the reuse distance (RD) of each block is known by oracle, and a practical model that is applicable without the oracle RDD. It then proposes simple runtime mechanisms to determine the optimal fraction for each workload dynamically. Our models are orthogonal to prior insertion policies and can significantly improve the performance of the prior state-of-the-art insertion policies when applied on top of them.

Evaluating new energy efficient processors requires accurate information of area, delay, and power of the new architectures. Prior works on energy efficient processor architectures either fail to obtain such information or rely on modeling frameworks such as Wattch and McPAT, which are of limited accuracy. Unlike prior works, this dissertation uses FabScalar, a circuit-level infrastructure, to accurately evaluate area, delay, and power of the new architectures.

Acknowledgments

I owe great thanks to my advisor, Professor John Lach, who has given me tireless support and mentoring throughout my PhD study. Prof. Lach has provided me valuable guidance on my research and motivated me to complete one project after another. I highly appreciate the tremendous amount of time and effort he spent on my research, which finally made it possible for me to complete this dissertation. I also greatly appreciate his support when I was exhaustively searching for my first job and my research lagged behind. His suggestions on making my final job decision were also greatly appreciated. I truly enjoy this mentorship with Professor Lach and it will be an unforgetable memory in my life.

I would like to thank Professor Samira Khan for her guidance on my last cache project. Many thanks to her for pushing me to develop the analytical models and eventually implement a new mechanism that finds the optimal fraction using the proposed analytical models. I really appreciate her great comments and edits of our cache paper, which significantly improved the quality of the paper.

I would like to thank Hang Zhang for his participation in the adaptive processor project and the low-power cache project. In addition, many thanks to him for his referral in Samsung when I was searching for my first job.

I would like to thank all the committee members for their time and involvement in my PhD proposal and defense. It was hard to schedule a time for my defense. Many thanks to them for filling out one doodle poll after another.

I would like to thank all my colleagues and friends who have worked and lived with me during my study at UVa for making my PhD life more enjoyable.

I would like to thank my parents for their spiritual support since the start of my PhD. Their encouragement made me strong and persevere in overcoming the difficulties encountered in my PhD life.

Finally, I would like to thank the funding agencies that supported my PhD research. The projects in this dissertation were funded in part by the National Science Foundation under grants EF-1124931 and IIS-1065262.

Wei Zhang

December 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance has always been the top priority when designing modern processors. Ever since the end of Dennard scaling [21], power wall has become the hurdle that prevents the performance of modern processors from growing. Nowadays, due to the thermal constraint, processors are usually designed with a limited power budget. Extracting every bit of performance with a limited power budget has become critical for processor companies to make their products competitive. Any reduction in power consumption without hurting performance will allow the processor to run at a higher frequency, thus improving performance.

However, power is not the most direct metric for measuring the efficiency of processors. Ultimately, the real concern is *the amount of energy consumed to complete a certain amount of work, defined as energy efficiency*. The less energy consumed to complete a certain amount of work, the more energy efficient the processor is. Not only does energy efficiency affect performance, but it also has a big impact on the battery life of mobile electronic devices and the operating costs of datacenters and warehouse-scale computers, a large fraction of which are incurred by electricity costs.

Thus, performance and energy efficiency have become two important considerations when designing modern processors. In the past decades, numerous architectural, circuit, and software techniques have been proposed to improve these two metrics. Yet, there are still many more opportunities to further push the limit of processor performance and reduce processor energy consumption. This dissertation focuses on using architectural techniques to realize these goals. More specifically,

it scrutinizes the utilization of datapath and cache resources in modern superscalar processors for opportunities to improve performance and energy efficiency.

Scrutinizing datapath resource utilization. Modern superscalar processors often use datapath resources aggressively to achieve high performance. They are usually designed with a one-size-fits-all approach that delivers the best overall performance for a wide variety of applications. The one-size-fits-all approach allocates a fixed amount of resources to all applications at all times, which, however, is not always efficient because both the application behavior and the use scenario are changing all the time.

Applications have diverse behavior, and even for a single application, its behavior changes from phase to phase. Thus, the demand for processor resources changes from application to application and from phase to phase. Always allocating a fixed amount of resources while ignoring application behavior often results in resource under-utilization, which degrades the energy efficiency.

In addition to application behavior, the use scenario is also changing all the time. Modern processors often need to switch among different power states based on the use scenario and the thermal condition. The performance of the core is often adjusted according to the workloads, the available power budget, and the chip temperature. In scenarios where high performance is demanded, the processor should run to deliver the highest performance. In scenarios where high performance is not needed (e.g., a user study [46] has shown that the low performance mode of an Intel Pentium CPU with dynamic voltage and frequency scaling already satisfies the users for many applications) or allowed (e.g., the power budget is constrained or the chip temperature is high), performance can be lowered to reduce energy consumption. However, with the one-size-fits-all approach, processors always operate with full set of resources, even in scenarios where high performance is not needed, thus hurting energy efficiency.

Scrutinizing cache resource utilization. Caches have a big impact on performance and consume a significant amount of energy in modern processors, as they are large, require high performance, and are accessed mostly every cycle. However, cache resources are not utilized efficiently in modern processors.

Set-associative caches are commonly used to reduce miss rate and achieve better performance.

To reduce the access latency of set-associative level-one (L1) caches, the data in all ways are read out in parallel with tag lookup. However, only the data in the matching way is used and the others are discarded, resulting in significant energy waste. Accessing the non-matching data ways do not contribute to getting the desired data and results in inefficient cache resource utilization.

The off-chip main memory is a major bottleneck for system performance, so modern processors employ a large last-level cache (LLC) to hide the long latency of memory accesses. However, the LLC is not utilized efficiently. Since the reference stream to the LLC is filtered by the L1 and L2 caches, it has been shown that a large portion of blocks in the LLC are not reused between insertion and eviction if using the traditional least recently used (LRU) replacement policy [4, 5, 39, 43, 59]. The LLC stores a large amount of useless information and is poorly utilized. With the increasing number of cores and new data intensive applications, it is now more important to manage cache space as efficiently as possible.

Goal. The goal of this dissertation is to improve processor performance and energy efficiency via more efficient utilization of datapath and cache resources. To improve the utilization of datapath resources, this dissertation proposes an adaptive processor that dynamically allocates datapath resources based on the needs of applications and use scenarios. The adaptive processor has better energy efficiency and can greatly improve power management. To improve the utilization of cache resources, this dissertation proposes a low power technique that dramatically reduces the power consumption of the L1 instruction cache without incurring any performance penalty, and an improved insertion policy that better utilizes the LLC and outperforms the recently proposed state-of-the-art insertion policies.

1.1 Improving Datapath Resource Utilization

Modern superscalar processors, such as IBM POWER8 [51], can use aggressive datapath resources to extract every bit of performance. These aggressive datapath resources, such as issue queue (IQ), load/store queue (LSQ), and reorder buffer (ROB), account for a significant fraction of energy consumption and often suffer from under-utilization when not needed by applications or use scenarios.

This dissertation proposes an adaptive processor that dynamically adjusts the active datapath resources in a large out-of-order core based on the needs of applications and use scenarios. The datapath resources that can be dynamically adjusted include front-end width, issue width, and sizes of IQ, LSQ, and ROB, all of which significantly influence performance and energy consumption. Two use cases are then studied to show that the adaptive processor has better energy efficiency and can greatly improve power management. The first use case applies the adaptive processor to throttle the front-end instruction delivery rate, which is able to improve energy efficiency in different use scenarios. The second use case applies the adaptive processor to trade off performance and energy by dynamically scaling the active datapath resources rather than voltage, which greatly extends the limit of DVFS in performance-energy trade-off.

1.1.1 Adaptive Front-End Throttling

Front-end instruction delivery consumes a significant fraction of energy and has a big impact on the performance of dynamically scheduled superscalar processors. To achieve high performance, the front-ends of conventional superscalar processors deliver instructions at peak rate at all times to expose as much Instruction-Level Parallelism (ILP) as possible. However, this fixed peak-rate instruction delivery scheme is often sub-optimal.

Previous work [8, 9, 23, 49] has shown that delivering instructions at peak rate often brings a large number of wrong-path and early-fetched instructions into the pipeline, causing energy waste. In scenarios where high performance is not required, instruction delivery could be slowed to save energy without sacrificing user satisfaction. However, the conventional peak-rate instruction delivery cannot adapt when the priority metric shifts between high performance and low energy consumption.

In addition, programs have diverse behaviors. The optimal instruction delivery rate that achieves the lowest energy consumption differs from program to program and from phase to phase in a single program. The conventional fixed-rate instruction delivery cannot adapt to the program behavior changes and often puts the superscalar processor in sub-optimal state.

Given the aforementioned problems, it is desirable to have the capability to dynamically ad-

just the instruction delivery rate of superscalar processors to adapt to changes in priority metrics and program behaviors. Based on the adaptive processor, this dissertation proposes an adaptive front-end throttling technique and circuit-level design that dynamically adjusts the front-end width, which controls the instruction delivery bandwidth, using software profiling or a run-time hardware controller to optimize a target metric, being performance, energy, or an arbitrary trade-off between them. In the software profiling approach, programs are analyzed prior to deployment and an optimal front-end width is selected for each program or phase given the target optimization metric. In the runtime approach, a hardware controller samples the program's execution information upon triggering events and automatically adjusts the front-end width to optimize the target metric.

1.1.2 Dynamic Core Scaling

Dynamic voltage and frequency scaling (DVFS) is an effective way to trade off performance and energy and has been commonly used in modern processors for power management. DVFS relies on scaling the supply voltage, whose upper-bound is determined by the nominal supply voltage V_{nom} and whose lower-bound is determined by the minimum operating voltage V_{min}. As transistor size scales down, V_{nom} tends to decrease, and V_{min} tends to increase due to the larger process variation and transistor count at smaller technology nodes [78]. Therefore, the voltage range where DVFS can operate is shrinking, which limits its effectiveness. Moreover, once the voltage is scaled to V_{min}, DVFS stops reducing energy.

It has been shown in [7] that small out-of-order cores tend to have lower performance but better energy efficiency, and large out-of-order cores tend to have better performance but worse energy efficiency. Based on this observation, this dissertation proposes dynamic core scaling (DCS) to extend the performance-energy trade-off capabilities in modern processors. DCS dynamically adjusts the active datapath resources in the adaptive processor and *allows it to run at a given percentage of its maximum speed while minimizing energy consumption*, creating a performance-energy trade-off by *scaling datapath resources rather than the supply voltage*. The maximum speed is the speed at which the core runs when its datapath resources are fully sized.

Prior works [8, 9, 14, 23, 26, 36, 49, 50, 54, 57] have proposed various energy-saving techniques

that dynamically allocate datapath resources according to the needs of applications. These energy-saving techniques suffer from two problems. First, they aim to allocate just the amount of datapath resources needed by the applications so as to reduce energy while incurring minimal performance penalty. Thus, their energy savings are limited due to the constraint that performance should not be hurt. Second, their resulting performance loss is often unpredictable and ranges from a few percent to more than 20% in worst cases. However, performance-energy trade-off often requires the ability to control performance when reducing energy. For example, when performance can be sacrificed to reduce energy consumption, it is desirable to run an application at a known percentage, say 70%, of its maximum speed rather than run it at a random reduced speed. Because of the limited energy savings and the lack of mechanisms to control performance, these previously proposed techniques have very limited ability to trade off performance and energy.

In contrast, this dissertation proposes a hardware controller to effectively manage performance-energy trade-off using DCS. The hardware DCS controller allows an arbitrary target performance (a certain percentage of the maximum performance) to be set and dynamically scales the datapath resources to minimize energy while trying to ensure that applications are run at the target performance, achieving much preciser and wider performance-energy trade-off.

In addition, DCS does not rely on voltage scaling and can be combined with DVFS to achieve greater energy savings. Combining DCS and DVFS to trade off performance and energy is challenging. This dissertation proposes three control mechanisms to effectively manage performance-energy trade-off using a combination of DCS and DVFS. First, an oracle controller is proposed to demonstrate the optimal control strategy when the resulting performance and energy of DCS is known by oracle. However, no such information is available at runtime. Then, two practical controllers that are applicable in real implementations are proposed, including a simple controller that performs effectively and a sophisticated controller that performs comparably to an oracle controller. Detailed analysis of these control mechanisms in different scenarios are also provided.

DCS is able to provide precise performance control and is more effective in performance-energy trade-off than DVFS at the high performance end. In addition, applying DCS together with DVFS greatly extends the limits of DVFS in performance-energy trade-off.

1.2 Improving Cache Resource Utilization

To improve the utilization of cache resources, this dissertation first proposes an early tag lookup (ETL) technique that is able to remove the majority of non-matching data way accesses in the L1 instruction cache and dramatically reduce power consumption without incurring any performance penalty. Then the dissertation proposes a dynamic insertion throttling (DIT) technique that better utilizes the LLC and improves its performance.

1.2.1 Early Tag Lookup

To reduce the access latency of set-associative L1 caches, the data in all ways are read out in parallel with tag lookup. However, only the data in the matching way is used and the others are discarded, resulting in significant energy waste. Techniques that reduce cache power consumption without hurting performance are therefore highly desirable.

A simple way to completely remove the accesses to the non-matching data ways is to perform tag lookup and data access sequentially, such as phased caches. The tag array is accessed first to get the matching way, and then (in the same cycle) the data array is accessed by reading out only the matching way, eliminating accesses to non-matching ways. However, such a 2-phase single-cycle method increases the cache access latency and hurts performance.

This dissertation proposes the early tag lookup (ETL) technique to reduce the read energy of set-associative L1 instruction caches. ETL is also a 2-phase method, but it tries to determine the matching way *one cycle earlier* than the actual cache access. If the matching way was determined successfully, the matching data way can be accessed directly, eliminating non-matching way accesses and saving energy. If determining the matching way failed, cache access happens as normal by accessing all the data ways, saving no energy. However, the first case accounts for more than 90% of the cache accesses, thus ETL can dramatically reduce the read energy of L1 instruction cache. In addition, since data access always succeeds no matter determining the matching way in the previous cycle is successful or not, ETL does not incur any performance penalty.

1.2.2 Dynamic Insertion Throttling

Recent works [4, 5, 31, 32, 34, 39, 43, 56, 59, 66, 73, 76] have shown that the traditional LRU replacement policy is highly ineffective for two kinds of workloads. *(i)* When the working set is larger than the cache size, high-reuse blocks evict each other due to the lack of space, making the cache ineffective. This problem is referred to as cache thrashing. *(ii)* When the working set has mixed reuse, little or no reuse blocks degrade performance by evicting blocks with high reuse from the cache. This problem is referred to as cache pollution. A large number of works propose insertion policies to solve these problems [31, 32, 34, 56, 59, 66, 73, 76]. To solve thrashing, recent insertion policies protect a fraction of the working set in the cache (e.g., BIP [59], BRRIP [32]) by inserting most of the missed cache blocks with low priority and only a small fraction (typically referred to as $1/\varepsilon$) of the missed blocks with high priority. Blocks inserted with low priority get evicted from the cache quickly, protecting a fraction of the working set in the cache long enough to get reused. To solve pollution, recently proposed insertion policies predict the reuse behavior of missed cache blocks and only insert high-reuse blocks with high priority so that these blocks can get reused (e.g., SHiP [76], EAF [66]).

The Problem. The aforementioned insertion policies all depend on inserting only a fraction of the cache blocks with high priority to improve performance. Unfortunately, their approaches are ad hoc. The works addressing thrashing empirically determine the fraction ($1/\varepsilon$) of the working set kept in the cache for thrashing workloads (ε is 32 or 64) [32, 59]. The works addressing pollution attempt to predict the high-reuse blocks and *always* insert them with high priority [66, 76]. However, such a mechanism without the knowledge of the optimal fraction cannot fully address thrashing and can cause cache under-utilization. In this work, we argue that *maximizing performance by addressing cache pollution and thrashing depends on accurately determining the optimal fraction of the working set that should be kept in the cache*.

Our Goals. Our goals are twofold. First, we want to provide an understanding of how insertion policies help improve cache performance and (with that understanding) develop an analytic model to determine the optimal workload fraction that should be kept in the cache to maximize performance. Second, we want to develop and demonstrate a simple and efficient mechanism to

determine this optimal fraction dynamically for each workload at runtime.

This dissertation demonstrates that the optimal fraction depends on the reuse distance distribution (RDD) of the workloads. An oracle reuse model is first proposed to determine this optimal fraction when the insertion policy has the accurate knowledge about the reuse distance (RD) of the workload. Unfortunately, in practical cases, it is hard to have precise information about the RD of the workload. Two practical models are then proposed to determine the optimal fraction in real implementations.

Based on the practical models, we propose Dynamic Insertion Throttling (DIT) to insert the optimal fraction of the cache blocks with high priority that maximizes hit rate. We propose two simple and effective mechanisms to implement DIT. DIT is independent of the insertion policy used in the cache and can improve any prior insertion policy.

1.3 Circuit Infrastructure for Accurate Evaluations

Any new energy efficient processors need to be validated, which usually requires an accurate evaluation of the area and delay overhead incurred by the new architecture and the resulting energy savings. Prior works on energy efficient processor architectures either fail to obtain such information or rely on modeling frameworks such as Wattch [13] and McPAT [45], which are of limited accuracy. To accurately evaluate the area, delay, and power of the new architecture, this dissertation utilizes a circuit infrastructure, FabScalar [17], which is an open-source tool that automatically generates synthesizable RTL code of diverse superscalar cores. Included in the infrastructure is FabMem, a multi-ported RAM/CAM compiler that can estimate read/write times and energies and areas of user-specified RAMs/CAMs and can generate layouts of the desired RAMs/CAMs. A couple of modifications are made to the FabScalar tool to meet the project needs. A two-level cache is added to FabScalar (no cache originally), and a store-set memory dependence predictor is also added to solve a performance bug (performance decreases as instruction window increases due to load violations). Multiple memory address generation units are added and the load/store unit is redesigned to process multiple loads/stores per cycle.

The area, delay, and power of the new architecture are accurately evaluated using the circuit-level infrastructure. The RTL processor is synthesized using the FreePDK 45nm library [70] to generate the gate-level netlist and the area and timing reports. Gate-level simulation is then performed to collect the switching activity of the processor when executing benchmarks. The switching activity is used to annotate the gate-level netlist, which the power compiler takes to generate the power report. The power of the unsynthesizable SRAM/CAM blocks are estimated using FabMem. Because FabMem does not support large-size SRAMs/CAMs, the power of caches are estimated using CACTI [1].

1.4 Dissertation Contributions

This dissertation identifies the problem that the datapath and cache resources in modern superscalar processors are inefficiently utilized, leading to suboptimal performance and energy efficiency. The main contributions of this dissertation are the four techniques proposed to improve processor resource utilization and thus its performance and energy efficiency. The contributions of each proposed technique are summarized as follows.

Front-End Throttling (FET) dynamically adjusts the front-end width, which controls the instruction delivery bandwidth, to optimize a target metric, being performance, energy, or an arbitrary trade-off between them. The contributions of FET are as follows:

- Performed architectural simulations to show that the optimal front-end instruction delivery bandwidth varies as target optimization metric or program behavior changes.
- Proposed a hardware controller that effectively manages front-end throttling to optimize a given target metric for superscalar processors.
- Performed circuit-level synthesis (45nm FreePDK) and simulations to accurately evaluate the overhead of adaptive front-end throttling and quantify the resulting energy savings.

Dynamic Core Scaling (DCS) extends the performance-energy trade-off capabilities in superscalar processors. The contributions of DCS are as follows:

- Proposed DCS as an alternative to DVFS that trades off performance and energy by dynamically scaling datapath resources rather than voltage.
- Proposed a hardware DCS controller that minimizes energy and, at the same time, tries to ensure that applications run at a given percentage of the maximum speed they would achieve if datapath resources were fully allocated.
- Proposed an oracle controller, a simple controller, and a sophisticated controller for combining DCS and DVFS to extend performance-energy trade-off, and provided detailed analysis of these control mechanisms in different scenarios.
- Demonstrated the effectiveness of DCS and DCS+DVFS in performance-energy trade-off and compared them with DVFS alone.
- Accurately evaluated the overhead of DCS and the resulting energy savings using 45nm circuit infrastructure.

Early Tag Lookup (ETL) removes the majority of non-matching data way accesses and dramatically reduces energy consumption in L1 instruction caches. ETL has the following advantages over previous works:

- ETL does not cause any performance penalty. Since instruction cache misses can be discovered one cycle earlier, ETL even improves performance slightly.
- ETL only needs a few simple and low-cost extensions to the existing hardware, and does not incur any significant design complexity.
- ETL is more effective at removing non-matching data way accesses than way prediction and L0 caches, saving significant dynamic energy in L1 instruction caches.

Dynamic Insertion Throttling (DIT) determines the optimal fraction of the cache blocks that should be inserted with high priority in the LLC at runtime so that the hit rate is maximized. The contributions of DIT are as follows:

- Demonstrated that maximizing performance for cache polluting and thrashing workloads requires an *optimal* fraction of the working set to be in the cache, and that prior insertion policies that try to solve cache pollution and thrashing empirically determine this fraction and, as a result, cannot maximize the performance.

- Provided an oracle model to determine this optimal fraction when the RD of each block is already known. The model builds upon the key insight that by inserting only a fraction of the blocks with a high priority, insertion policies effectively reduce the RD of blocks in the cache. These blocks receive hits only if the effective RD becomes smaller than the cache associativity. We show that it is possible to determine the optimal RD such that the fraction of blocks with effective RD smaller than the cache associativity can be maximized.

- Provided two practical models based on the RDD when the oracle RD information of each block is not available. *(i)* EBM treats all blocks equally and is applicable for simple insertion policies (e.g., BIP, BRRIP) *(ii)* RDM differentiates high and low RD blocks and is applicable for insertion policies that depend on prediction mechanisms (e.g., SHiP, EAF).

- Proposed two simple mechanisms, DIT-RD and DIT-SM, to determine the optimal fraction dynamically at runtime. Experimentally demonstrated that our mechanisms (when applied on top of prior insertion policies - BRRIP, SHiP and EAF) significantly improve performance over a wide range of workloads in single/4-core configurations.

1.5 Dissertation Organization

Chapter 1, this chapter, identifies the resource utilization problem in modern processors, provides the background information, and introduces the four techniques that this dissertation proposes to improve processor resource utilization for better performance and energy efficiency.

Chapter 2 introduces the prior works that are closely related to this dissertation and differentiates this dissertation from them.

Chapter 3 presents adaptive front-end throttling, its hardware controller, and the evaluation results [81].

Chapter 4 presents dynamic core scaling, DCS control, combining DCS with DVFS for greater energy savings, and the evaluation results [82].

Chapter 5 presents early tag lookup, its implementation, and the evaluation results [83].

Chapter 6 presents dynamic insertion throttling, the analytical models that determine the optimal fraction, DIT-RD and DIT-SM implementations, and the evaluation results.

Chapter 7 concludes the dissertation.

Chapter 2

Related Work

This chapter looks at prior works that are closely related to this dissertation and differentiates this dissertation from them.

2.1 Front-End Throttling

Throttling front-end instruction delivery bandwidth helps reduce the number of wrong-path and early-fetched instructions delivered into the pipeline, improving processor energy efficiency. Pipeline gating [49] stalls instruction fetch when a low-confidence branch instruction is encountered to reduce wrong-path instructions. Just in time instruction delivery [36] stalls instruction fetch when the number of in-flight instructions exceeds a threshold. Instruction flow based front-end throttling [9] throttles instruction fetch to match the decode rate with the commit rate and reduces early-fetched instructions. Selective throttling [6] focuses on reducing energy dissipated by wrong-path instructions and dynamically chooses the optimal throttling technique applied to each branch depending on the branch prediction confidence level. Fetch halting [50] stops fetching instructions when long-latency critical load misses occur to reduce early-fetched instructions.

Compiler-based fetch throttling techniques were proposed in [74, 75], where the compiler is used to analyze the ILP of the program and instruction fetch is stalled when the program's ILP is low. Software-directed fetch throttling was proposed in [35], in which the processor uses information gained from the compiler to dynamically resize the issue queue to reduce power consumption.

However, these existing front-end throttling techniques all focus on reducing energy consumption while minimizing performance loss. Thus they cannot adapt to use scenario changes where the priority metric changes between high performance and low energy. In contrast, the front-end throttling technique proposed in this dissertation can optimize an arbitrary priority metric, is orthogonal to, and can even leverage, most existing techniques, providing even greater savings. In addition, the previous works either do not have a direct way to quantify the overhead of the throttling technique and the resulting energy savings, or get this information relying on architecture-level modeling frameworks, such as Wattch [13] and McPAT [45], which are known to have limited accuracy. In this proposal, front-end throttling is implemented at the register transfer level (RTL), and circuit-level synthesis and simulation are used to accurately analyze the area, delay, and power overhead of the throttling technique and resulting energy savings.

2.2 Dynamic Datapath Resource Scaling

Many prior works have studied scaling datapath resources to save energy in superscalar processors. These works can be grouped into three categories and DCS will be compared with each of them.

2.2.1 Saving Energy with Minimal Performance Penalty

In [57], the sizes of IQ, LSQ, and ROB are dynamically adjusted based on the demands of applications to save energy while minimizing performance loss. In [26], the major SRAM and CAM structures are dynamically resized during cache miss events to reduce energy dissipated by these structures. In [23], the IQ is dynamically resized to reduce the wake-up of empty entries and operands that are ready, saving energy with minimal impact on performance. In [14], energy is saved by matching the instruction fetch rate and the IQ size to the parallelism of the application. In [54], an IQ adaptation mechanism is proposed to avoid hurting memory level parallelism and reduce performance penalty associated with IQ adaptation. Pipeline balancing [8] dynamically adjusts the issue width to reduce energy in IQ and execution units.

All these previous techniques suffer from two limitations. First, their energy saving efforts cannot tolerate any significant performance penalty, thus limiting the resulting energy savings. In contrast, DCS can achieve a much wider trade-off range of performance and energy. Second, these previous techniques have no mechanisms to control performance and the resulting performance is random, limiting their ability to trade off performance and energy. In contrast, DCS is able to maintain a variety of target performances through dynamic feedback while still minimizing energy at the same time.

2.2.2 Pipeline Stage Unification

Dynamic pipeline scaling [42] and pipeline stage unification [68] lower the clock frequency and merge pipeline stages to form shallower pipelines, which have better IPC than deep pipelines, thus saving energy without scaling voltage. However, energy saving is limited by the number of pipeline stages that can be merged. The resulting performance is also very restricted. For example, merging two pipeline stages reduce performance by half, resulting in sub-optimal performance-energy trade-off. In contrast, DCS is much more fine-grained.

2.2.3 Other Energy-Aware Processors

Flicker [55] divides a single core into equal slices and selectively powers on and off single slices to trade off performance and energy. However, this approach is course-grained and performance-energy trade-off is limited by the number of slices. In addition, datapath resources are allocated by slice and this often results in suboptimal allocation as applications' demands are quite diverse. For example, Flicker puts fetch, decode, ROB, rename, and dispatch in one pipeline region, and increasing ROB must increase front-end width, which may often be sub-optimal since programs have diverse and changing demands on front-end widths. In contrast, DCS achieves fine-grained performance-energy trade-off and balanced resource allocation by fine tuning the microarchitecture parameters.

Core fusion [28] morphs groups of fundamentally independent cores into a larger CPU, or uses them as distinct processing elements, according to the runtime needs of applications. Morph

Core [40] dynamically transforms a traditional high performance out-of-order core into a highly-threaded in-order SMT core when necessary. Both of them focus on exploiting both ILP and TLP using the same reconfigurable hardware. Whereas DCS focuses differently on exploiting dynamic reconfiguration to trade off performance and energy beyond DVFS.

Multiple Clock Domain (MCD) processors [29, 65] use independent frequency/voltage domains, and reduce the frequency of some domains if possible to save energy without significantly impacting performance. While DCS does not consider increasing clock frequency when resource scaling reduces critical path delay, the MCD technique can be applied together with DCS and will further improve the results.

2.3 Reducing Set-Associative Cache Energy

Previous work on cache energy reduction can be classified into three categories: way prediction, L0 caches, and acquiring matching way early. ETL is more effective at removing non-matching way accesses than way prediction and L0 caches, and incurs no performance penalty and low hardware overhead.

2.3.1 Way Prediction

Various way prediction techniques have been proposed to reduce the energy of set-associative caches. In [27], the matching way is predicted using a most recently used algorithm. In [58], way-prediction [12, 15] and selective direct-mapping [12] are used to predict the matching way. In [71], BTB is extended by caching way predictions, and way prediction is done in parallel with branch target prediction so that the matching way is known by the time cache access happens.

The aforementioned way prediction techniques suffer from two drawbacks. One is that they incur performance penalty when the the prediction is wrong, because an extra cycle is needed to access the correct way, increasing cache access latency. The other is that they require significant amount of extra hardware to do way prediction. In [58], additional tables with significant sizes are required to store way prediction information. In [71], non-branch instructions are also stored in

BTB, significantly increasing the size of BTB as well as BTB power consumption. In comparison, ETL does not cause any performance penalty and only needs some simple low-cost extensions to the existing hardware.

2.3.2 L0 Caches

The filter cache [41] is a small low-power cache placed between the CPU and the L1 cache. It reduces direct accesses to the L1 cache, thus saving energy. The L-Cache [24] is an additional mini cache located between the CPU and the instruction cache. It saves energy by buffering instructions that are nested within loops and are otherwise fetched from the instruction cache. The HotSpot cache [77] adds a steering mechanism to the filter cache and improves the L0 cache utilization.

One drawback of the L0 caches is that they may significantly degrade performance, because cache access latency increases when the access misses the L0 cache. Although HotSpot is claimed to incur no performance penalty, the evaluations in [77] indeed show a worst-case 2-3% performance degradation. In comparison, ETL does not incur any performance penalty. Another drawback of the L0 caches is that they need an extra cache, which increases design complexity and hardware costs. While ETL does not need any extra piece of complex hardware and incurs insignificant overhead.

2.3.3 Acquiring Matching Way Early

Tag check elision [84] determines the matching way early in the pipeline by doing a memory address bounds check. Speculative tag access [10] speculatively compares tags earlier in the pipeline to determine the matching way prior to cache access. The early tag access (ETA) cache [19] assumes that the accesses to the LSQ and L1 data cache are performed in series in embedded processors. Thus, ETA can be performed before L1 data cache access. The early load data dependency detection technique [11] sequentially accesses the tag and data array when there is data dependency that will cause stall cycles for in-order pipelines. However, accessing the tag array and the data array sequentially is slow and not commonly used in high-performance processors. In addition, all these

techniques only apply to the data cache and not the instruction cache. While ETL focuses on reducing L1 instruction cache energy.

The Tag-Less Cache (TLC) [64] adds way information to the TLB and looks up the information before accessing the cache, thus accessing only the matching way. However, it accesses the TLB first to obtain the matching way before accessing the cache, which increases cache access latency. Although a micro-page sub-banking technique is proposed to reduce the increased latency, the technique adds extra complexity to the design of TLB and cache. Way-halting cache [80] and cache line address locality [52] both store way information in a fully associative cache and search it to determine the matching way before accessing the L1 cache. However, searching the way information cache increases access latency and hurts performance. In comparison, ETL does not increase cache access latency at all. The implementation efforts needed by ETL are low.

2.4 Replacement and Insertion Policies for LLC

Recent works [4,5,31,32,34,39,43,56,59,66,73,76] have shown that the traditional LRU replacement policy is highly ineffective for workloads with cache thrashing and pollution. A large number of works propose insertion policies to solve these problems [16,18,31,32,34,44,56,59,61,66,73,76] by either statistically keeping a fraction of the working set in the cache or exploiting special mechanisms to differentiate the reuse locality of cache blocks and use that information to assist cache block bypassing or insertion.

The aforementioned insertion policies all depend on inserting only a fraction of the cache blocks with high priority to improve performance. However, the fraction of the cache blocks these policies keep in the cache is either fixed or randomly determined, which often leads to suboptimal performance. This dissertation demonstrates that maximizing performance for cache polluting and thrashing workloads requires an optimal fraction of the working set to be in the cache, and builds analytical models to provide an understanding of how the RDD of workloads affects the optimal fraction and proposes simple and efficient mechanisms to determine this optimal fraction at runtime. However, none of the aforementioned policies provide any insight into this problem.

Many cache replacement policies have been proposed in the past to improve cache performance [20, 25, 32, 33, 60, 63]. Instead of focusing on inserting missed cache blocks, these policies take various approaches to find the best victim blocks to replace on cache misses. Dead block prediction predicts dead blocks that are unlikely to be reused in the near future and replace them on cache misses [38, 39, 43, 47]. Reuse distance prediction directly predicts the reuse distances of cache references [37]. The prediction information is then used to make cache replacement decisions. Protecting distance (PD) is proposed to prevent replacing a cache block until a certain number of accesses occur to the cache set to improve cache hit rate [22]. DIT is independent of all these aforementioned cache replacement policies and can be combined with them to further improve cache performance.

Chapter 3

Front-End Throttling

3.1 The Need for Flexibility

The necessity of a flexible front-end instruction delivery scheme that dynamically adapts to changes in priority metrics and program behaviors is demonstrated in Figure 3.1, which plots the performance and energy consumption of several SPEC CPU2000 benchmarks executed on an 8-way superscalar processor under different front-end widths. The configurations of the 8-way processor are shown in Table 3.1.

The optimal front-end width changes as the priority metric changes. When the priority metric is high performance, width 8 is optimal for *gcc*, however, when the priority metric becomes low energy, width 1 is optimal for *gcc*.

The optimal front-end width also changes as the program changes. For example, the lowest energy consumption for *gcc* happens at front-end width 1, while this happens for *bzip* at width 6. Even for a single program, such as *bzip*, the optimal front-end widths in different phases are different. For example, the lowest energy consumption for *bzip-p1* and *bzip-p2*, representing two phases of *bzip*, happens at front-end width 6 and 2, respectively.

Given these observations, a fixed conventional superscalar processor cannot adapt to priority metric changes or program behavior changes and often works in a suboptimal state, making a flexible front-end instruction delivery scheme that dynamically adapts to these changes highly desirable.

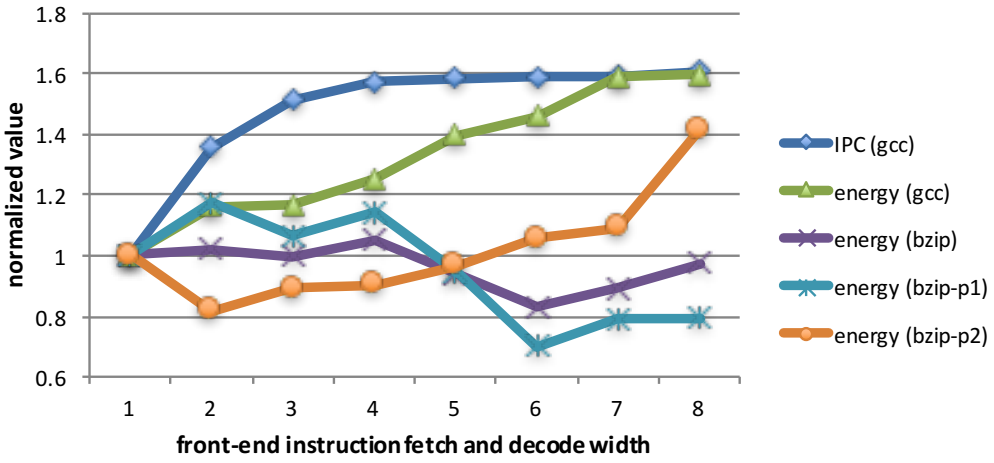


Figure 3.1: IPC and energy under different front-end widths of an 8-way superscalar processor. Energy is measured for a fixed amount of workload.

3.2 Adaptive Front-End Throttling

Adaptive front-end throttling dynamically searches the optimal front-end width for the target metric by comparing program execution information under all widths and adjusts the front-end to that optimal width. Figure 3.2 illustrates how the flexible front-end instruction delivery works. The instruction delivery paths, consisting of fetch, decode, rename, and dispatch, are enabled or disabled dynamically by the control registers, which are added to each pipeline stage of the front-end. Corresponding logic in each pipeline stage is modified if necessary to accommodate the flexible instruction processing width. The instruction delivery paths to the branch predictor, branch target buffer, decode units, instruction buffer, rename map tables, dispatch logic, issue queue, and reorder buffer are all selectively enabled or disabled by the control registers. The instruction cache only fetches the required number of instructions dictated by the control register. Unused pipeline resources are clock-gated, but are still powered on.

Special instructions are added to the Instruction Set Architecture to directly access the width control registers, allowing software or compiler based approaches to control front-end throttling. Front-end width changes in a pipeline fashion, starting from the fetch stage and propagating down to the subsequent stages until instructions reach the issue queue and the reorder buffer. Width

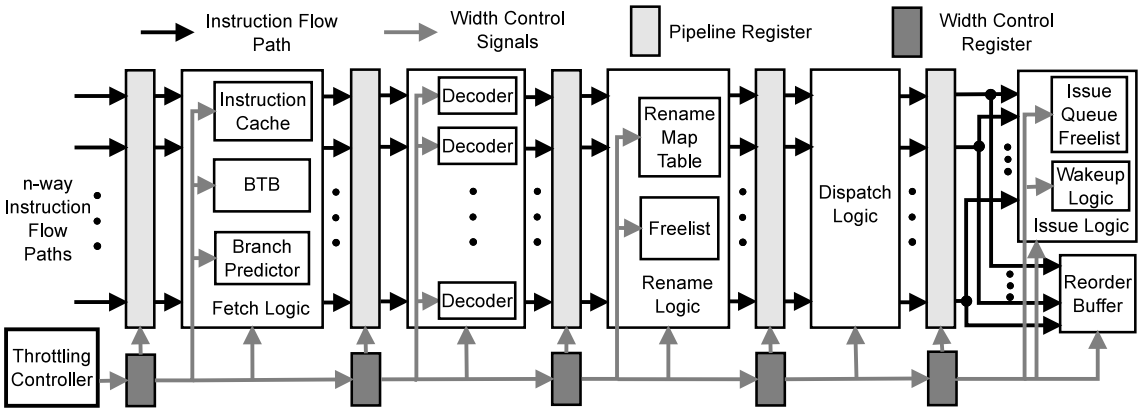


Figure 3.2: Flexible front-end instruction delivery.

change can happen in one cycle for a single pipeline stage and does not need to be consecutive. It can jump arbitrarily between any two values ranging from one to the maximum allowable width.

The basic unit that adaptive front-end throttling performs optimization on is an instruction chunk, which consists of a fixed number of instructions. Programs are divided sequentially into equal-sized instruction chunks, and optimization is made chunk-wise. Various control techniques can be proposed to dynamically throttle the front-end width. In this work, software profiling and run-time hardware controller are proposed as two approaches to throttle the front-end.

3.2.1 Software Profiling

Dynamic program behavior can be analyzed by software profiling, from which the front-end width control information can be extracted and inserted into the programs. To find the optimal width, programs are run once under each front-end width prior to deployment, during which performance and average power consumption of executing each chunk of instructions under that width are collected. Using this information, two types of optimization are applied to the programs. One is optimization by program, which uses a single fixed front-end width throughout the whole program but this width is flexible when choosing it. The target optimization metric for executing the entire program under each front-end width is calculated, and the width that achieves the best result for the target metric is chosen as the optimal front-end width for that program under that metric. For example, on a 4-way superscalar processor, the same program is executed four times under each of the four front-end

widths. In each execution, the performance and average power of executing the entire program are collected. Using this information, the target optimization metric under all four different widths are calculated and compared. The width that achieves the best result for the target metric is selected as the optimal width.

The other is optimization by phase, which allows a single program to use different optimal front-end widths for different instruction chunks during execution. Optimization is made chunk-wise, in which the target optimization metric for executing a single chunk of instructions under each front-end width is calculated, and the width that achieves the best result for the target metric is chosen as the optimal front-end width for that chunk of instructions under that given metric. The above optimization process is repeated for each chunk of instructions in the program. For example, on a 4-way superscalar processor, the same program is executed one time under each of the four front-end widths. During each execution, the performance and average power consumption of executing each chunk of instructions in the program are collected. For every chunk of instructions in the program, the target optimization metric for that instruction chunk under each width from 1 to 4 is calculated and the width that performs the best is selected as the optimal front-end width for that instruction chunk under that metric.

Width control information can be inserted into the program by compilers via special width control instructions. Alternately, the width control information can be stored in a hardware controller that throttles the front-end width when the width changing point arrives during program execution.

3.2.2 Run-Time Hardware Controller

Although software profiling is useful for characterizing dynamic program behavior, its efficacy of identifying optimal widths could be degraded when program input changes. To effectively capture dynamic program behavior changes, a dedicated hardware controller is proposed to throttle the front-end width during run-time. The hardware controller samples the program's execution information and uses the sampling results to set the optimal front-end width in the near future. The rationale behind sampling is that programs have such temporal locality that the front-end width that achieves the optimal result for the target metric at present tends to achieve the optimal result in the

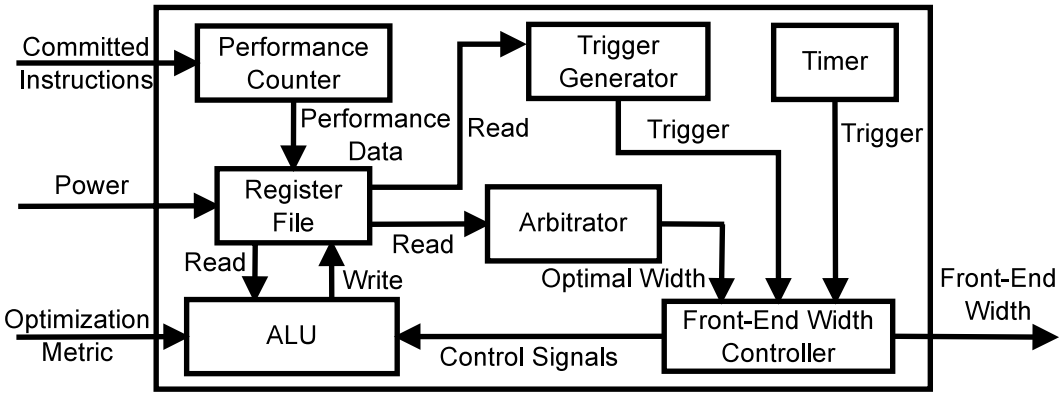


Figure 3.3: Block diagram of the hardware controller.

near future as well. As observed in architectural simulations, this temporal locality can be as long as the time taken to commit hundreds of millions of instructions. The question is when to sample? Software profiling shows that program behavior changes are often accompanied by performance and power changes. The hardware controller uses sudden changes in performance or power as indications of potential program behavior changes, which may necessitate optimal front-end width changes. A number of most recent performance and power data are kept by the hardware controller and their average values are dynamically calculated. When the difference between the new performance or power sample and the average historical performance or power sample, defined as the absolute value of $(newSample - averageSample)/averageSample$, exceeds a threshold, a sudden change in performance or power is identified. The hardware controller monitors the performance and average power of executing each chunk of instructions, and upon detecting a sudden change in either performance or power triggers a sampling process.

The block diagram of the hardware controller is shown in Figure 3.3. The performance counter counts the number of cycles taken to commit every chunk of instructions. An on-chip digital power meter, such as the one in the Intel Sandy Bridge microprocessor [62], is assumed to have already been built on the processor chip that provides the power information. Performance and power data are written into the register file. The trigger generator triggers a sampling process when detecting a sudden performance or power change. A timer generates periodic trigger signals in case the trigger generator misses certain program behavior changes. The target optimization metric for every single

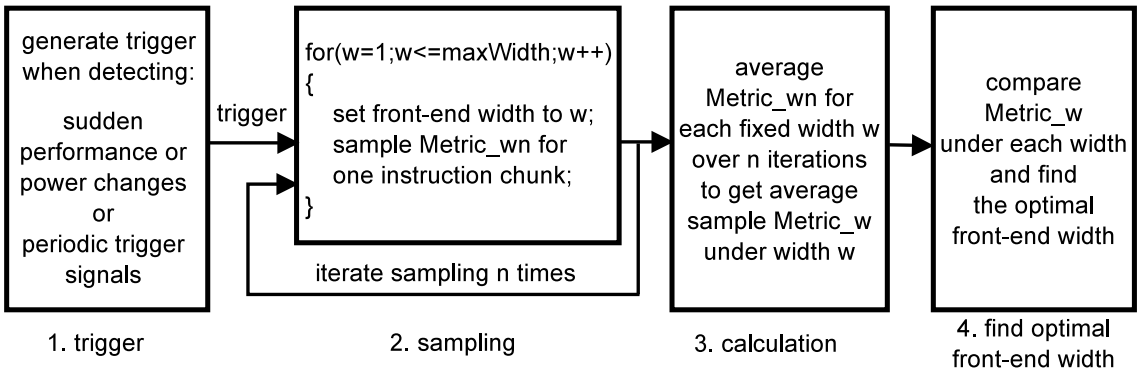


Figure 3.4: Run-time control flow. $Metric_{wn}$ is the sampling result of the target metric under width w in sampling iteration n. $Metric_w$ is the average sampling result of the target metric under width w. $maxWidth$ is the maximum front-end width.

chunk of instructions is calculated in the arithmetic logic unit (ALU), and the results are stored in the register file. The arbitrator compares the sampling results and determines the optimal front-end width. The width controller manages the sampling process and sets the front-end width.

Figure 3.4 shows the major steps in the control flow using the hardware controller. In step 1, a trigger signal is generated if any of the following events is detected: sudden performance change, sudden power change, and periodic trigger signals. In step 2, upon detecting a trigger signal, the controller initiates a sampling process which consists of n iterations. In each iteration, the controller takes a sample under each front-end width, starting from one and increasing the width until the maximum, by first setting the front-end to that width and then sampling the target metric for executing a single chunk of instructions under that width. Figure 3.5 shows an example of the sampling process on a 4-way superscalar processor. In each sampling iteration, the controller first sets the front-end width to 1 and takes a sample, then sets the front-end width to 2 and takes a sample, and repeats this until width 4 is sampled. Next, the above sampling iteration is iterated n times. The reason for making n sampling iterations is as follows. The performance and power profile of program execution often fluctuates and Figure 3.6 shows an example of the performance fluctuations over a few instruction chunks on a 4-way superscalar processor. If a sampling iteration (s1, s2, s3, and s4) happens in locations shown in Figure 3.6, it would give the result that width 1 yields the best performance, which is not true. The sample under width 1 (s1) just happens to

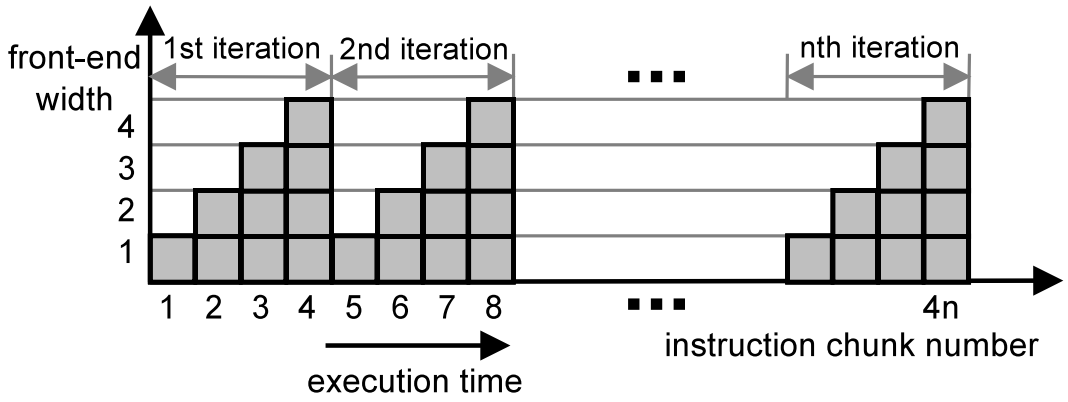


Figure 3.5: Sampling process on a 4-way processor.

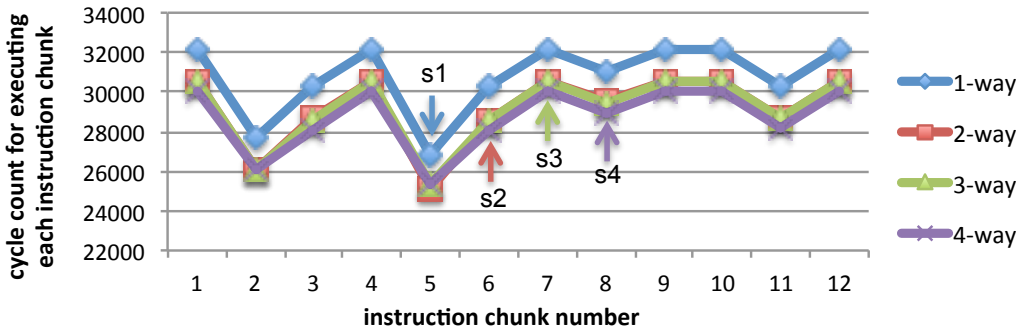


Figure 3.6: An example of performance fluctuations in *gcc* on a 4-way processor. s1-s4 represent 4 sampling points under front-end width 1 to 4, respectively.

be in the valley. To reduce the influence of program fluctuations, the same sampling iteration is repeated n times to average out the fluctuations. Because of the inherent temporal locality, despite the large fluctuations in some programs the iterated sampling is still very effective as verified by the evaluation results in Section 3.3. The workload does not have to be very stable in order for the sampling process to work. In step 3, the sampling results under the same front-end width are averaged over n iterations. In step 4, an optimal front-end width is selected by comparing the average sampling results of the target metric under all widths. The width that gives the best result is selected as the optimal width and used to set the front-end.

The run-time controller incurs control overhead, which comes from two sources. One is that the controller needs to try suboptimal front-end widths in the sampling process before settling on the optimal width. To limit this overhead, a minimum interval is enforced to prevent overly

Table 3.1: Configurations of experimental processors.

Core type	**4-way**	**6-way**	**8-way**
Front-end width	4	6	8
Issue width	4	6	8
Functional units (simple,mult./div., branch,ld/st)	1,1,1,1	3,1,1,1	5,1,1,1
Issue queue	32	64	128
Load/Store queue	32	32	64
ROB	128	256	512
Branch predictor, BTB	bimodal, 64K branch history table, 4K BTB		
L1 I-Cache	32K, 64-byte block, 4-way, 1 cycle		
L1 D-Cache	64K, 64-byte block, 4-way, 1 cycle		
Unified L2	2M, 64-byte block, 8-way, 18 cycles		

frequent sampling. After a sampling process is triggered, a second sampling is not allowed until the minimum interval elapses. Because of the long temporal locality, a properly chosen minimum interval won't miss too many optimal front-end width changes. The other is that the run-time controller may fail to detect some program behavior changes and use suboptimal front-end widths for those program phases. To reduce such misses of program behavior changes, periodic trigger signals are generated by the timer to trigger a sample if no program behavior change is detected for too long.

3.3 Evaluation

3.3.1 Experimental Infrastructure

This work utilizes FabScalar [17] as the experimental platform (Section 1.3). Front-end throttling and the hardware controller are implemented on the RTL processor generated by FabScalar. The configuration of the experimental processor is shown in Table 3.1.

3.3.2 Overhead of Front-End Throttling

Table 3.2 shows the overhead of adaptive front-end throttling on the 8-way superscalar processor, evaluated by the circuit-level analysis flow. A default switching activity is assumed when estimating

Table 3.2: Overhead of adaptive front-end throttling on the 8-way superscalar processor.

	Area (μm^2)	**Delay** (ns)	**Power** (mW)
Conventional fetch	431267	4.9501	62.2151
Adaptive throttling (Overhead)	444623 (3.1%)	4.9503 ($\approx$ 0%)	63.0909 (1.4%)
Hardware controller (Overhead)	3742 (0.9%)	3.7030 (0%)	0.2720 (0.4%)

the power. The adaptive throttling overhead in the table only represents the overhead of making the front-end instruction delivery flexible and does not include the overhead of the hardware controller, which is evaluated separately. A 64-bit integer ALU already provides enough precision for the arithmetic operations in the hardware controller. Evaluation results show that adaptive front-end throttling has almost no effect on critical-path delay and incurs negligible area and power overhead.

3.3.3 Evaluation Methodology

To evaluate adaptive front-end throttling, the simulation points, generated by the SimPoint tool [67], of six SPEC CPU2000 benchmarks are executed on the RTL processors. Only integer benchmarks are used because floating point instructions are not supported in FabScalar. SimPoint assigns a weight to each simulation point and the weight sum of the performance of all the simulation points is used to represent the performance of the entire benchmark. Each instruction chunk consists of 10000 instructions and each simulation point is sequentially divided into instruction chunks. The selection of instruction chunk size is flexible, and 10000 is chosen in this evaluation because it is relatively fine-grained but not too small to lose the meaningfulness of averaged program behavior. Optimization is made chunk-wise. The average power of executing each chunk of instructions is estimated using the power analysis flow mentioned in Section 1.3. Performance, energy, ED product, and ED^2 product are calculated for each instruction chunk. Energy consumed by each simulation point is the sum of the energy consumed by all instruction chunks in that simulation point. Energy consumed by each benchmark is estimated as the weight sum of the energy of all the simulation points. In software profiling by phase and run-time hardware controller, it is impossible

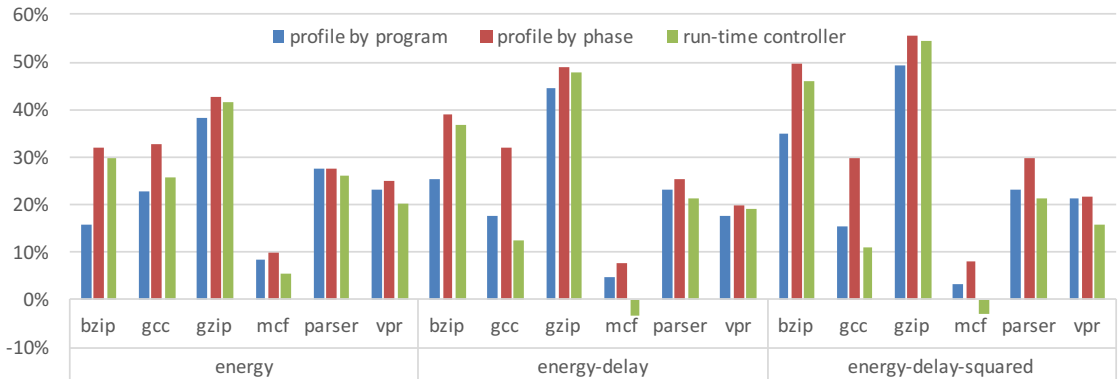


Figure 3.7: Average improvements of adaptive front-end throttling on the 8-way superscalar processor. *Energy*, *ED*, and ED^2 are all normalized to the conventional fixed-width 8-way superscalar processor.

to optimize the *ED* or ED^2 of the entire program at runtime. Instead, optimization is made to the *ED* or ED^2 of each single chunk of instructions, which still optimizes the *ED* or ED^2 of the overall program. The *ED* and ED^2 of the overall program are calculated using the performance and energy of the entire program, the estimation of which is described above.

Adaptive front-end throttling is applied to three superscalar processors with different superscalar widths (number of pipeline "ways"). Table 3.1 shows their configurations.

3.3.4 Evaluating Software Profiling

The results of adaptive front-end throttling using software profiling on the 8-way superscalar processor are shown in Figure 3.7, in which profiling by program and profiling by phase are evaluated against the conventional fixed-width 8-way superscalar processor. The input to the program is fixed. For each of the metrics, the adaptive throttling is optimized for that metric. Adaptive throttling is also applied to the 4-way and 6-way superscalar processors. Figure 3.8 shows the average improvements of energy, *ED*, and ED^2 over all benchmarks on all three processors.

Results show that adaptive throttling using software profiling always does better than conventional fixed-width instruction delivery on all three processors for energy, *ED*, and ED^2 metrics. Compared with profiling by program, profiling by phase further enhances the average improvements by 2-8% on the 8-way processor. Significant further improvements using profiling by phase

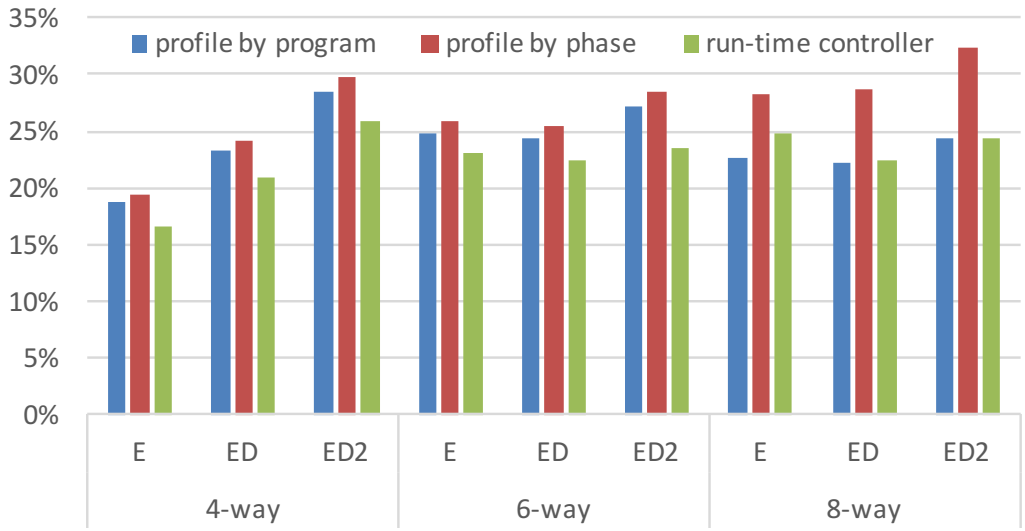


Figure 3.8: Average improvements of adaptive front-end throttling on 4-way, 6-way, and 8-way superscalar processors.

are observed for programs with large behavior variations, such as *bzip* and *gcc*. For processors with smaller superscalar widths, 4-way and 6-way, adaptive throttling also achieves significant average improvements over the aforementioned three metrics, shown in Figure 3.8.

The energy savings mainly come from reducing the number of wrong-path and early-fetched instructions, which waste a significant amount of energy, and from reducing the switching activity of the pipeline by disabling certain instruction delivery paths.

3.3.5 Evaluating Run-Time Controller

The runtime controller is evaluated by applying the runtime control algorithm to the performance and power data obtained from software profiling. Through trial and error, the threshold for detecting sudden performance or power changes and the number of history performance and power data are chosen as 0.125 and 16, respectively. The sampling iteration number n should be kept small to reduce the overhead of sampling but should also be large enough to average out the program fluctuations. The minimum sampling interval should be much larger than the sampling duration to prevent overly frequent sampling but should not be too large in case program behavior changes are missed. The above two parameters are chosen as 12 and 5 million instructions, respectively. The

timer generates a periodic trigger signal every 100 million cycles. Given the above parameters, the duration of a complete sampling process on the 4-way superscalar processor is the time taken to execute $4 \times 12 = 48$ instruction chunks, which is very short compared with the time between two adjacent optimal front-end width changes.

The results of adaptive front-end throttling using run-time controller on the 8-way superscalar processor are shown in Figure 3.7 and the average improvements of energy, ED, and ED^2 over all benchmarks on the 4-way, 6-way, and 8-way superscalar processors are shown in Figure 3.8.

Compared with software profiling by phase with fixed input, which represents the oracle control, the hardware controller performs 10%, 29%, and 26% worse for energy, ED product, and ED^2 product, respectively, on the 8-way superscalar processor, and performs 5%, 9%, and 11% worse, respectively, over all benchmarks on the same processor. The runtime controller causes negative ED and ED^2 results for *mcf*. The reason is that *mcf* has low ILP and the program's performance and power consumption plateau after the front-end width exceeds two, leaving little room for optimization, which is offset by the control overhead of the run-time controller. In this case, the runtime controller can be disabled or better control algorithms can be developed.

3.4 Summary

This work differs from previous work in a number of ways. First, existing techniques often focus on reducing energy consumption while minimizing performance loss, but adaptive front-end throttling can optimize an arbitrary priority metric. Second, this work is not targeted at improving any existing techniques. Adaptive front-end throttling is orthogonal to, and can even leverage, most existing techniques, providing even greater savings. For example, fetch gating based on branch prediction confidence [6,49] and dynamic issue queue, reorder buffer, and load/store queue re-sizing [8,14,26, 54,57] can be applied together with adaptive front-end throttling to achieve greater savings. Third, previous work either does not have a direct way to quantify the overhead of the throttling technique and the resulting energy savings, or gets this information relying on architecture-level modeling frameworks, such as Wattch [13] and McPAT [45], which are known to have limited accuracy. In

this work, the new architecture is implemented at the register transfer level (RTL), and circuit-level synthesis and simulation are used to accurately analyze the area, delay, and power overhead of the throttling technique and resulting energy savings.

Chapter 4

Dynamic Core Scaling

This chapter presents the dynamic core scaling technique. Modern superscalar processors, such as IBM POWER8 [51], can use aggressive datapath resources to extract every bit of performance. These aggressive datapath resources, such as issue queue (IQ), load/store queue (LSQ), and reorder buffer (ROB), account for a significant fraction of energy consumption. DCS trades off performance and energy by scaling these datapath resources. The datapath resources that are dynamically scaled include front-end width, issue width, and sizes of IQ, LSQ, and ROB, all of which significantly influence the energy consumption of superscalar processors. Clock gating and power gating are performed on ususued datapath resources to save energy.

4.1 Scaling Datapath Resources

This section describes the hardware modifications needed to implement DCS. Although scaling datapath resources may reduce the critical path delay of the processor, this work does not consider increasing clock frequency or merging pipeline stages.

4.1.1 Resize Instruction Scheduling Components

DCS powers off portions of IQ, LSQ, and ROB to trade off performance and energy. To resize IQ, LSQ, and ROB, the SRAMs and CAMs in these components are partitioned into a number of independent sub-blocks, each of which is a standalone and usable SRAM/CAM block with its

own peripheral circuitry. A controller selectively assembles the sub-blocks to form SRAMs and CAMs with different sizes. Using smaller independent SRAM/CAM blocks reduces access latency and read/write energy, but increases area because the peripheral circuitry, such as prechargers, sense amplifiers, etc, cannot be shared and are duplicated in each sub-block. Partitioning the SRAM/CAM into more sub-blocks offers finer granularity but causes higher area overhead. To provide fine granularity while avoiding high area overhead, the IQ, LSQ, and ROB are partitioned and assembled as shown in Table 4.1. The full sizes of the IQ, LSQ, and ROB used in the baseline experimental processor are 128 entries, 64 entries, and 256 entries, respectively.

Table 4.1: Available sizes of IQ, LSQ, and ROB.

	Partition sizes	**Sizes that can be assembled**
IQ	16, 16, 32, 64	16, 32, 48, 64, 80, 96, 112, 128
LSQ	8, 8, 16,32	8, 16, 24, 32, 40, 48, 56, 64
ROB	32, 32, 64, 128	32, 64, 96, 128, 160, 192, 224, 256

Instruction dispatch is stalled when resizing IQ, LSQ, and ROB to first drain these components to empty, and then change their sizes. If branch mispredictions or other recovery events that flush the pipeline happen during resizing, IQ, LSQ, and ROB are resized immediately after the flush. Stalling instruction dispatch incurs performance loss. The duration of instruction dispatch stall depends largely on the occupancy of IQ, LSQ, and ROB when resizing happens. Simulations on the experimental processor show that most instruction dispatch stalls are less than 100 cycles, discussed in detail in Section 4.4.6. By carefully controlling the resizing events, the resulting performance loss can be negligible. IQ, LSQ, and ROB can be resized independently and simultaneously. When the processor starts, they are initialized to full sizes, and later are scaled quickly to the right sizes according to application needs.

4.1.2 Throttle Front-End Instruction Delivery

DCS dynamically adjusts the front-end instruction delivery rate to the demands of applications, which offers two benefits. First, the number of wrong-path and early-fetched instructions are reduced and less energy is wasted on processing them. Second, delivering fewer instructions lowers

the occupancy of IQ, LSQ, and ROB, saving energy in these components. Flexible front-end instruction delivery is implemented on the experimental processor using similar techniques in [81], shown in Figure 3.2. The instruction delivery paths, consisting of fetch, decode, rename, and dispatch, are enabled or disabled dynamically by the control registers added to each front-end stage. Corresponding logic in each stage is modified if necessary to accommodate the flexible instruction delivery width. The instruction cache only fetches the required number of instructions dictated by the control register. Pipeline registers corresponding to the disabled instruction delivery paths are clock-gated to save energy. Front-end width changes in a pipeline fashion, starting from the fetch stage and propagating down to the subsequent stages until instructions reach the issue queue and the reorder buffer. Width change takes only one cycle for a single stage, and can jump arbitrarily between any two values ranging from one to the maximum allowable width.

4.1.3 Adjust the Issue Width

DCS reduces the effective issue width when necessary to save energy. The baseline experimental processor has a total of 8 execution units, including 3 simple function units, 1 multiply/divide function unit, 1 branch unit, and 3 load/store address generation units. The number of simple instructions or load/store instructions issued per cycle in the baseline experimental processor can change from 1 to 3. To reduce the issue widths of these two types of instructions, two of the three granted instructions can be hold without being issued by disabling their grant signals using simple control logic. The instructions being hold can participate in the selection process in the next cycle. When the issue width is reduced, the back-end pipeline registers corresponding to the disabled execution lanes are clock gated on a cycle by cycle basis.

4.2 Performance-Energy Tradeoff via DCS

4.2.1 Overview of DCS Control

Controlling DCS for performance-energy trade-off is challenging, because there is no fixed relationship between performance/energy and the microarchitecture of the core. Scaling the core to half

size does not necessarily reduce its performance or energy by half, and may result in very different performance and energy consumption for different applications. However, it is often desirable to have the ability to control performance when doing performance-energy trade-off. To achieve this goal, this chapter proposes a hardware DCS controller that allows an arbitrary target performance to be set and dynamically scales the datapath resources to try to ensure that the core runs at the target performance. The target performance is a certain percentage of the maximum performance the core would achieve if all datapath resources were allocated, and it can be set by the operating systems, etc. At the same time with performance control, the DCS controller minimizes energy consumption by dynamically allocating just the amount of datapath resources needed to meet the target performance. Through simultaneous performance control and energy minimization, DCS achieves a wide range of performance-energy trade-offs.

The performance metric which the DCS controller controls is the instructions per cycle (IPC). To achieve a target IPC that is a certain percentage of the maximum IPC, it is sufficient to achieve the target IPC at every instant of the program execution. This implies that the DCS controller must know the maximum performance the program would achieve if the program was run on a full-size core, and that the DCS controller must be able to effectively control performance at every instant of the program execution. The proposed DCS controller realizes these two goals via *calibration* and *evaluation*, which are two distinct phases during program execution. During the calibration phase, the core is scaled to full size and the IPC of the program is sampled as the maximum IPC for the near future. This is based on the assumption that the IPC of the program will be stable in the near future after the sample. This assumption is true most of the time based on the SPEC benchmark simulations done in this work, which show that programs have distinct phases and the IPC in each phase can be relatively stable for long period of time. In situations where the IPC of the program fluctuates, the effectiveness of using calibration to acquire maximum IPC suffers. However, this is not the common case for most of the SPEC benchmarks. During the evaluation phase, the core's runtime IPC is monitored, and at the end of each evaluation phase, the core is resized if needed to meet the target performance by comparing the runtime IPC with the target IPC.

To monitor performance, the program is divided sequentially into equal-sized instruction

chunks, each of which consists of a fixed number of instructions. Each *calibration* phase and each *evaluation* phase consists of one instruction chunk. Several *evaluation* phases follow one *calibration* phase to reduce the time when the core is in full size. Figure 4.1 shows the control flow. Upon periodic performance calibration triggers, the core is set to full size and the maximum IPC is sampled by executing one instruction chunk. The target IPC is then re-calculated by multiplying the target percentage with the sampled maximum IPC. When the performance calibration finishes, the core enters the evaluation phase and is scaled back to its previous size right before the performance calibration triggers. The runtime performance for executing each instruction chunk is monitored. At the end of each evaluation phase, an evaluation process is triggered, upon which the sampled runtime IPC is compared against the target IPC. The DCS controller then decides how to resize the datapath resources based on the comparison results.

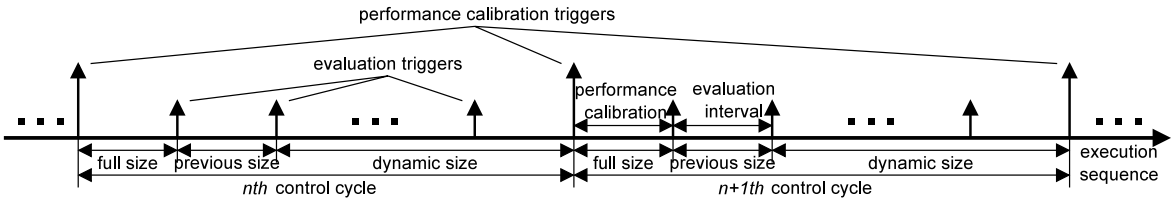


Figure 4.1: Control flow for performance-energy trade-off using DCS.

The rest of the section will provide details on how the DCS controller dynamically scales datapath resources to control performance while minimizing energy consumption at the same time.

4.2.2 Resize IQ, LSQ, and ROB

The controller calibrates the maximum IPC as well as the target IPC in the *calibration* phase, and monitors the runtime IPC of the core in the *evaluation* phase. At the end of each *evaluation* phase, if the runtime IPC is higher than the target IPC, the IQ, LSQ, and ROB are over-allocated and should be downsized to save energy. If the runtime IPC is lower than the target IPC, the IQ, LSQ, and ROB are under-allocated and should be upsized to raise the performance. Although the IQ, LSQ, and ROB can be resized independently and simultaneously, the controller only resizes one of them at a time by one smallest granularity. The reason is to avoid imbalanced resource allocation, such as big IQ and small ROB, that causes low performance but high power consumption. The following

Algorithm 1 Control algorithm for performance-energy trade-off using DCS.

```
1:  procedure RESIZE_CONTROL
2:      if performance < target performance then
3:          if ROB stall was more than IQ and LSQ then
4:              upsize ROB by 32
5:          else if IQ stall was more than ROB and LSQ then
6:              upsize IQ by 16
7:          else if LSQ stall was more than ROB and IQ then
8:              upsize LSQ by 8
9:          end if
10:     else if performance > target performance then
11:         if ROB stall was less than IQ and LSQ then
12:             downsize ROB by 32
13:         else if IQ stall was less than ROB and LSQ then
14:             downsize IQ by 16
15:         else if LSQ stall was less than ROB and IQ then
16:             downsize LSQ by 8
17:         end if
18:     end if
19: end procedure

20: procedure FRONT-END_WIDTH_CONTROL
21:     if IQ_size < 32 and LSQ_size < 16 and
22:       ROB_size < 64 then
23:         front-end_width = issued_instructions/cycle
24:     else if IQ_size < 64 or LSQ_size < 32 or
25:       ROB_size < 128 then
26:         front-end_width = issued_instructions/cycle + 1
27:     else
28:         front-end_width = issued_instructions/cycle + 2
29:     end if
30: end procedure

31: procedure ISSUE_WIDTH_CONTROL
32:     if IQ_size < 32 and LSQ_size < 16 and
33:       ROB_size < 64 then
34:         issue_width = 4
35:     else
36:         issue_width = 8
37:     end if
38: end procedure
```

method determines which component should be resized. The IQ, LSQ, and ROB generate signals that stall the instruction dispatch if those components are full. The controller counts the number of dispatch stalls generated by each component during the execution of each instruction chunk. If upsize is needed, the component that generates the most number of stalls is upsized, because it suffers from under-allocation more severely than the others and is likely the major contributor to performance loss. If downsize is needed, the component that generates the fewest number of stalls is downsized, because it is over-allocated more than the others and downsizing it is likely to cause the least amount of performance loss. The control method is shown by the RESIZE_CONTROL procedure in Algorithm 1.

The instruction chunk size should be large enough to represent the averaged program behavior, but should not be excessively large to lose fine-grained control. The performance calibration period should be large to reduce the time when the core is in full-size for low energy, however, if it is too large the calibrated full-size core's performance may not be accurate. By varying the two parameters in simulations, satisfying results are achieved when the instruction chunk size is 6000 and the performance calibration period is 20 instruction chunks.

4.2.3 Throttle the Front-End Width

The front-end width is kept the same as the number of instructions issued per cycle to deliver only the instructions needed by the issue queue. If no instructions are issued in a cycle, the front-end width is set to 1. If the number of instructions issued per cycle exceeds the maximum front-end width, the front-end width is kept at the maximum. However, in real implementation, this scheme is found to degrade performance more than desired because the instruction window is narrowed. Given that larger IQ, LSQ, and ROB generally favor higher instruction delivery rate, 1 or 2 extra instructions are fetched and delivered according to the sizes of these components, shown by the FRONT-END_WIDTH_CONTROL procedure in Algorithm 1. Evaluations show that this scheme gives better results than simply keeping the fetch width the same as the number of instructions issued per cycle.

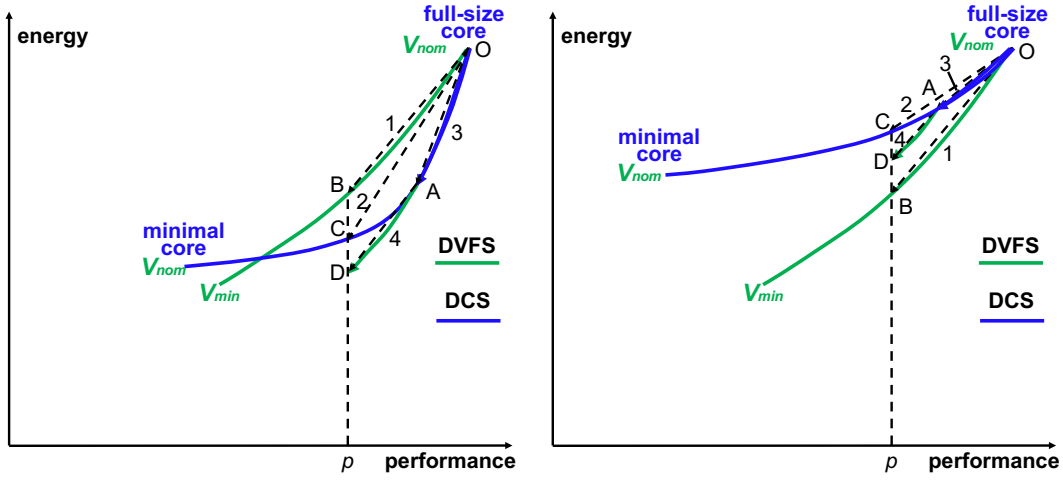
4.2.4 Adjust the Issue Width

The back-end pipeline registers are clock gated on a cycle by cycle basis. Since the total number of instructions to be executed in a program is fixed, the issue width should be kept at the maximum to issue ready instructions as soon as possible to maximize performance. However, recovery events such as branch mispredictions flush the pipeline and the actual number of instructions issued is usually larger than that in the program. Fetch gating based on branch prediction confidence has been proposed in [49] to reduce wrong-path instructions. The same technique can also be applied to instruction issue. To make DCS control simple, this work keeps the issue width at the maximum most of the time, but reduces it to 4 when the core is scaled very small, shown by the ISSUE_WIDTH_CONTROL procedure in Algorithm 1.

4.2.5 Combine DCS with DVFS for Greater Savings

Applying DCS together with DVFS achieves greater energy savings, however, controlling DCS and DVFS together is challenging, which is explained as follows. Assuming that the goal is to maintain a target performance of p while minimizing energy, multiple approaches exist to achieve this performance-energy trade-off, illustrated by Figure 4.2a. The first approach applies DVFS and reaches point B through path 1. The second approach applies DCS and reaches point C through path 2. The third approach combines DCS and DVFS. First, DCS is applied to reach point A through path 3, and then DVFS is applied to reach point D through path 4. The controller has to determine which approach saves energy most, but this is complicated by two problems. First, the resulting energy savings from DCS is unknown prior to scaling. When DCS is more effective than DVFS, shown in Figure 4.2a, the third approach is most effective. When DCS is less effective than DVFS, shown in Figure 4.2b, the first approach becomes the most effective. Unless all three approaches are tried and compared, it is challenging to determine which one is the best. Second, finding out the most effective combination of DCS and DVFS is virtually impossible at runtime, because there exist numerous possible combinations. For example, in the third control approach when scaling from point O to point D, the selection of point A on the DCS curve affects the performance-energy

trade-off results, however, it is impractical to try all the possible points at runtime and then select the most effective one.



(a) DCS is more effective.

(b) DCS is less effective.

Figure 4.2: The challenges of controlling DCS+DVFS.

To address these challenges, the rest of this section proposes three control mechanisms to effectively manage performance-energy trade-off using a combination of DCS and DVFS. Section 4.2.5.1 proposes an oracle controller to demonstrate the optimal control strategy when the resulting performance and energy of DCS is known by oracle. Section 4.2.5.2 and Section 4.2.5.3 propose two practical controllers that are applicable in real implementations when no oracle information of DCS is known.

4.2.5.1 The Oracle Controller

The oracle controller assumes that the resulting performance and energy of DCS is known beforehand, thus it knows the most effective combination of DCS and DVFS to achieve a certain performance-energy trade-off. Before going into the details of the oracle controller, a few notations are introduced first. Performance is denoted by p. Energy is denoted by E, or $E(p)$, a function of performance. The effectiveness of performance-energy trade-off is defined as the ability to reduce energy when a certain performance is traded off, given by $Eff = E'(p) = \frac{dE(p)}{dp}$.

The oracle control mechanism is shown in Figure 4.3, where O represents the point where the

core is at the nominal voltage and in full size, P represents the point where only DVFS is applied and the voltage is scaled to V_{min}, and Q represents the point where only DCS is applied and the core is scaled to minimal size. The effectiveness of performance-energy trade-off at points O, P, and Q for DCS and DVFS are represented as $Eff_{DCS,O}$, $Eff_{DVFS,O}$, $Eff_{DCS,P}$, and $Eff_{DVFS,Q}$. The effectiveness of DCS and DVFS decreases as they are applied from point O to point P or Q. Thus, the following relationships are always true: $Eff_{DCS,Q} < Eff_{DCS,O}$ and $Eff_{DVFS,P} < Eff_{DVFS,O}$. Based on the effectiveness of DCS and DVFS, the oracle control is categorized into four scenarios:

1. $Eff_{DCS,Q} < Eff_{DCS,O} < Eff_{DVFS,P} < Eff_{DVFS,O}$, where DCS is always less effective than DVFS, and

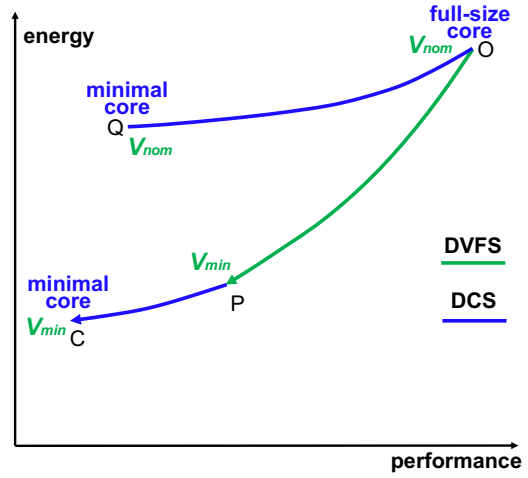
2. $Eff_{DVFS,P} < Eff_{DCS,O} < Eff_{DVFS,O}$, where DCS is partially less effective than DVFS, and

3. $Eff_{DCS,Q} < Eff_{DVFS,O} < Eff_{DCS,O}$, where DCS is partially more effective than DVFS, and

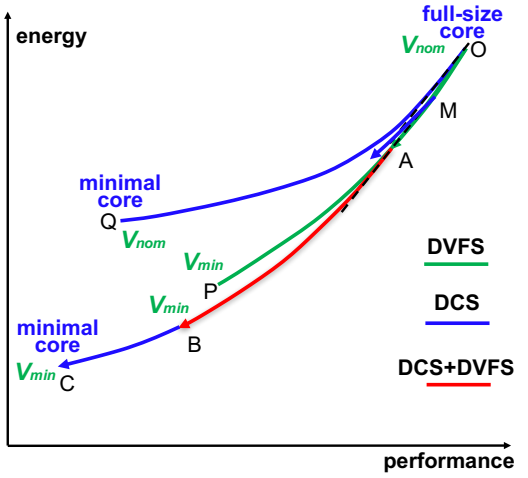
4. $Eff_{DVFS,P} < Eff_{DVFS,O} < Eff_{DCS,Q} < Eff_{DCS,O}$, where DCS is always more effective than DVFS.

In the first scenario where DCS is always less effective than DVFS, shown in Figure 4.3a, there is no need to apply DCS before DVFS is applied until the voltage is scaled to V_{min}, because applying DCS at any middle point of DVFS only leads to inferior performance-energy trade-off. Thus, the oracle controller first applies DVFS from point O to point P where voltage is scaled to V_{min}, then it applies DCS from point P to point C where the core is scaled to minimum. In this scenario, DCS is able to further extend performance-energy trade-off at V_{min} when DVFS stops working.

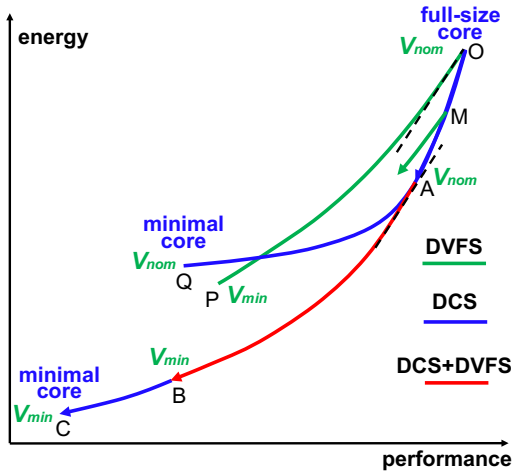
In the second scenario where DCS is partially less effective than DVFS, shown in Figure 4.3b, applying DCS has the potential of improving performance-energy trade-off over DVFS. Since $Eff_{DCS,O} < Eff_{DVFS,O}$, DVFS should be applied first until a certain point A where $Eff_{DCS,O} = Eff_{DVFS,A}$. From point O to point A on the DVFS curve, there is no need to apply DCS, because applying DCS at any middle point M between O and A only leads to inferior performance-energy trade-off since DCS is less effective than DVFS. Once DVFS is applied below point A, the effectiveness of DVFS becomes smaller than $Eff_{DCS,O}$ and applying DCS becomes desired. But as DCS



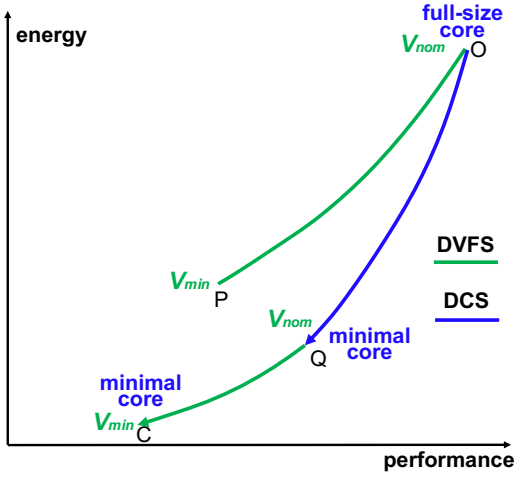
(a) Scenario 1: DCS is always less effective than DVFS.



(b) Scenario 2: DCS is partially less effective than DVFS.



(c) Scenario 3: DCS is partially more effective than DVFS.



(d) Scenario 4: DCS is always more effective than DVFS.

Figure 4.3: The oracle control mechanism when combining DCS with DVFS.

is applied, its effectiveness will decrease and become smaller than that of DVFS, making applying DVFS desired again. Thus, the desire to apply DVFS and DCS alternates from point A until point B where the voltage is scaled to V_{min}, during which the oracle controller should apply the optimal combinations of DVFS and DCS that achieve Pareto-optimal performance-energy trade-offs, represented by the red curve in Figure 4.3b. Once the voltage is scaled to V_{min}, DVFS stops working and DCS is applied until point C where the core is scaled to minimum. In this scenario, the oracle control method is able to extend DVFS slightly before the voltage is scaled to V_{min} and significantly at V_{min}.

In the third scenario where DCS is partially more effective than DVFS, shown in Figure 4.3c, applying DCS improves performance-energy trade-off beyond DVFS. Since $Eff_{DCS,O} > Eff_{DVFS,O}$, DCS should be applied first until a certain point A where $Eff_{DCS,A} = Eff_{DVFS,O}$. From point O to point A on the DCS curve, there is no need to apply DVFS because DVFS is less effective than DCS. Once DCS is applied below point A, the effectiveness of DCS becomes smaller than $Eff_{DVFS,O}$. The situation becomes similar to phase AB in Figure 4.3b. The oracle controller should apply the optimal combinations of DVFS and DCS that achieve Pareto-optimal performance-energy trade-offs from point A to B, represented by the red curve in Figure 4.3c. Once the voltage is scaled to V_{min}, DCS is applied until point C where the core is scaled to minimum. In this scenario, the oracle control method is able to extend DVFS at all voltages.

In the fourth scenario where DCS is always more effective than DVFS, shown in Figure 4.3d, applying DVFS before applying DCS until the minimum core only leads to inferior performance-energy trade-off. Thus, the oracle controller first applies DCS from point O to point Q where the core is scaled to minimum, then it applies DVFS from point Q to point C where the voltage is scaled to V_{min}. In this scenario, DCS is able to significantly extend DVFS in performance-energy trade-off.

The oracle controller demonstrates the optimal strategy for controlling performance-energy trade-off using a combination of DCS and DVFS. However, no such oracle information of DCS exists in practice. The next two sections propose a simple practical controller that performs well when DCS is partially more effective than DVFS, and a sophisticated controller that performs comparably to an oracle controller in all four scenarios.

4.2.5.2 A Simple Practical Controller

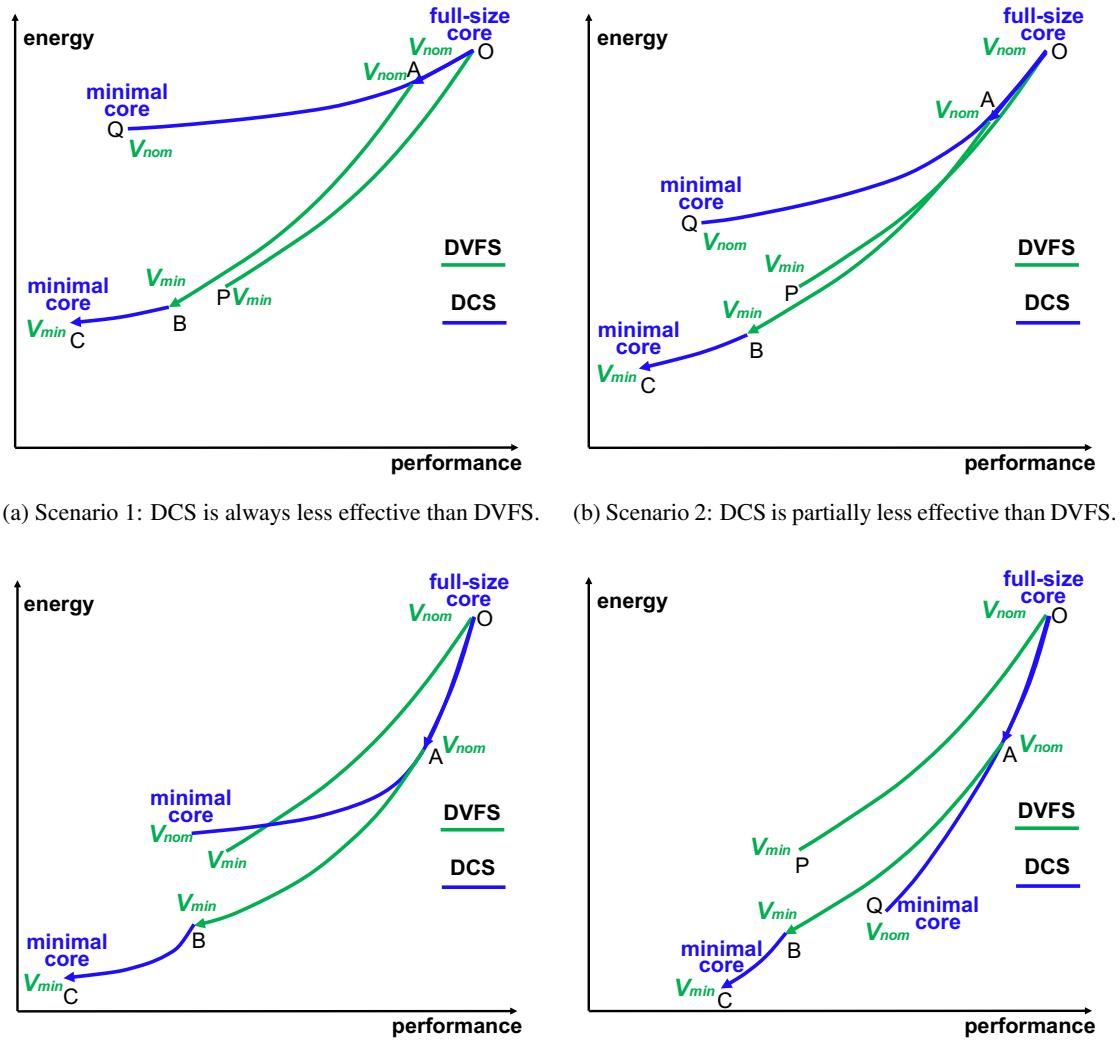
The control mechanism of the simple controller is shown in Figure 4.4. The simple controller makes no effort to determine the effectiveness of DCS at runtime and always applies DCS first until a certain point A on the DCS curve. The selection of point A will be discussed later. Once DCS is applied until point A, the controller then applies DVFS until the voltage is scaled to V_{min} at point B. At this point, DVFS no longer works and the controller re-applies DCS until the core is scaled to minimum at point C. In sum, the simple control method consists of three phases - OA where DCS is applied, AB where DVFS is applied until the voltage is scaled to V_{min}, and BC where DCS is applied again until the core is scaled to minimum. Next the effectiveness of the simple controller is analyzed for the aforementioned four scenarios.

In the first scenario where DCS is always less effective than DVFS, shown in Figure 4.4a, applying DCS at the beginning will lead to inferior performance-energy trade-off. The controller is only able to extend performance-energy trade-off when the voltage is scaled to V_{min}.

In the second scenario where DCS is partially less effective than DVFS, shown in Figure 4.4b, applying DCS at the beginning causes inferior performance-energy trade-off. As DVFS is applied from point A, since $Eff_{DVFS,P} < Eff_{DCS,O}$, curve AB will go under curve DVFS after the effectiveness of DVFS drops below $Eff_{DCS,O}$. The simple controller causes inferior performance-energy trade-off in the early phase of the scaling, but can slightly improve performance-energy trade-off at the later phase of the scaling.

In the third scenario where DCS is partially more effective than DVFS, shown in Figure 4.4c, applying DCS at the beginning improves performance-energy trade-off. If point A is properly chosen such that $Eff_{DCS,A} = Eff_{DVFS,O}$, energy savings can be maximized. Applying DVFS subsequently from point A always saves more energy than DVFS alone. When the voltage is scaled to V_{min}, applying DCS further extends performance-energy trade-off. Compared with the oracle controller, the drawback of the simple controller in this scenario is that DVFS is used from A to B instead of the optimal combinations of DCS and DVFS, leading to less energy savings. However, the effectiveness of the simple controller is already satisfying in this scenario.

In the fourth scenario where DCS is always more effective than DVFS, shown in Figure 4.4d,



(a) Scenario 1: DCS is always less effective than DVFS.

(b) Scenario 2: DCS is partially less effective than DVFS.

(c) Scenario 3: DCS is partially more effective than DVFS.

(d) Scenario 4: DCS is always more effective than DVFS.

Figure 4.4: The simple control mechanism when combining DCS with DVFS.

applying DCS at the beginning improves performance-energy trade-off. However, applying DVFS before the core is scaled to minimum decreases energy savings. Thus, compared with the oracle controller, the simple controller is less effective but still performs better than DVFS alone.

The remaining question is how to select point A to maximize energy savings. In the first two scenarios, energy savings are maximized when point A is located at O. In the third scenario, energy savings are maximized when $Eff_{DCS,A} = Eff_{DVFS,O}$. In the fourth scenario, the energy savings are maximized when point A is located at Q. The optimal point A is different in different scenarios. However, the evaluations in this chapter show that the third scenario is most common for the SPEC benchmarks studied. Thus, the selection of point A is optimized for the third scenario and it is chosen such that $Eff_{DCS,A} = Eff_{DVFS,O}$. But the point A that satisfies this condition varies for different workloads. Since the simple controller has no mechanism to determine the effectiveness of DCS at runtime, a workaround is to use a fixed point A for all the workloads. Evaluations show that placing point A around the target performance of 0.9 is most effective on average. The drawback of this approach is that a fixed point A cannot always be optimal and can lead to inferior performance-energy trade-offs. Even so, the evaluations show that the simple controller is still relatively effective.

4.2.5.3 A More Sophisticated Practical Controller

The lack of mechanisms to determine the effectiveness of DCS at runtime limits the simple controller's ability to make smarter control decisions. The on-chip digital power meter, such as the one in the Intel Sandy Bridge microprocessor [62], has already been built in commercial processors to assist power management. The sophisticated controller assumes that such a digital power meter has already been built on the processor chip and provides power information. Using this power information, the sophisticated controller is able to calculate the effectiveness of DCS at runtime and make smarter control decisions.

In commercial processors, DVFS is often applied using power states [3], which define a couple of voltage and frequency operating points. The processor is only allowed to operate in these power states. Similarly, power states can also be defined for DCS and only allow it to operate on certain

performance points, although DCS is able to operate at any performance. An example of power states for DVFS and DCS is shown in Table 4.2, where $f_1 > f_2 > f_3 > \cdots > f_N$ and $p_{DCS,1} > p_{DCS,2} > p_{DCS,3} > \cdots > p_{DCS,N}$.

Table 4.2: Power states used in the sophisticated controller.

Power states	**DVFS**			**DCS**	
	Frequency	**Voltage**	**Power**	**Performance**	**Power**
1	f_1	V_1	$P_{DVFS,1}$	$p_{DCS,1}$	$P_{DCS,1}$
2	f_2	V_2	$P_{DVFS,2}$	$p_{DCS,2}$	$P_{DCS,2}$
3	f_3	V_3	$P_{DVFS,3}$	$p_{DCS,3}$	$P_{DCS,3}$
⋮	⋮	⋮	⋮	⋮	⋮
N	f_N	V_N	$P_{DVFS,N}$	$p_{DCS,N}$	$P_{DCS,N}$

The sophisticated controller controls performance-energy trade-off by combining DCS and DVFS in different power states. To achieve a certain performance-energy trade-off, the sophisticated controller always tries DCS first for one instruction chunk, defined in Section 4.2.1, no matter which power state the processor is in. Using the on-chip digital power meter, the controller can dynamically calculate the effectiveness of DCS for executing this instruction chunk. The effectiveness of DVFS in different power states can be pre-computed using $P = CV^2 f$ and stored in the controller. Knowing the effectiveness of both DCS and DVFS, the controller can easily decide which one is more effective. If DCS is more effective, it is confirmed that applying DCS is correct and the controller continues to find the next power state. If DVFS is more effective, the controller reverts DCS to the previous power state and applies DVFS instead. The control method is further illustrated in Algorithm 2, where S represents power state, an integer ranging from 1 to N. Depending on whether performance is scaled up or down, the value of the power state is decremented or incremented. The above process is repeated until the desired power target is met.

The effectiveness of the sophisticated controller is analyzed for the aforementioned four scenarios. In the first scenario where DCS is always less effective than DVFS, the controller is able to detect this every time DCS is tried, thus the controller will apply DVFS until the voltage is scaled to V_{min}. Then the controller starts to apply DCS. The sophisticated controller is able to find the optimal combination of DCS and DVFS and performs like the oracle controller, except that

Algorithm 2 Control method for the sophisticated controller.

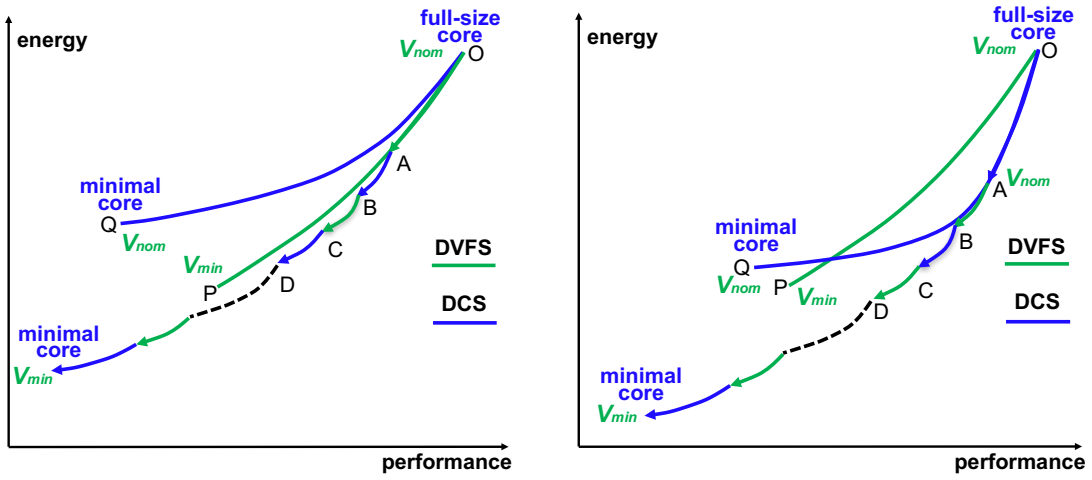
```
1:  procedure CONTROL_POWER_STATE
2:      while power target is not met do
3:          S_DCS,previous = S_DCS,current
4:          S_DCS,current = S_DCS,next
5:          S_DVFS,previous = S_DVFS,current
6:          S_DVFS,current = S_DVFS,next

7:          run one instruction chunk

8:          if Eff_DCS ≥ Eff_DVFS then
9:              if down scale then
10:                 S_DCS,next = S_DCS,current + 1
11:             else if up scale then
12:                 S_DCS,next = S_DCS,current − 1
13:             end if
14:             S_DVFS,next = S_DVFS,current
15:         else if Eff_DCS ≤ Eff_DVFS then
16:             S_DCS,next = S_DCS,previous
17:             if down scale then
18:                 S_DVFS,next = S_DVFS,current + 1
19:             else if up scale then
20:                 S_DVFS,next = S_DVFS,current − 1
21:             end if
22:         end if
23:     end while
24: end procedure
```

trying DCS incurs a little overhead. The performance-energy trade-off curve is similar to Figure 4.3a. The same is true for the fourth scenario where DCS is always more effective than DVFS. The performance-energy trade-off curve is similar to Figure 4.3d. In both scenarios, the sophisticated controller performs almost identical to the oracle controller.

In the second scenario where DCS is partially less effective than DVFS, the controller is able to detect that and apply DVFS at the beginning until it discovers that DCS becomes more effective. Then, the sophisticated controller applies DCS and DVFS alternatively, shown in Figure 4.5a. Although this is less effective than the oracle controller which could always find the optimal combinations of DVFS and DCS, the sophisticated controller is able to approximate the oracle controller and improves over the simple controller. The same is true for the third scenario where DCS is partially more effective than DVFS. The performance-energy trade-off curve is shown in Figure 4.5b. In both scenarios, the sophisticated controller is more effective than the simple controller and approximates the oracle controller.



(a) Scenario 2: DCS is partially less effective than DVFS.

(b) Scenario 3: DCS is partially more effective than DVFS.

Figure 4.5: The sophisticated control mechanism when combining DCS with DVFS.

In sum, having the ability to determine the effectiveness of DCS at runtime makes the sophisticated controller more powerful at making wiser control decisions and able to perform well in all four scenarios.

4.3 Evaluation Methodology

4.3.1 Experimental Platform

This work utilizes FabScalar [17] as the experimental platform (Section 1.3) to accurately evaluate the area, delay, and power of the DCS architecture. Table 4.3 shows the parameters of the experimental processor. DCS and the hardware controller are implemented on the RTL processor generated by FabScalar. The simulations use SPEC CPU2000 benchmarks. A single simulation point is generated using SimPoint [67] for each benchmark. Only integer benchmarks are studied because the FabScalar infrastructure does not support floating point instructions.

Table 4.3: Parameters of the baseline experimental processor.

Fetch/decode/rename/dispatch width	6
Issue/RR/execute/WB width	8
Function units (simple, multiply/divide, branch, ld/st)	3,1,1,3
Issue queue	128
Load/Store queue	64
Reorder buffer (ROB)	256
Branch predictor, BTB	gshare, 8-bit GHR, 64K PHT, 4K BTB
L1 I-Cache	32K, 64-byte block, 4-way, 1 cycle
L1 D-Cache	64K, 64-byte block, 4-way, 1 cycle
Unified L2	2M, 64-byte block, 8-way, 18 cycles

4.3.2 Methodologies for Getting DCS Results

Figure 4.6 shows the average results of DCS over all benchmarks. The detailed results for each benchmark are shown in Figure 4.7 and Figure 4.8. For DVFS, the voltage-delay relationship of the experimental processor is characterized using FO4 inverters, which are simulated under different voltages, from 0.7V until the nominal 1.1V. For simplicity, V_{min} is assumed to be 0.7V, although in reality it should be found out experimentally. Energy is estimated using the formula: $E \propto \alpha CV^2 f$.

The power states are selected based on performance, normalized to the nominal voltage for DVFS and to the full-size core for DCS. The normalized performance in two adjacent power states have a difference of 0.5. For DVFS, eight power states are defined with normalized performance

from 1 to 0.65. For DCS, ten power states are defined with normalized performance from 1 to 0.5.

The DCS-potential and DCS+DVFS-potential curves show the potential of DCS in performance-energy trade-off. These curves are obtained through an extensive design space exploration. A large number of carefully selected cores with varying DCS parameters are simulated (under different voltages for DCS+DVFS-potential) and plotted on the performance-energy diagram. No control techniques are applied at this moment and the datapath resources of the cores are fixed. The cores located on the performance-energy Pareto frontier are selected and plotted to form the potential curves. These cores have diverse DCS parameters, ranging from the smallest core to the full-size core. The DCS-controller curve shows the performance-energy trade-off using the DCS hardware controller. The DCS+DVFS-simple and the DCS+DVFS-sophisticated curves show the performance-energy trade-off using the simple controller (Section 4.2.5.2) and the sophisticated controller (Section 4.2.5.3) when combining DCS and DVFS. DCS+DVFS-potential can be viewed as an approximation of the oracle controller (Section 4.2.5.1).

4.4 Results

This section first shows the effectiveness of DCS at controlling performance while minimizing energy consumption. Then, it presents the results of using DCS to extend performance-energy tradeoff. Finally, some further discussions about DCS are made.

4.4.1 The Effectiveness of DCS Performance Control

DCS tries to ensure that the core runs at the target performance and minimizes energy consumption at the same time. Table 4.4 shows the average performance and energy when applying DCS to the experimental processor under different target performances. The DCS hardware controller is most effective for target performances ranging from 0.7 to 1, during which energy drops quickly as performance is lowered. Below 0.7, energy plateaus and decreasing performance leads to limited energy reduction. The accuracy of performance control drops when the target performance is below 0.7, because the lowest performance achievable by the smallest core is around 0.7 for some

benchmarks, which prevents the average performance from being lowered. For target performances between 0.7 and 1, the small inaccuracy in performance control is partly because performance sampling in the *calibration* phase cannot accurately reflect the full-size core's performance, and partly because setting the core at full size in the *calibration* phase increases performance.

Table 4.4: Average performance and energy over all benchmarks for different target performances using the hardware controller. The data are normalized to the full-size core.

target performance	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
average performance	0.56	0.59	0.61	0.66	0.71	0.81	0.88
average energy	0.49	0.49	0.49	0.51	0.53	0.61	0.69

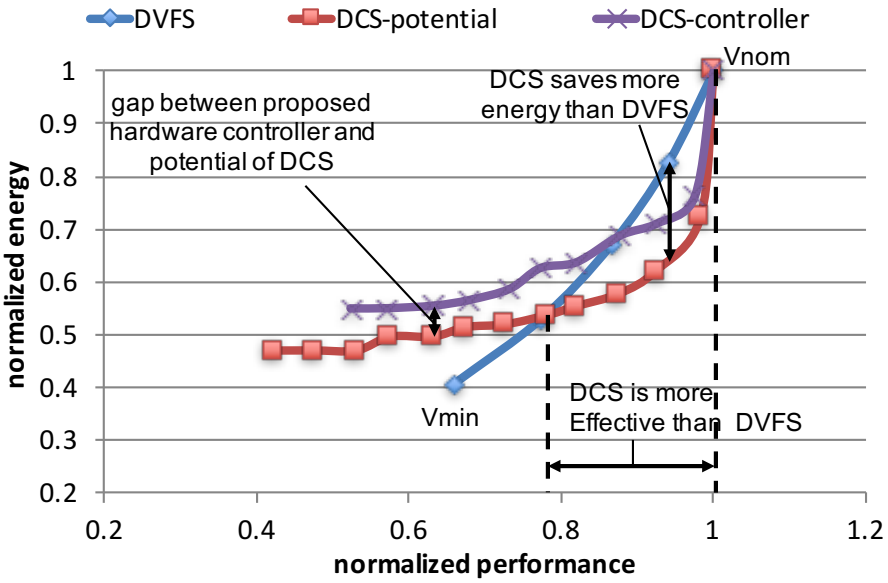
4.4.2 Balancing Datapath Resource Allocation

Since scaling the core may result in faster performance loss than power reduction, could DCS decrease performance but raise energy consumption at the same time? The evaluation results in Table 4.4 show that energy only decreases monotonically first and then plateaus with very small variations as the target performance decreases, and no significant energy increase is observed for all the benchmarks studied. The reason behind this phenomenon is that the DCS controller resizes datapath resources very carefully. Only one of the IQ, LSQ, and ROB that causes the most/least instruction dispatch stalls is allowed to change size by one smallest granularity at a time. This effectively avoids the imbalanced allocation of datapath resources, such as big IQ and small ROB, that causes low performance but high power consumption.

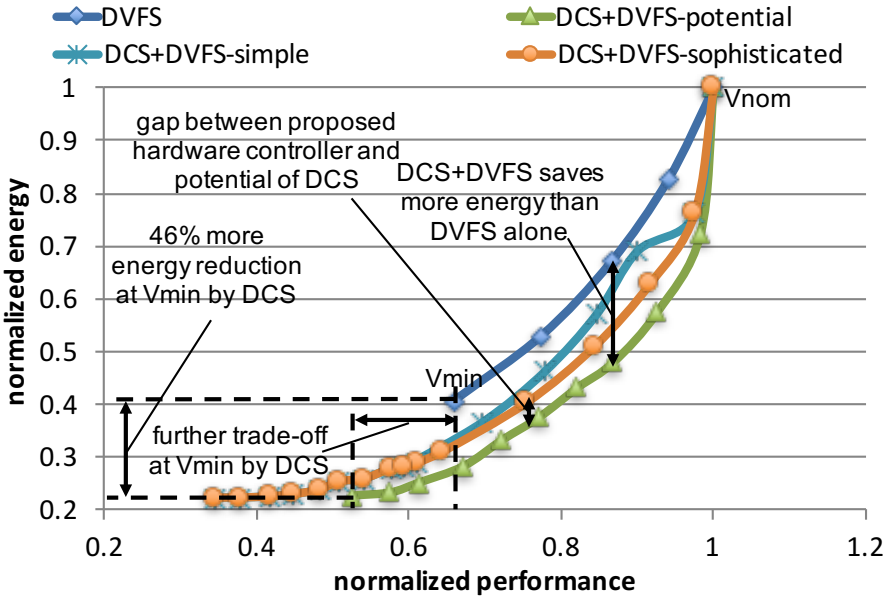
4.4.3 Performance-Energy Trade-off Using DCS

Figure 4.6 gives an overview of DCS results. Beware that DCS-potential and DCS+DVFS-potential do not represent the maximum ability of DCS in trading off performance and energy, instead, they show that DCS has at least the potential shown in the figures. A number of observations are as follows.

First, the DCS-potential curve lies below the DVFS curve at the high performance end, meaning that DCS saves more energy than DVFS at the same performance and thus is more effective than DVFS. The reason behind this is that using aggressive superscalar datapath resources to extract the



(a) DCS



(b) DCS+DVFS

Figure 4.6: Average results of DCS over all benchmarks. Performance and energy are normalized to the full-size core at V_{nom}.

last bit of performance incur high energy costs. The DCS-potential curve then crosses the DVFS curve and goes above it, where it becomes less effective than DVFS. On average, energy reduction using DCS slows down when normalized performance drops below 70%.

Second, applying DCS on top of DVFS significantly extends the performance-energy Pareto frontier of DVFS. At the same performance, DCS+DVFS-potential saves more than DVFS by around 20% of a full-size core's energy at nominal voltage on average.

Third, the effectiveness of DVFS stops when the supply voltage is scaled to V_{min}. However, DCS-potential further reduces 46% energy on average at V_{min} by trading off performance.

Fourth, the DCS hardware controllers are effective in controlling DCS for performance-energy trade-off. The controller curves display very similar characteristics as the potential curves and they are close to each other. Like DCS-potential, DCS-controller is also more effective than DVFS at the high performance end. At V_{min} when DVFS becomes ineffective, DCS-controller also further reduces energy by an average of 46%. When DCS and DVFS are combined, both the simple controller and the sophisticated controller effectively extend DVFS in performance-energy trade-off. The simple controller saves energy more than DVFS by 24-36% within the viable voltage range and by 28-57% at V_{min}, across all benchmarks studied. The sophisticated controller is able to find better combinations of DCS and DVFS and performs comparably to DCS+DVFS-potential.

Figure 4.7 and Figure 4.8 show the detailed results of DCS and DCS+DVFS for each benchmark. Analyzing the results of each individual benchmark is similar to analyzing the average results, and the aforementioned observations can also be found in each benchmark. By scaling voltage from 1.1V to 0.7V, DVFS reduces 60% energy by trading off 34% performance. Compared with DVFS, DCS-potential reduces 37-66% energy by trading off 29-59% performance, and DCS+DVFS-potential reduces 68-84% energy by trading off 34-48% performance, across all benchmarks. For *gap* and *parser*, DCS-potential always saves more energy than DVFS. *mcf* exhibits a rapid drop of energy without performance loss. It is the most memory intensive SPEC CPU2000 integer benchmark and this limits its ILP. A core with 16-entry IQ, 32-entry LSQ, and 64-entry ROB already achieves the same performance as the full-size core. Any core larger than that only causes energy waste for *mcf*.

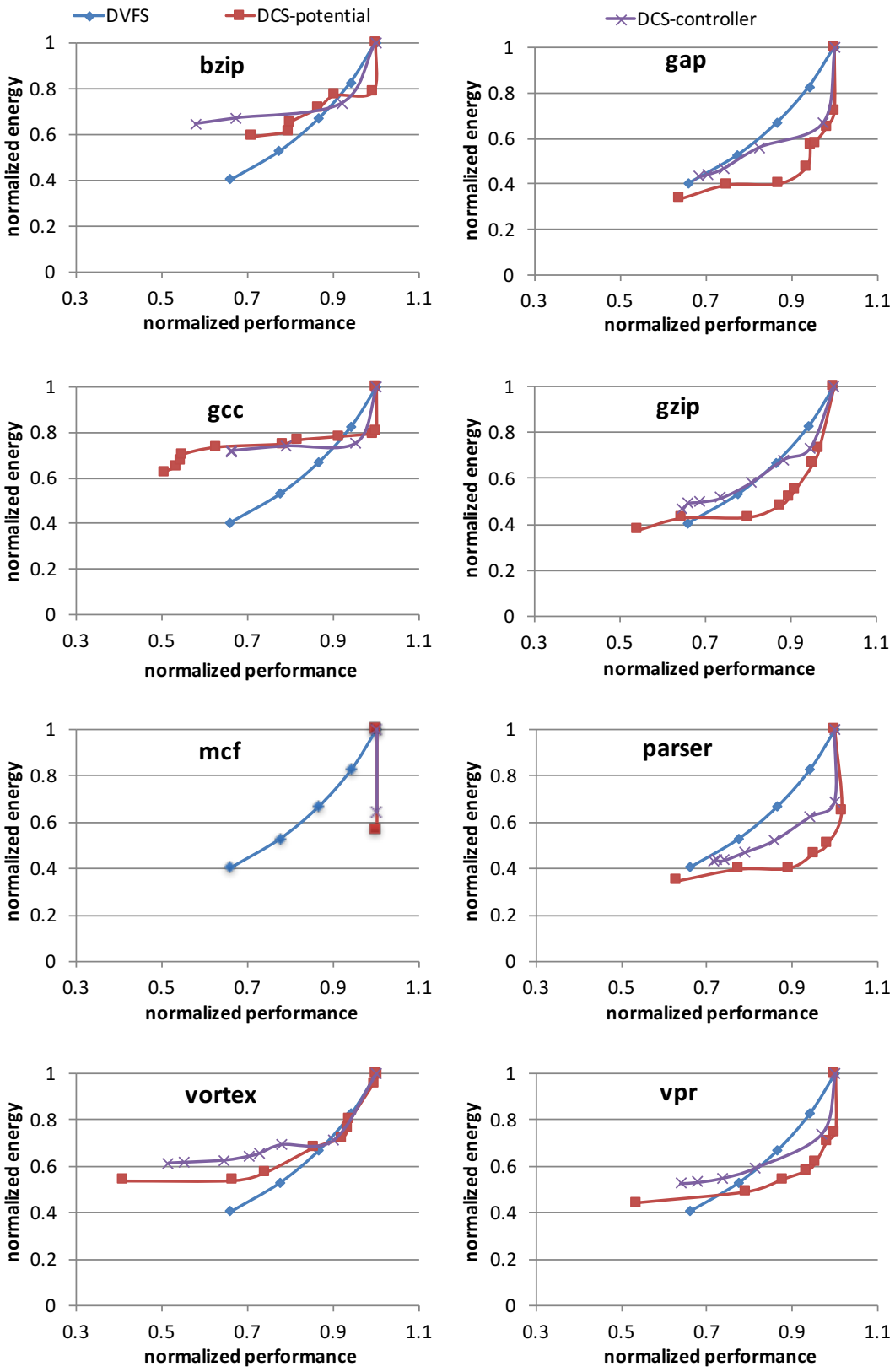


Figure 4.7: Results of DCS for selected benchmarks.

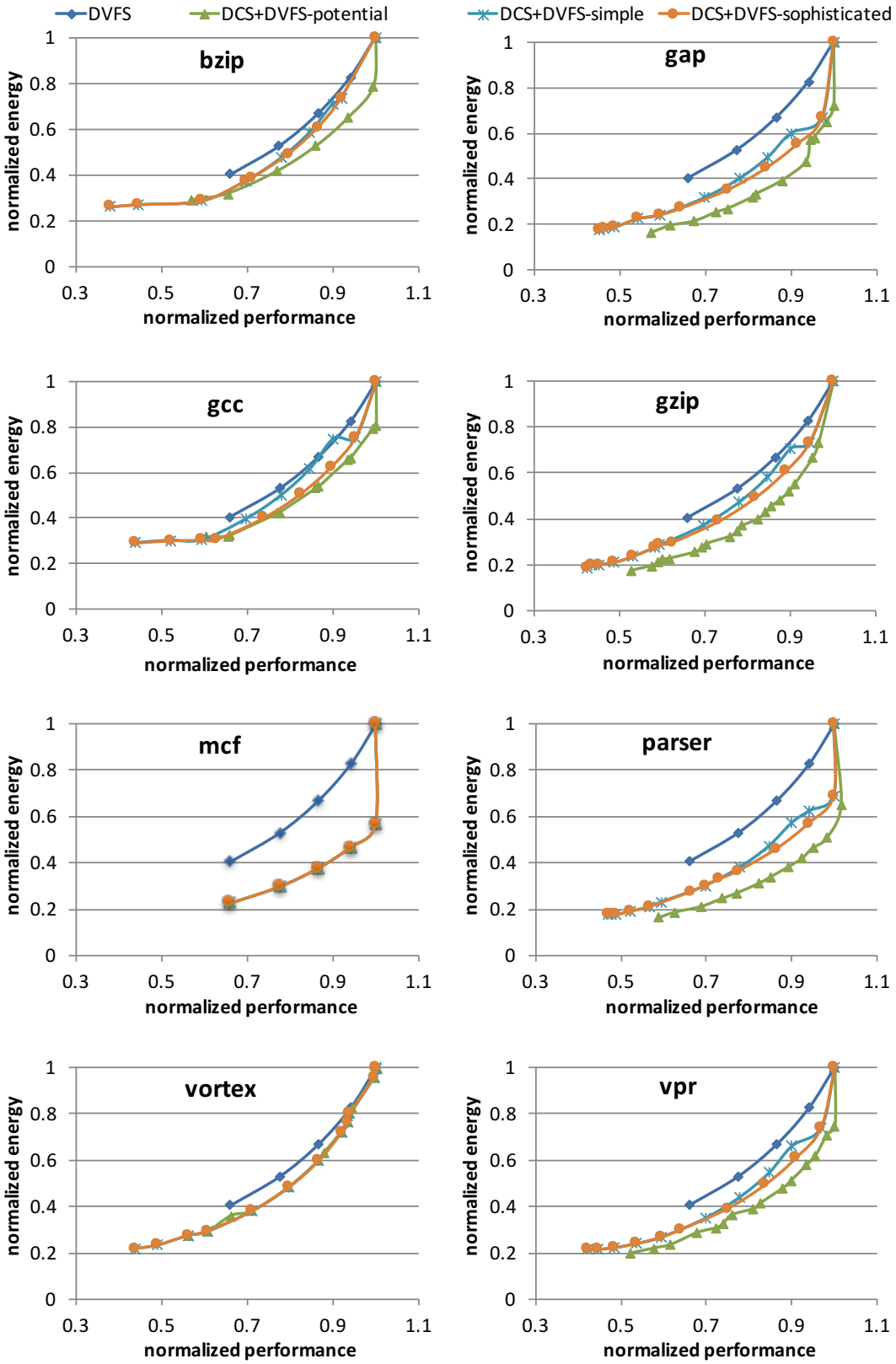


Figure 4.8: Results of combining DCS and DVFS for selected benchmarks.

The DCS hardware controller achieves less energy savings than DCS-potential. The amount of diminishing energy savings is because the core is operated in full size during performance calibration, which increases the core's energy consumption. Despite the imperfectness of the hardware controller, it still achieves 28-57% energy reduction with 0-48% performance trade-off across all the benchmarks. We believe that better DCS controller can still be developed to exploit more potential of DCS, which is worth of future study.

4.4.4 Combining DCS with DVFS for Greater Savings

The major drawback of the simple controller is that it always starts to apply DVFS at a fixed performance point for all benchmarks, which is often sub-optimal and leads to inferior performance-energy trade-off. Since the sophisticated controller knows the effectiveness of DCS at runtime, it is able to find better combinations of DCS and DVFS and make more effective performance-energy trade-offs. Figure 4.6b shows that the sophisticated controller is more effective than the simple controller around the normalized performance of 0.9, which is the point where the simple controller starts to apply DVFS.

The sophisticated controller is able to perform comparably to the oracle controller. In Figure 4.6b, the gap between the DCS+DVFS-sophisticated curve and the DCS+DVFS-potential curve is due to the limited effectiveness of the hardware DCS controller. Since the hardware DCS controller is unable to fully exploit the potential of DCS, the sophisticated controller is unable to fully exploit the potential DCS+DVFS either. This chapter also evaluated the effectiveness of the sophisticated controller by using the oracle DCS controller. The gap between the DCS+DVFS-sophisticated curve and the DCS+DVFS-potential curve then becomes very small and the sophisticated controller is almost as effective as the oracle controller.

4.4.5 Overhead of DCS

Table 4.5 shows the overhead of DCS compared with the baseline experimental processor without DCS. The core with DCS in the table only includes the overhead of making the core scalable and does not include the overhead of the hardware controller, which is evaluated separately. DCS and

the hardware controller together incur less than 5% area overhead and about 3% power overhead, and almost do not affect critical-path delay.

Table 4.5: Percentage overhead of DCS compared with the conventional experimental processor without DCS.

	Area	**Delay**	**Power**
Core with DCS	2.16%	0.96%	1.13%
DCS controller	2.95%	0%	2.33%

4.4.6 Resizing Penalty

When resizing IQ, LSQ, and ROB, instruction dispatch is stalled until these components are drained to empty, which incurs negligible performance penalty. Table 4.6 shows the average number of cycles when instruction dispatch is stalled in each IQ, LSQ, and ROB resize event for executing one million instructions, during which 21-116 resizing events happened across all benchmarks. The worst-case total instruction dispatch stall is within 6000 cycles, which is very short compared with the number of cycles taken to execute one million instructions. In addition, stalling instruction dispatch does not stall the entire pipeline and there are still instructions being issued and executed during that time. Based on the above analysis, the performance loss due to resizing IQ, LSQ, and ROB is less than 1% in the worst case.

Table 4.6: Average number of cycles taken for resizing IQ, LSQ, and ROB when the DCS target performance is 0.9.

	bzip	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**vortex**	**vpr**
IQ	16	13	45	20	4	27	18	22
LSQ	20	11	41	21	26	25	18	19
ROB	27	12	48	22	9	16	21	20

4.4.7 Resource Utilization

The reduced occupancy of IQ, LSQ, and ROB reduce both dynamic and static energy in these components, shown in Figure 4.9. Resource utilization under different target performances is shown in Figure 4.10 and Figure 4.11. As the target performance decreases, the occupancy of IQ, LSQ, and ROB and the front-end width decrease. The energy savings mainly come from reducing the

effective sizes of the instruction scheduling components and the widths of the front-end and back-end. Reducing the effective sizes of the instruction scheduling components reduces both dynamic and static energy. Reducing the front-end and back-end widths reduces the switching activity and thus the dynamic energy. The resource utilization drops most quickly at the high performance end, which is in accordance with the fact that DCS energy reduction is most effective in this performance range.

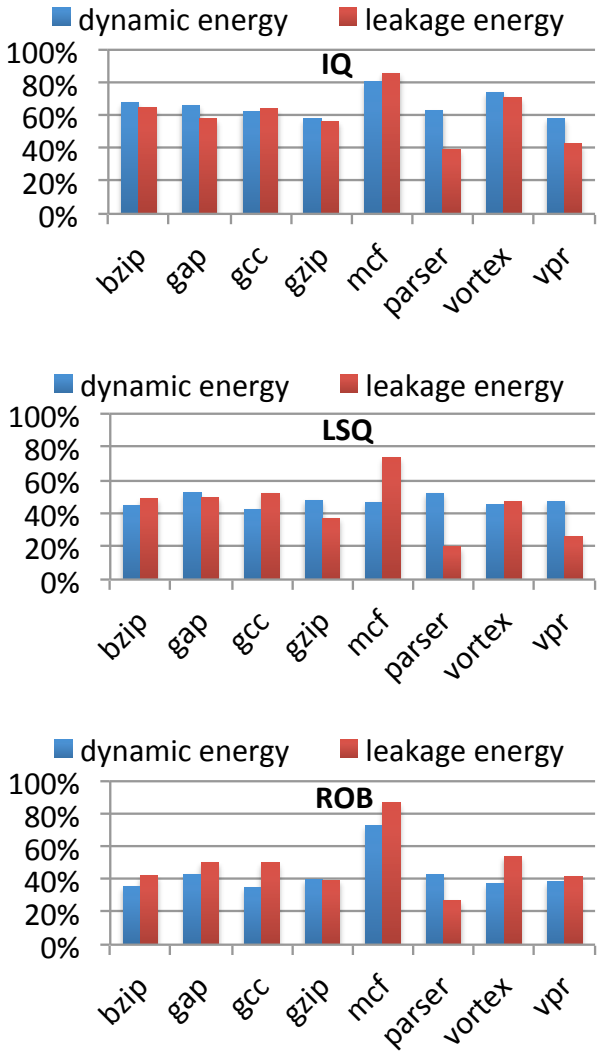


Figure 4.9: Dynamic and leakage energy reduction in IQ, LSQ, and ROB with target performance of 0.9 at nominal voltage.

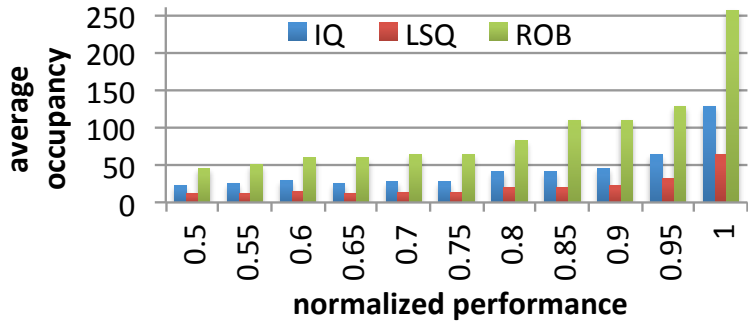


Figure 4.10: Average sizes of IQ, LSQ, and ROB with different target performances over all benchmarks.

4.4.8 Effectiveness of DCS on Smaller Processors

The baseline experimental processor is relatively large. The purpose of selecting a large processor is to adapt to diverse performance and energy demand, which changes greatly as applications and user scenarios change. Some user scenarios require high performance and can tolerate high energy consumption, while others require low energy consumption and can tolerate performance loss. A single fixed design can never satisfy both demand. A large fixed processor can deliver high performance, but is inferior when energy becomes the top concern. A small fixed processor has better energy efficiency, but is inferior when performance becomes the top concern. In contrast, a large processor with DCS implemented has the flexibility to dynamically adapt to both high performance and low energy demand with only a single design. However, applying DCS to smaller processors diminishes this benefit, because smaller processors inherently cannot deliver high performance.

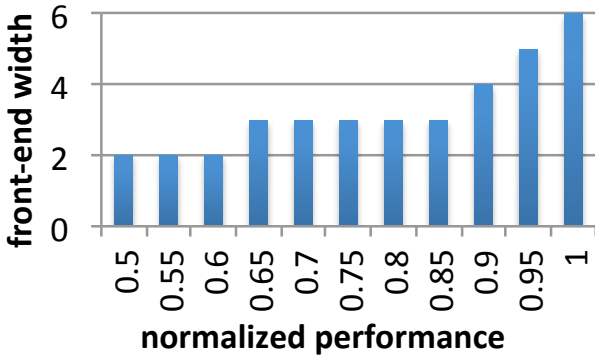


Figure 4.11: Average front-end widths with different target performances over all benchmarks.

The effectiveness of DCS on smaller processors is also studied in this evaluation. DCS is

Table 4.7: Applying DCS to a smaller processor with IQ, LSQ, and ROB at half sizes of those in the baseline experimental processor. Performance and energy are normalized to the full-size processor.

	performance tradeoff range	**energy tradeoff range**	**performance range where DCS is more effective than DVFS**
bzip	0.71 - 1	0.75 - 1	none
gap	0.65 - 1	0.43 - 1	0.76 - 1
gcc	0.61 - 1	0.76 - 1	none
gzip	0.56 - 1	0.48 - 1	0.78 - 1
mcf	1	0.61 - 1	0 - 1
parser	0.62 - 1	0.46 - 1	0.77 - 1
vortex	0.44 - 1	0.67 - 1	0.97 - 1
vpr	0.54 - 1	0.55 - 1	0.85 - 1

applied to another superscalar processor with the same parameters as the baseline experimental processor shown in Table 5.2, except that the sizes of its IQ, LSQ, and ROB are reduced by half. The potential of DCS in performance-energy trade-off using the extensive design space exploration method is shown in Table 4.7. The results show the range of performance-energy trade-off by DCS, and the range of performance where DCS is more effective than DVFS. It is observed that after reducing the sizes of IQ, LSQ, and ROB by half, DCS is still significantly more effective than DVFS in trading off performance and energy for five out of eight benchmarks. For *gap*, *gzip*, *mcf*, *parser*, and *vpr*, DCS-potential saves more than DVFS by around 10-20% of a full-size cores energy at nominal voltage on average. For *bzip*, *gcc*, and *vortex*, the advantage of DCS over DVFS diminishes to none or a few percent. *mcf* exhibits a rapid drop of energy without performance loss when DCS is applied, thus its performance trade-off range is 1 rather than a range, and DCS is thought as more effective than DVFS at all performances for *mcf*.

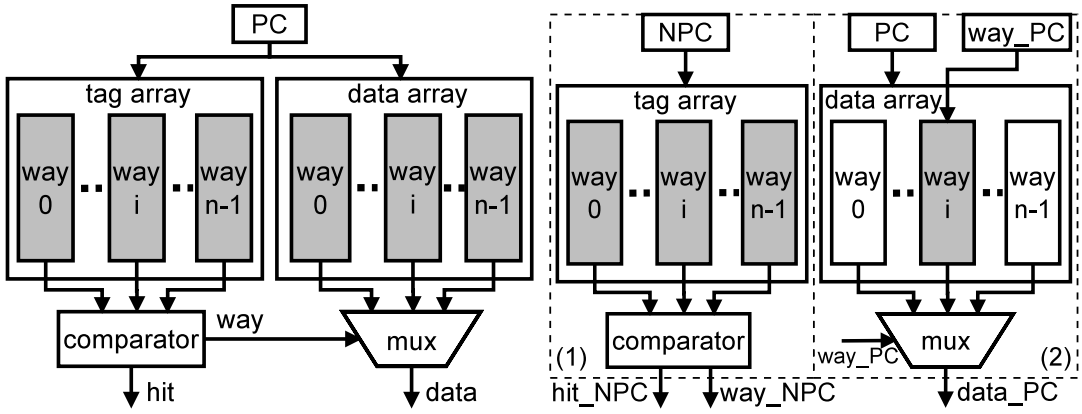
Although shrinking the processor will diminish the gain of DCS, DCS is still very effective on smaller processors. The advantage of DCS is most pronounced when the processor is large, which provides the largest flexibility in adapting between high performance and low energy consumption.

Chapter 5

Early Tag Lookup

5.1 Overview

The traditional L1 instruction caches read out the data in all ways in parallel with tag lookup in order to reduce the access latency of set-associative level-one (L1) caches, shown in Figure 5.1a. However, only the data in the matching way is used and the others are discarded, resulting in significant energy waste.



(a) Conventional cache (b) Early tag lookup

Figure 5.1: Comparison of conventional cache access and early tag lookup. Shaded blocks are accessed.

To reduce this energy waste, ETL determines the matching way *one cycle earlier* than the actual cache access, eliminating non-matching way accesses without sacrificing performance, shown in Figure 5.1b. Unlike conventional instruction caches, ETL keeps two instruction fetch addresses.

One is the current fetch address, stored in the program counter (PC), and the other is the next fetch address, stored in the next program counter (NPC). In cycle *i* when instructions at PC are fetched, the matching way has already been determined in the previous cycle (cycle *i-1*), when the current PC was the next fetch address stored in NPC. The matching way was determined by looking up the tag array using NPC, shown in Figure 5.1b. In the case when the matching way lookup in cycle *i-1* failed, ETL fetches instructions at PC by accessing all the data ways in parallel, like in conventional caches. Thus, ETL does not incur any performance penalty.

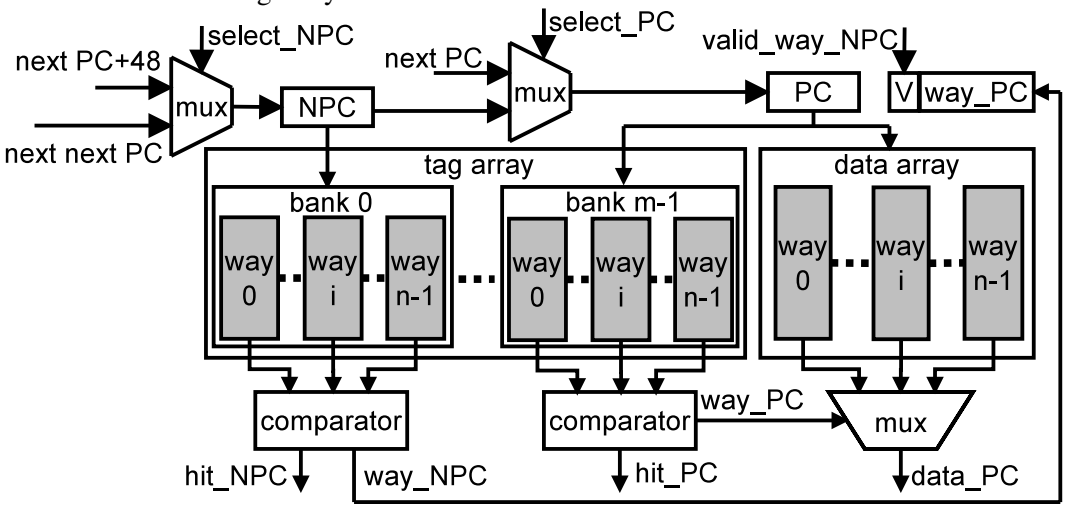
5.2 Early Tag Lookup

5.2.1 The Basics of ETL

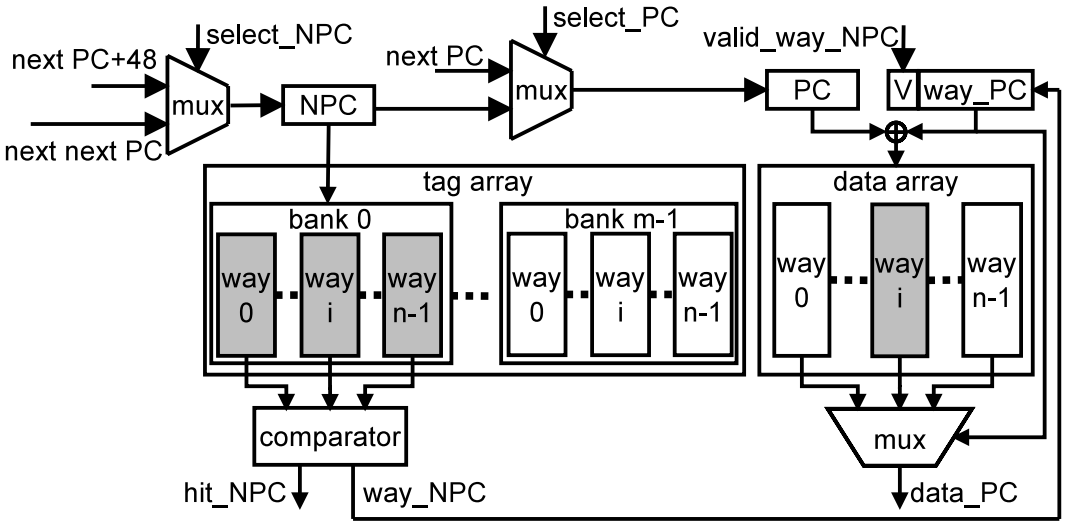
ETL accesses the tag array and determines the matching way earlier than the actual cache access. In each cycle, fetching instructions at the current fetch address and determining the matching way for the next fetch address are performed in parallel. To do this, ETL needs both the current fetch address, PC, and the next fetch address, NPC, at the same time. For now, we assume that NPC can be obtained in some way and focus on the operations of the L1 instruction cache. The details of acquiring NPC will be discussed in Section 5.2.2.

When program starts or exceptions happen, PC is loaded with a new address and its matching way is unknown. The operations of the cache in this situation are shown in Figure 5.2a. PC accesses the tag array and the data array in parallel, same as the conventional cache access shown in Figure 5.1a. Thus, the non-matching data ways are also accessed and no energy is saved. Simultaneously, NPC also accesses the tag array to determine the matching way for the next fetch address. If NPC is the correct next fetch address and hits the tag array, NPC is loaded into PC, its matching way way_NPC is loaded into way_PC, and way_PC is set to valid at the beginning of the next cycle. If NPC is the correct next fetch address but misses the tag array, NPC is still loaded into PC but way_PC is set to invalid. A cache miss service is started. If NPC is not the correct next fetch address, the correct next PC is loaded into PC and way_PC is set to invalid. PC and NPC need to look up the tag array simultaneously when way_PC is unknown. To support two simultaneous

accesses, the tag array is multi-banked with interleaved addresses. As it turns out later, there won't be bank conflicts in the tag array.



(a) Matching way for PC is unknown.



(b) Matching way for PC is known.

Figure 5.2: Cache access in ETL. Assume six 64-bit instructions are fetched per cycle. Shaded blocks are accessed.

When the matching way for PC is known, the operations of the cache are shown in Figure 5.2b. PC accesses the matching data way directly without touching the non-matching ways, thus saving energy. PC does not need to access the tag array now, and only NPC accesses the tag array to determine its matching way in advance. At the beginning of the next cycle, PC and way_PC are updated in the same way as when way_PC is unknown. The remaining question is how to acquire

NPC at the same time with PC, which is explained next.

5.2.2 Acquiring NPC

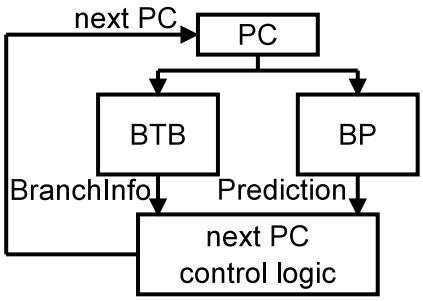
5.2.2.1 Multiple Branch Prediction

Branch target buffer (BTB) and branch predictor (BP) are commonly used in modern processors to predict the next fetch address without decoding the instructions. One approach to acquire both PC and NPC simultaneously is to extend BTB and BP to predict two fetch addresses in one cycle. Predicting multiple branches in one cycle has been studied in [79]. However, this approach significantly increases the size of the BTB by storing the secondary branch target information, and requires multiple ports in BP to read counter bits of secondary branches, which is very expensive in area, power, and timing. Due to the significant overhead, this approach will not be discussed further.

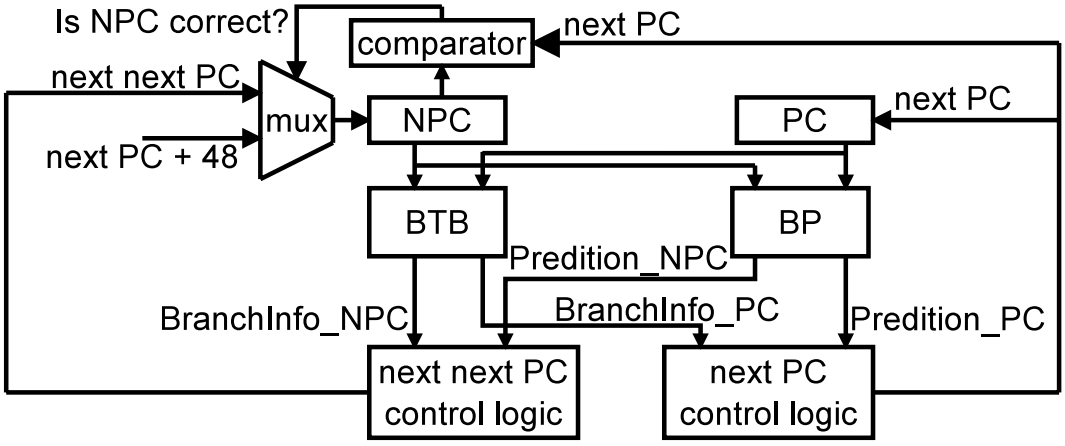
5.2.2.2 A Simple Way to Predict NPC

To avoid expensive hardware overhead, an alternative approach is to obtain NPC through prediction. This approach incurs insignificant hardware overhead, but successfully obtains the correct NPC most of the time. The conventional processor acquires the next PC using information from BTB and BP, shown in Figure 5.3a. An observation is that the next PC can be determined as long as PC is known. If PC is substituted with NPC in Figure 5.3a, then the next next PC can be determined as well. Suppose that both PC and the correct NPC are known at the beginning, then the next next PC can be obtained by accessing BTB and BP using NPC. In the next cycle, the next next PC is loaded into NPC and the address in NPC is loaded into PC, thus both PC and NPC are known simultaneously. The remaining question is how to acquire the correct NPC at the very beginning?

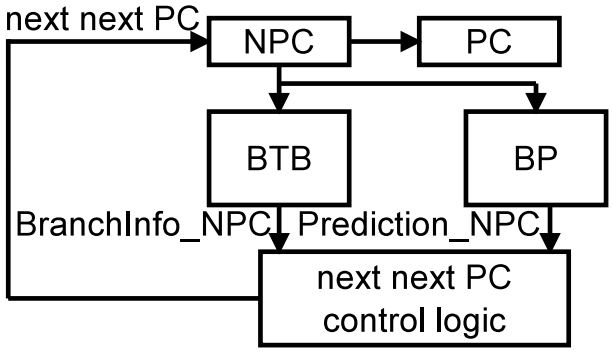
The solution is to predict that NPC is the fall through of PC and later verify that the prediction is correct. When NPC is unknown, the processor enters the prediction mode, meaning that NPC is being predicted and may not be correct, shown in Figure 5.3b. NPC accesses BTB and BP and obtains the next next PC. To verify that NPC is predicted correct, PC also accesses BTB and BP to acquire the correct next PC. If NPC equals the correct next PC, NPC is predicted correct and the



(a) Conventional.



(b) Prediction mode is true.



(c) Prediction mode is false.

Figure 5.3: NPC prediction. Assume the processor fetches six 64-bit instructions per cycle.

processor exits the prediction mode. In the next cycle, the next next PC is loaded into NPC and NPC is loaded into PC, thus both PC and NPC are acquired. If NPC does not equal the correct next PC, NPC is predicted wrong and the processor stays in the prediction mode. In the next cycle, PC is loaded with the correct next PC and NPC is loaded with the fall through of PC. The prediction process is repeated until a correct NPC is obtained. Evaluations show that it is easy to encounter a fall-through next PC after a few number of tries.

When the prediction mode ends, PC does not access BTB and BP anymore, and only NPC accesses BTB and BP to get the next next PC, shown in Figure 5.3c. In each new cycle, the next next PC is loaded into NPC and NPC is loaded into PC, thus both PC and NPC are acquired.

The processor enters the prediction mode when programs start or exceptions happen. The exceptions include BTB misses, branch mispredictions, load violations, and other exceptions that change program order. The processor exits the prediction mode once NPC is predicted correct.

The above discussion assumes that the instruction cache has single-cycle latency. Some modern processors, such as the Intel Core i7, pipeline L1 caches to achieve high clock frequency and may still access the tag array and the data array in parallel to reduce latency. ETL can be applied to pipelined instruction caches as well. The implementation depends on the pipeline depth of the tag array. For instance, if the tag array is pipelined into two cycles, the tag lookup should start two cycles earlier than the actual cache access for ETL to work. An additional next next PC (NNPC) should be obtained together with PC and NPC. The method of acquiring NPC can be extended to acquire NNPC. For simplicity, the rest of this chapter only considers instruction caches with single-cycle access latency. Some hardware modifications are needed to support ETL, which is discussed next.

5.2.3 Hardware Support for ETL

In the prediction mode, both PC and NPC access the tag array of the instruction cache, BTB, and BP simultaneously. The number of banks in the tag array of the instruction cache is doubled to support two simultaneous accesses. Each bank is independent and has its own address port. The bank addresses are interleaved. The data array of the instruction cache is not modified. Since the tag

array is much smaller than the data array, doubling the number of tag banks has only small impact on the overall cache. The number of banks in BTB is doubled similarly to support simultaneous accesses from both PC and NPC. Doubling the number of banks increases area but decreases access time and energy. Since NPC is always predicted as the fall through of PC in the prediction mode, there is no bank conflict.

The pattern history table (PHT) in BP is an array of two-bit counters implemented using SRAM. To balance the word line and bit line lengths and delay, the PHT SRAM is organized in square or near square shape, shown in Figure 5.4. Assume that the processor fetches eight instructions per cycle and eight continuous counters are read out to predict eight branches in the worst case. When a word line is activated, the entire row is accessed. Because each row contains many two-bit counters, 16 in this example, the existing PHT is already able to provide enough bandwidth for both PC and NPC accesses. Thus, the structure of PHT is not changed, and only its output data width is doubled, which incurs small area overhead. The PHT has two banks to support fetching across line boundaries.

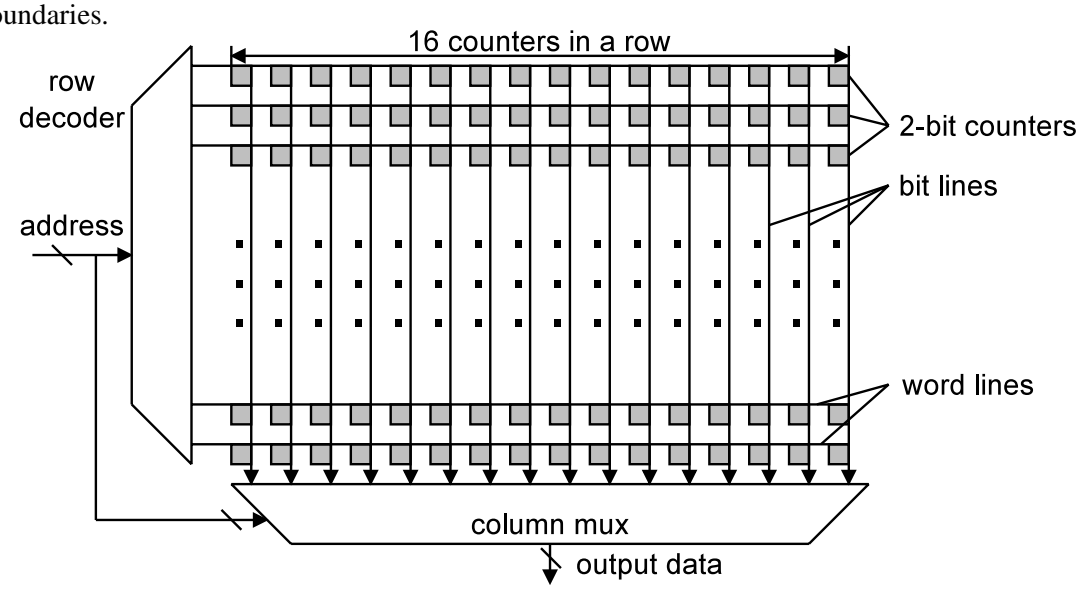


Figure 5.4: PHT SRAM array.

ETL does not need any modifications to the translation lookaside buffer (TLB). Since NPC is the fall through of PC in the prediction mode, their page addresses are the same and a single access to TLB is enough. In the rare case when their page addresses differ, NPC prediction is paused for

one cycle. When the processor is out of the prediction mode, only NPC accesses the TLB.

5.2.4 Working Flow of ETL

The working flow of ETL is shown in Figure 5.5, and a working example is given in Table 5.1. In cycle 1, the program starts and the prediction mode is entered. PC is loaded with 0, and NPC is predicted as the fall through of PC, which is 48. Because the matching way for PC is unknown, PC accesses all data ways. There is a taken branch in the fetch block at address 0 and its target address is 192. Thus, NPC is predicted wrong and the processor stays in the prediction mode.

Table 5.1: Working example of ETL. Assume six 64-bit instructions are fetched per cycle and program starts at address 0.

cycle	PC	NPC	prediction mode?	prediction correct?	cache access	exception
1	0	48	yes	no	all ways	no
2	192	240	yes	yes	all ways	no
3	240	288	no	null	matching way	no
4	288	48	no	null	matching way	no
5	48	96	no	null	matching way	yes
6	256	304	yes	yes	all ways	no
7	304	128	no	null	matching way	no
8	128	176	no	null	matching way	no

In cycle 2, PC is loaded with the branch target address, 192, and NPC is predicted as the fall through of PC, which is 240. PC accesses all data ways in the cache. There is no taken branch in the current fetch block, thus NPC is predicted correct and the processor will exit the prediction mode.

In cycle 3, PC is loaded with NPC and NPC is loaded with the next next PC acquired in cycle 2, which is 288. Since the matching way has been determined in cycle 2, PC only accesses the matching data way. In cycle 4 and 5, the processor remains out of the prediction mode and works similarly to cycle 3. Note there is a taken branch in the fetch block at address NPC in cycle 3. In cycle 5, an exception happens and brings the program execution to address 256. Since the next fetch address following address 256 is unknown, the processor will enter the prediction mode in the next cycle.

In cycle 6, the prediction mode is entered. PC is loaded with the exception target address, 256,

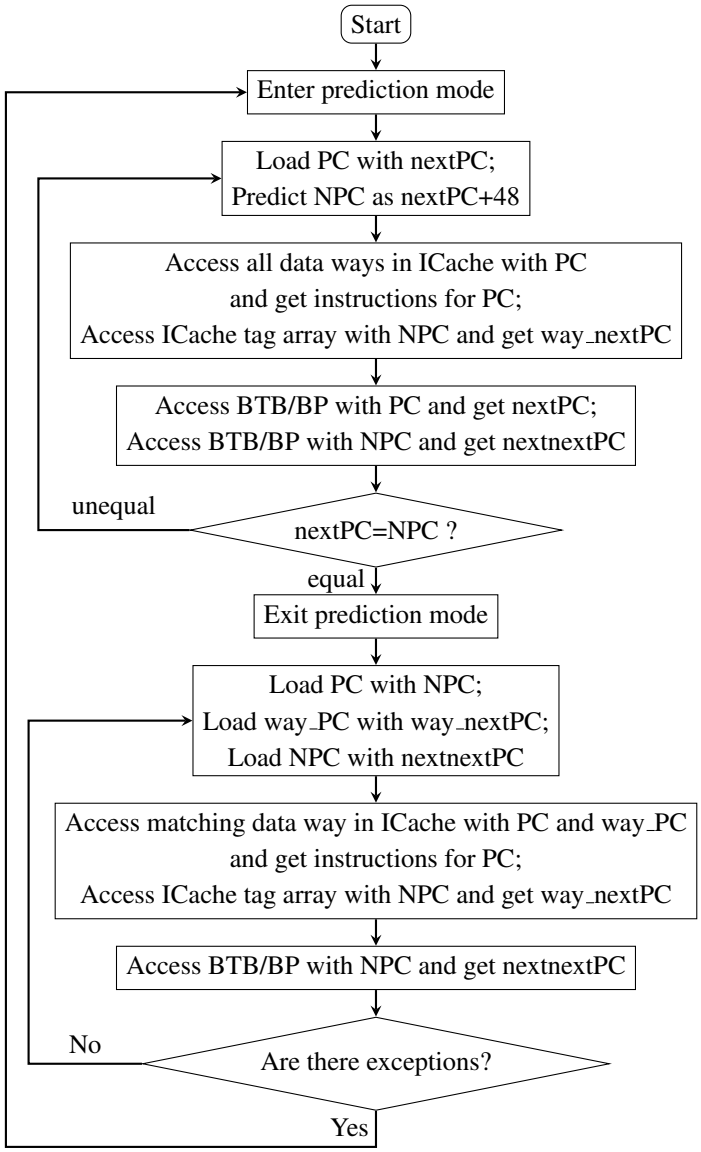


Figure 5.5: Working flow of ETL. Assume six 64-bit instructions are fetched each cycle.

and NPC is predicted as the fall through of PC, 304. The other operations in this cycle are similar to those in cycle 2. Cycles 7 and 8 are similar to cycles 3 and 4. There is a taken branch in the fetch block at address 304 in cycle 6, and the target address is 128.

5.3 Experimental Platform

This work utilizes FabScalar [17] as the experimental platform (Section 1.3) to accurately evaluate the area, delay, and power incurred by ETL. ETL is implemented on such an RTL superscalar core generated by FabScalar. The new processor is synthesized and simulated to accurately quantify the changes in area, delay, and power due to ETL. The parameters of the experimental processor are shown in Table 5.2.

Table 5.2: Parameters of the experimental processor.

Fetch/decode/rename/ dispatch width	6
Issue/RR/execute/WB width	8
Function units (simple, mult./div., branch, ld/st)	3,1,1,3
IQ, LSQ, ROB	128, 64, 256
Branch predictor	bimodal, 64K-entry PHT
BTB	4K-entry BTB
L1 I-Cache	32KB, 64-byte block, 4-way, 1 cycle
L1 D-Cache	64KB, 64-byte block, 4-way, 1 cycle
Unified L2	2MB, 64-byte block, 8-way, 18 cycles

The performance and cache activities are evaluated by simulating SPEC CPU2000 benchmarks on the experimental processor. A single simulation point is generated using SimPoint [67] for each benchmark. Only integer benchmarks are studied because the FabScalar infrastructure does not support floating point instructions. In each benchmark simulation, the total number of accesses to the instruction cache are collected. The total dynamic energy of the instruction cache is estimated as the product of the energy consumed by each access and the total number of accesses. The dynamic energy of BTB and BP are estimated similarly.

5.4 Results

5.4.1 Overhead of ETL

The hardware modifications required by ETL incur overhead. In the original experimental processor, the instruction cache contains two banks to provide enough bandwidth for fetching six instructions per cycle, and BTB contains eight banks to support six simultaneous read from the six fetched instructions. ETL doubles the number of banks in the cache tag array from 2 to 4, and the number of banks in BTB from 8 to 16. The overhead incurred to the instruction cache and BTB are shown in Table 5.3. Doubling the banks in the cache tag array increases the tag area by 23.39% but decreases the access time and read energy. Because the tag array is much smaller than the data array, the impact on the overall instruction cache is tiny. Thus, ETL causes negligible overhead in the instruction cache. BTB has an area overhead of 23.91%, but the access time and read energy both decrease.

Table 5.3: Percent overhead incurred by ETL in instruction cache, BTB, and PHT.

	Area	**Access time**	**Read energy**
Tag overhead	23.39%	-9.07%	-3.15%
ICache overhead	0.24%	0%	-0.03%
BTB overhead	23.91%	-15.14%	-0.46%
PHT overhead	4.66%	0.14%	20.78%

Doubling the output data width of PHT incurs small area overhead and negligible access time increase, shown in Figure 5.3. The read energy of PHT increases by 20.78%, however, this energy increase only happens in the prediction mode when both PC and NPC access the PHT. When the prediction mode ends, only NPC accesses PHT and the read energy is as normal.

ETL adds extra control logic in the fetch stage, the overhead of which is shown in Table 5.4. The area of the extra fetch control logic is only a small fraction of the core area, there is no delay overhead, and the power overhead is negligible.

Table 5.4: Percent overhead of the extra instruction fetch control logic incurred by ETL.

	Area	**Delay**	**Power**
Overhead	1.95%	0%	0.43%

Overall, ETL incurs insignificant area overhead and does not affect the critical path delay of the

processor. The small energy increase in PHT and the extra control logic will be completely offset by the savings in the instruction cache.

5.4.2 Energy Savings of ETL

ETL effectively removes most of the non-matching data way accesses in the L1 instruction cache. Figure 5.6 shows the percent of accesses that read all data ways and that read only the matching way. On average, ETL removes 92% instruction cache accesses that read all data ways.

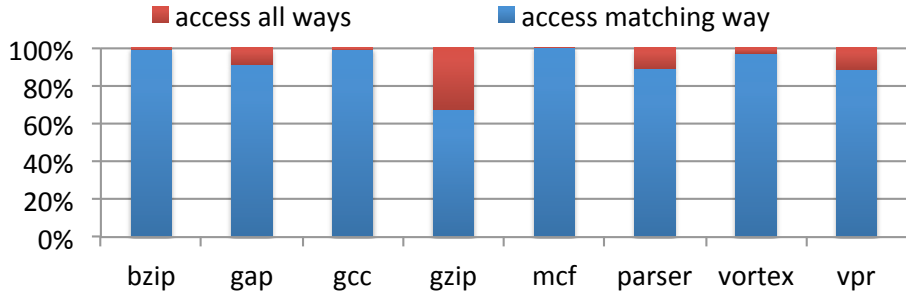


Figure 5.6: Percent of L1 instruction cache accesses that read all ways and that read only the matching way using ETL.

In the prediction mode, both PC and NPC access the cache tag array, BTB, and BP, which increases the accesses to these components compared with the conventional processor without ETL, shown in Figure 5.7. The increase of the accesses is small for most of the benchmarks except for *gzip*, in which it takes longer to encounter fall-through NPCs. The increased accesses to the tag array, BTB, and BP consume extra energy, however, this amount of energy can be fully offset by the energy savings in the cache data array. Note that outside the prediction mode, only NPC accesses the tag array, BTB, and BP, while PC does not need to access them.

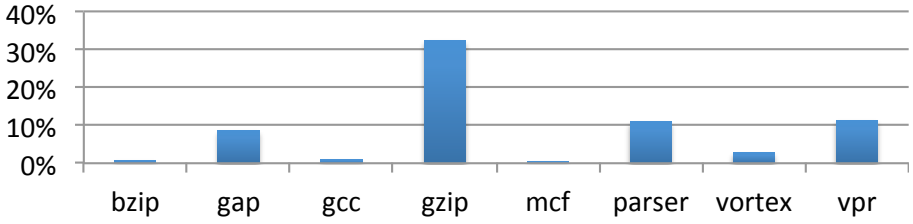


Figure 5.7: Percent increase of accesses to the tag array, BTB, and BP when ETL is used.

Figure 5.8 shows the resulting energy savings using ETL. The majority of energy reduction comes from the cache data array, since ETL removes most of the non-matching data way accesses.

On average, the read energy of the data array is reduced by 69%. Doubling the banks of the cache tag array contributes to energy reduction since the tag access energy is reduced, however, this contribution may be offset by the simultaneous tag accesses from PC and NPC in the prediction mode. Because the tag array is much smaller than the data array, its energy impact on the entire instruction cache is negligible. On average, the read energy of the L1 instruction cache is reduced by 68% after considering the tag energy. The simultaneous accesses to BTB and BP by both PC and NPC in the prediction mode cause some energy overhead. After taking this energy overhead into account, the average read energy reduction in instruction cache drops by only 1%. This is because BTB and BP have much lower energy consumption than the instruction cache, and the increased accesses to BTB and BP are only a few percent for most benchmarks.

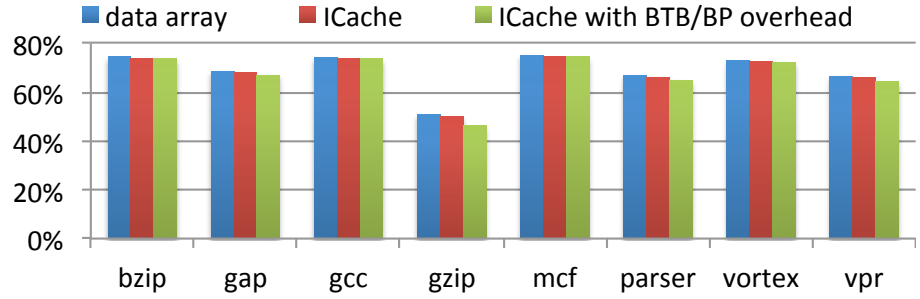


Figure 5.8: Percent read energy reduction in the cache data array, the L1 instruction cache, and the L1 instruction cache with BTB/BP overhead considered.

5.4.3 Further Discussion

The effectiveness of ETL depends on two factors. One is the duration of the prediction mode, and the other is how often the processor enters the prediction mode. ETL is more effective when the duration of the prediction mode is short and the processor enters the prediction mode less often.

The duration of the prediction mode depends on the number of tries taken to predict NPC correct. NPC prediction is correct when there are no taken branches in the current fetch block and the next PC is fall through. Thus, the number of tries depends on the frequency of branches in the program and the processor's fetch width. ETL is more effective for programs with less branches and for processors that fetch fewer instructions per cycle. Table 5.5 shows the average number of tries before NPC is predicted correct in the prediction mode. Most of the benchmarks, except

for *gzip*, have an average number of tries less than or equal to 5, which proves that NPC can be predicted correct quickly. On average, *gzip* needs 20 tries before a fall-through NPC is encountered, and this explains why it has less energy savings than the other benchmarks. The average number of matching-way cache accesses after NPC is predicted correct until the next exception happens, shown in Table 5.5, indicates that the energy-saving non-prediction mode is much longer than the prediction mode for most benchmarks.

Table 5.5: Average No. of tries until NPC is predicted correct, and average No. of matching-way cache accesses after NPC is predicted correct until the next exception happens.

	bzip	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**vortex**	**vpr**
Avg. tries	2	3	4	20	1	5	2	4
Avg. accesses	196	29	465	24	277780	38	70	33

Upon exceptions that change program execution order, the processor enters the prediction mode. Figure 5.9 shows the average number of exceptions for executing one million instructions in each benchmark. The more exceptions there are, the less effective ETL is, which can be observed by comparing Figure 5.8 with Figure 5.9. Techniques that reduce exceptions, such as larger BTB, better branch predictors, and memory dependence prediction, can enhance the effectiveness of ETL.

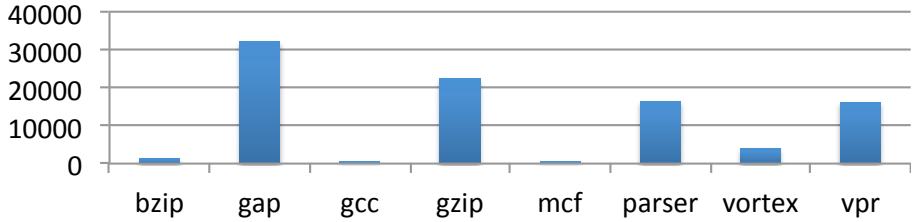


Figure 5.9: Average number of exceptions (BTB misses, branch mispredictions, load violations, etc.) during execution of one million instructions.

5.5 Comparison with Related Works

A couple of representative prior works are selected and compared with ETL, shown in Table 5.6. The data comes directly from [58,64,77] and the experimental methodology for each work can also be found there. The "all-way accesses removed" represents the percent of accesses to non-matching data ways that are removed by applying those techniques to the L1 instruction cache. For HotSpot cache [77], hitting the L0 cache is considered equal to removing accesses to non-matching data

ways. Compared with way prediction [58] and HotSpot, ETL removes more non-matching data way accesses, 92% versus 79% and 64%, respectively. The result of ETL will be even better if the experimental processor fetches fewer than six instructions per cycle. While TLC [64] stores way information in the extended TLB (eTLB) and theoretically removes all non-matching data way accesses, it flushes all cache lines in a replaced page on eTLB misses, causing unnecessary cache misses and thus performance loss. In addition, the eTLB increases the size of the TLB by more than four times. In terms of performance, way prediction incurs up to 20% performance loss for certain benchmarks and HotSpot incurs a worst-case 2-3% performance loss. The optimized TLC incurs a worst-case 2% CPI loss and increases the access latency of a 32KB cache by at least 14%. In comparison, ETL does not alter cache line replacement policies, increase cache access latency, or incur any performance loss. The hardware modifications required by ETL are much simpler than the other three techniques.

Table 5.6: Comparison with previous work.

	All-way accesses removed	**Performance loss**	**Hardware overhead**
Way prediction [58]	79%	Up to 20% performance loss, increase cache access latency on wrong predictions	Very large way prediction tables
HotSpot [77]	64%	Up to 3% performance loss, increase cache access latency on L0 misses	Additional L0 cache
TLC [64]	100%	Up to 2% CPI loss, increase 32KB cache access latency by at lease 14%	Increase TLB size by more than 4 times
ETL	92%	No performance loss, does not increase cache access latency	Simple low-cost extensions to existing hardware

Chapter 6

Dynamic Insertion Throttling

6.1 Overview

The recent insertion policies that address cache thrashing and pollution [31,32,34,56,59,66,73,76] all depend on inserting only a fraction of the cache blocks with high priority to improve performance. The works addressing thrashing empirically determine the fraction $(1/\varepsilon)$ of the working set kept in the cache for thrashing workloads (ε is 32 or 64) [32, 59]. The works addressing pollution attempt to predict the high-reuse blocks and *always* insert them with high priority [66, 76]. However, such a mechanism without the knowledge of the optimal fraction cannot fully address thrashing and can cause cache under-utilization.

In this work, we argue that *maximizing performance by addressing cache pollution and thrashing depends on accurately determining the optimal fraction of the working set that should be kept in the cache*. We provide an understanding of how insertion policies help improve cache performance and (with that understanding) develop an analytic model to determine the optimal workload fraction that should be kept in the cache to maximize performance. Our model and analysis build upon the reuse distribution of the cache blocks. The *reuse distance* (RD) of a block is measured as the number of unique accesses in a set between two consecutive accesses to that block. A *reuse distance distribution* (RDD) demonstrates the total number of blocks with each RD. To this end, we show that insertion policies improve cache hit rate by manipulating the RD of cache blocks. By inserting blocks with low priority, insertion policies essentially make sure that these blocks get evicted at the

next miss. Therefore, if we consider that these blocks do not contribute to the RD, the number of unique accesses between two consecutive accesses decreases significantly when insertion policies are applied. We refer to this new reduced RD as the *effective reuse distance* (ERD). The key insight presented in this work is: a block will hit in the cache *only when the ERD becomes less than the associativity of the cache*. Therefore, we demonstrate that in order to maximize the hit rate, the rate that blocks are inserted with a high priority (1/ε) must be controlled in a way that it maximizes the fraction of blocks with ERD smaller than the cache associativity. Based on this insight, we present an oracle reuse model (ORM) to determine this optimal fraction when the insertion policy has the accurate knowledge about the RD of each block.

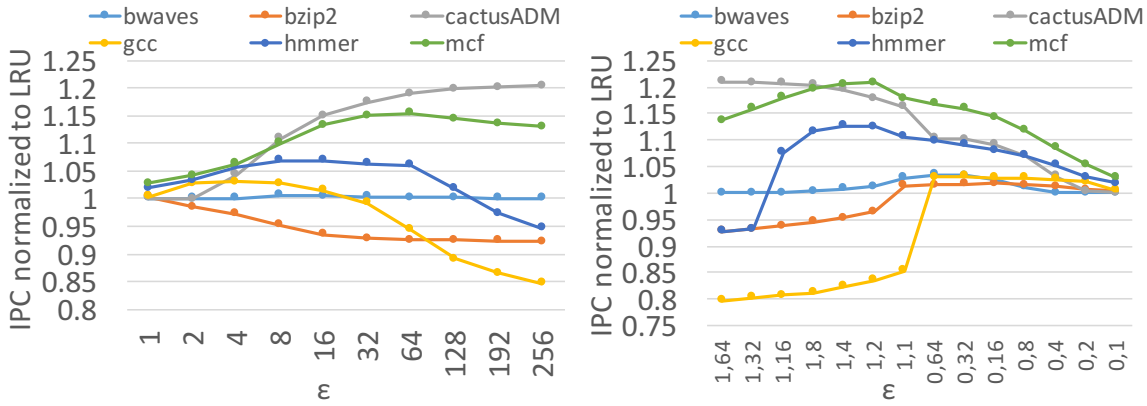
Unfortunately, in practical cases, it is hard to have precise information about the RD of each block. In this work, we present two practical models to determine the optimal fraction to maximize cache performance: *(i) Equal Block Model (EBM):* This model treats every block as equal. It is targeted towards the first category of works (e.g., BIP, BRRIP) that have no information of block RD and statically inserts every 32/64th block in the cache based on the empirically determined parameter ε. Our analytical model shows that by inserting a block with high priority after every ε misses, the ERD of high priority blocks essentially becomes reduced by ε times. Given the RDD of the workload, it is possible to obtain the optimal ε that would reduce the ERD such that the hit rate is maximized. *(ii) Reuse Differentiating Model (RDM):* This model is targeted towards the second category of works (e.g., SHiP, EAF) that use a predictor to differentiate between low-reuse and high-reuse blocks. Though these insertion policies increase the number of hits, RDM shows that such policies can result in thrashing or cache under-utilization when the fraction of the blocks inserted with high priority is not maintained properly.

Based on these practical models, we propose Dynamic Insertion Throttling (DIT) to insert the optimal fraction of the cache blocks with high priority that maximizes hit rate. First, we propose DIT-RD, which samples accesses to approximate the RDD and then uses that information to determine the optimal fraction based on our proposed models. Second, we propose DIT-SM, a simple set sampling approach to determine the optimal fraction from a set of possible values when no information about RD is available. We want to emphasize that our mechanisms are independent of

the insertion policy used in the cache and can improve any prior insertion policy.

6.2 Motivation

This section shows that *(i)* prior insertion policies that statistically insert $1/\varepsilon$ of the missed blocks with high priority (e.g., BIP, BRRIP) cannot fully address thrashing and *(ii)* prior insertion policies that attempt to predict high-reuse blocks and *always* insert them with high priority can cause either cache thrashing or under-utilization.



(a) BRRIP (b) SHiP

Figure 6.1: Normalized IPC over LRU for selected workloads as ε varies for BRRIP and SHiP insertion policies. The values on the x-axis represent ε. For SHiP, the 0/1 below the comma represents applying BRRIP to the predicted low-reuse/high-reuse blocks, and the values above the comma represent ε.

Figure 6.1 shows the sensitivity of performance as ε changes for one cache insensitive workload (`bwaves`), one LRU-friendly workload (`bzip2`), and four cache thrashing/polluting workloads (`cactusADM, gcc, hmmer, and mcf`) under BRRIP and SHiP insertion policies. For BRRIP, the performance of the cache-insensitive application `bwaves` almost does not depend on ε. LRU-friendly workload `bzip2` performs the best when all the blocks are inserted with high priority. The performance of the other thrashing/polluting workloads highly depends on what fraction of the blocks are protected in the cache. In some cases, it is possible to get 1.2X improvement by choosing the optimal ε, which varies depending on the workload.

The prediction mechanism used by SHiP is not ideal and can mispredict. If SHiP predicts too

many high-reuse blocks, always inserting them with high priority causes thrashing. To reduce this thrashing, only $1/\varepsilon$ of the predicted high-reuse blocks are inserted with high priority (apply BRRIP to predicted high-reuse blocks). Figure 6.1b shows that SHiP causes thrashing for `cactusADM`, `hmmer, and mcf`, because their performance benefit from keeping only a fraction of the predicted high-reuse blocks in the cache. This fraction varies depending on the application. On the other hand, if SHiP predicts too few high-reuse blocks, only inserting these blocks with high priority causes cache under-utilization. To address this problem, $1/\varepsilon$ of the predicted low-reuse blocks are inserted with high priority as well (apply BRRIP to predicted low-reuse blocks) in addition to inserting all the predicted high-reuse blocks with high priority. An example of such is `gcc` whose performance significantly improves when a fraction of the low-reuse blocks predicted by SHiP are also inserted with high priority.

We conclude that maximizing performance for cache polluting/thrashing workloads depends on dynamically detecting the optimal fraction of the working set that should be kept in the cache.

Goal. We have two goals in this work. First, we want to understand how insertion policies help improve performance and provide an analytical model to determine the optimal fraction. Second, we want a simple and efficient mechanism to detect this optimal fraction dynamically for each workload at runtime.

6.3 The Analytic Models

In this section, *(i)* we describe the definitions used in our analytical models, *(ii)* formalize the impact of insertion policies on cache hits, and *(iii)* describe the models that demonstrate the factors that determine the optimal fraction of the working set that should be kept in the cache.

6.3.1 Definitions

Cache Access Pattern. For a cache with size C and associativity A, the reference stream to a certain set S is represented as

$$[l_1][l_2]\cdots[l_k]\cdots[l_N]\cdots[l_N]\cdots[l_1]\cdots[l_2]\cdots[l_k],$$

where $l_k, 1 \leqslant k \leqslant N$ represents one of the N unique blocks mapped to set S. $[l_k]$ represents one or more consecutive references to the block l_k, and $\cdots$ represents zero or more references to any block in that set.

Reuse Distance (RD). *The **reuse distance** d of a block l_k, mapped to the set S, is defined as the number of unique block accesses to set S between two consecutive accesses to l_k.*

$$\cdots l_k \underbrace{[l_1][l_2]\cdots[l_d]}_{\text{reuse distance } d} l_k \cdots$$

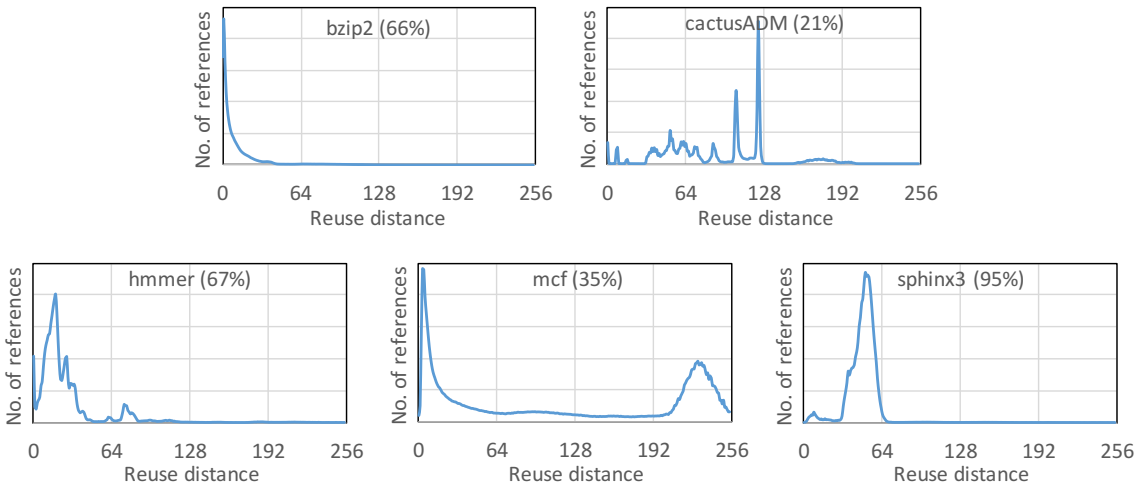


Figure 6.2: RDD of selected SPEC CPU2006 benchmarks. The percentages in brackets indicate the percent of blocks displayed in the diagram. The y-axis represents the number of references with the same RD.

Effect of RD on Workloads. The RD determines if a block will hit or miss the cache with the LRU replacement policy. A block with an RD, $d < A$ guarantees that the next reference to that block will hit the cache, as the number of unique blocks inserted in the cache between the consecutive accesses will not be able to evict the block. Similarly, an RD, $d \geqslant A$ guarantees that before the next reference to that block, it will get evicted from the cache and result in a cache miss. Figure 6.2 presents the reuse distance distribution (RDD) (i.e., the number of blocks with different RD) for some selected SPEC CPU2006 benchmarks (where the cache associativity is16). This figure shows that RDD varies significantly across the benchmarks. Based on RDD, cache references can be divided into three categories:

(i) LRU-friendly. If a large fraction of the blocks exhibits RD less than the associativity, the RDD curve will be skewed towards $d < A$. Most of the blocks will hit the cache in this case, making the workload LRU-friendly. Figure 6.2 shows that `bzip2` is an LRU-friendly workload.

(ii) Thrashing. If a workload has a large fraction of blocks with RD greater than the associativity, all these blocks will evict each other thrashing the cache without providing any cache hit. Figure 6.2 shows that `cactusADM` and `sphinx3` are thrashing workloads. A streaming workload has an RD, $d = \infty$ providing no cache hit.

(iii) Mixed. There are some workloads which have both types of accesses. For example, `hmmer` and `mcf` have some LRU-friendly references, but also a large fraction of thrashing references.

6.3.2 Insertion Policies and Reuse Distance

Recent insertion policies [31,32,59] address cache thrashing by inserting only $1/\varepsilon$ of missed blocks with high priority. The vast majority of the blocks inserted with low priority are evicted out of the cache very quickly (at the next miss). For simplicity, if we assume that these blocks occupy no cache space, only the blocks inserted with high priority will affect RD. Thus, a new RD, called the **effective reuse distance (ERD)** can be defined. *For a block l_k mapped to set S, its ERD, d' is defined as the number of unique block accesses to set S between two consecutive accesses to l_k, that are inserted with high priority*. As a result, *insertion policies essentially reduce the RD of the blocks and if the ERD becomes less than the cache associativity, the block will be retained long enough in the cache to receive a hit*.

Consider the following access pattern to a certain cache set S:

$$\cdots l_k \underbrace{[l_1][l_2]\cdots[l_d]}_{d>A} l_k \cdots,$$

where the RD of block l_k is larger than the cache associativity A. The second reference to l_k will miss under the LRU replacement policy. However, if only every ε_{th} block is actually kept in the set S and we make the assumption that the possibility of referencing any block l_i $(1 \leqslant i \leqslant d)$ is equally likely at any time, the blocks inserted in the cache becomes:

$$\cdots l_k \underbrace{[l_1][l_{\varepsilon+1}][l_{2\varepsilon+1}]\cdots[l_{\lfloor \frac{d}{\varepsilon} \rfloor \varepsilon+1}]}_{d' = \lfloor \frac{d}{\varepsilon} \rfloor + 1 = \lceil \frac{d}{\varepsilon} \rceil} l_k \cdots \ (d \mod \varepsilon \neq 0) \text{ or}$$
$$\cdots l_k \underbrace{[l_1][l_{\varepsilon+1}][l_{2\varepsilon+1}]\cdots[l_{(\lfloor \frac{d}{\varepsilon} \rfloor - 1)\varepsilon+1}]}_{d' = \lfloor \frac{d}{\varepsilon} \rfloor = \lceil \frac{d}{\varepsilon} \rceil} l_k \cdots (d \mod \varepsilon = 0).$$

The ERD, d' of block l_k then becomes approximately $\lceil \frac{d}{\varepsilon} \rceil$. Since only blocks inserted with a high priority affect the replacement of block l_k, the second reference to l_k will hit as long as

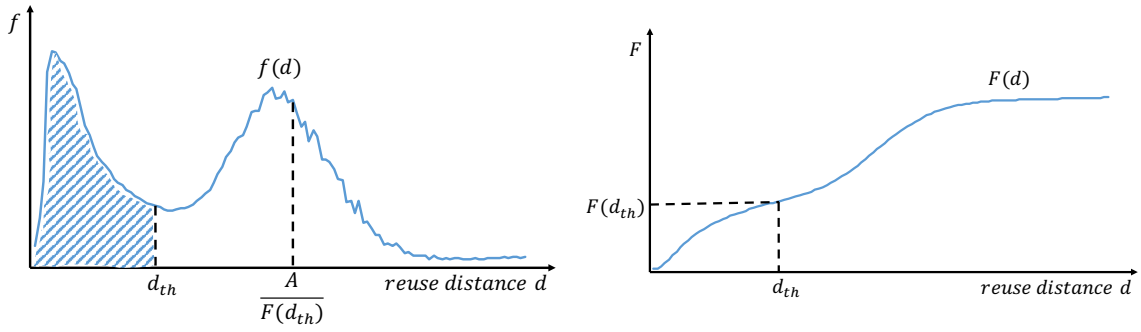
$$d' = \lceil \frac{d}{\varepsilon} \rceil < A \Rightarrow \varepsilon > \frac{d}{A} \tag{6.1}$$

Therefore, we provide a key insight that *the ERD, d' determines whether the second reference to a thrashing block will hit or miss the cache when insertion policies (e.g., BIP, BRRIP) are applied to the cache*. As a result, it is possible to control the insertion rate of the high priority blocks ($1/\varepsilon$) such that the ERD maximizes the number of cache hits. Based on this insight, we propose analytical models to determine the optimal fraction of the working set based on RDD. First, we provide an oracle model that has information about the RD of each block (Section 6.3.3). Then, we propose two practical models applicable to the current insertion policies that do not have any knowledge about the RD of blocks (Section 6.3.4 and 6.3.5).

6.3.3 Oracle Reuse Model (ORM)

To reveal the relationship between the optimal fraction and its determinant factors, the ORM assumes that the RD of all cache references are known by oracle. With this oracle information, missed blocks with small RD should be given higher priority than missed blocks with large RD. To meet this priority condition, the following ideal insertion policy can be applied when a cache miss happens: *the missed block is inserted with high priority only if its RD is smaller than or equal to a* threshold RD, *d_{th}, otherwise the missed block is inserted with low priority*. Next we discuss how to determine this *optimal threshold RD* so that the hit rate can be maximized.

The impact of d_{th} on hit rate is illustrated by Figure 6.3. This figure demonstrates two parameters: (i) the RDD of the blocks mapped to a certain cache set S, which is represented by function $f(d)$ in Figure 6.3a, and (ii) the cumulative distribution of the RD of the blocks mapped to set S,



(a) (b)

Figure 6.3: ORM under the case of $d_{th} < d_{opt}$. The RDD $f(d)$ represents the number of references that have an RD of d. The cumulative distribution $F(d)$ represents the percent of references that have an RD smaller than or equal to d.

which is represented by the function $F(d)$ in Figure 6.3b. This cumulative distribution represents the percent of references that have an RD smaller than or equal to d, and is defined as follows:

$$F(d) = \frac{\sum_{x=0}^{d} f(x)}{\sum_{x=0}^{\infty} f(x)} \tag{6.2}$$

Note that only a single set, rather than all the sets in the cache is considered here, which makes the analysis easier to understand. In addition to that, the RDD shown in Figure 6.3 is only for illustrative purpose. Our models do not depend on the actual RDD shape and are applicable for any RDD.

For a block l_k with an RD of d

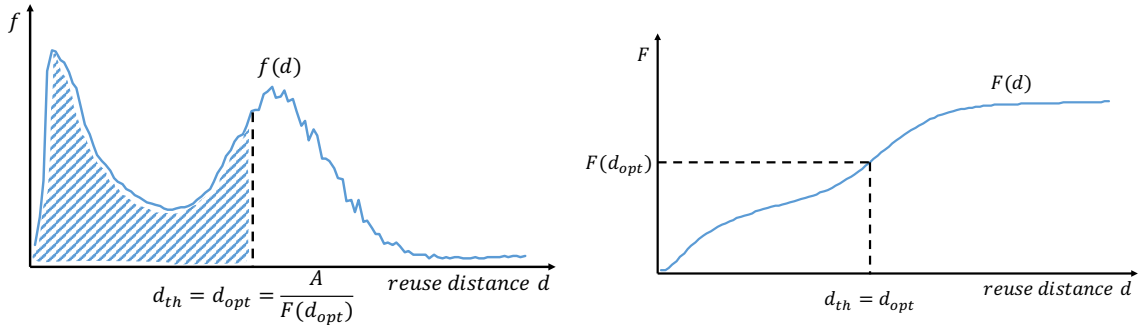
$$\cdots l_k \underbrace{[l_1][l_2]\cdots[l_d]}_{d} l_k \cdots,$$

its ERD, d' becomes $dF(d_{th})$ after the ideal insertion policy is applied. The access stream becomes

$$\cdots l_k \underbrace{[l_{i1}][l_{i2}]\cdots[l_{idF(d_{th})}]}_{d' = dF(d_{th})} l_k \cdots,$$

where $l_{i1}, l_{i2}, \cdots, l_{idF(d_{th})}$ are the blocks inserted with high priority and they are a subset of blocks $l_1, l_2, \cdots, l_d$. To understand this, recall that $F(d_{th})$ represents the percent of blocks with RD smaller than or equal to d_{th}. Thus, for d unique blocks, approximately $dF(d_{th})$ blocks will be inserted with a high priority using the ideal insertion policy, under the assumption that the possibility of referencing

any block in the working set is equally likely at any time. This assumption may not stand when the working set is large and consists of different program phases. However, if the working set is divided into individual phases and the ORM is applied to each single phase, this assumption will be mostly true.



(a) (b)

Figure 6.4: ORM under the case of $d_{th} = d_{opt}$.

When the ERD is smaller than the cache associativity such that:

$$d' = dF(d_{th}) < A \Rightarrow d < \frac{A}{F(d_{th})}, \tag{6.3}$$

the next reference to block l_k will hit, which means that any block with RD smaller than $\frac{A}{F(d_{th})}$ will receive a hit on the next reference. The hit rate is maximized for set S when

$$d_{th} = \frac{A}{F(d_{th})} \tag{6.4}$$

In order to determine why this condition maximizes hits, let's consider three cases: *(i)* $d_{th} < \frac{A}{F(d_{th})}$, *(ii)* $d_{th} = \frac{A}{F(d_{th})}$, and *(iii)* $d_{th} > \frac{A}{F(d_{th})}$, respectively shown in Figure 6.3, 6.4, and 6.5.

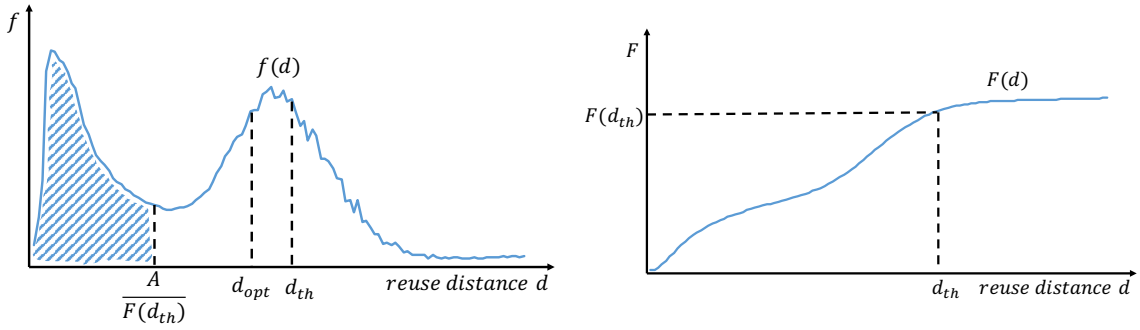
When $d_{th} < \frac{A}{F(d_{th})}$, shown in Figure 6.3, all the blocks inserted with a high priority will receive hit on the next references. The total number of hits in this case is given by

$$S = \sum_{d=0}^{d_{th}} f(d) \tag{6.5}$$

shown by the shaded area under the $f(d)$ curve in Figure 6.3a. However, any block with RD larger

than d_{th} and smaller than $\frac{A}{F(d_{th})}$ can receive hit too if inserted with high priority. This means that the total number of hits can still be increased if d_{th} is increased.

When $d_{th} = \frac{A}{F(d_{th})}$, shown in Figure 6.4, the total number of hits, represented by the shaded area, is maximized. This is because if $d_{th} > \frac{A}{F(d_{th})}$, as shown in Figure 6.5, any block with reuse distance larger than $\frac{A}{F(d_{th})}$ will be evicted out of the cache before the next reference, reducing the shaded area and thus the total number of hits. In other words, in this case too many high-reuse blocks are kept in the cache and they start to cause thrashing.



(a) (b)

Figure 6.5: ORM under the case of $d_{th} > d_{opt}$.

The optimal threshold RD, d_{opt} is given by

$$d_{opt} = \frac{A}{F(d_{opt})} \tag{6.6}$$

The maximum number of hits to set S is given by

$$S_{max} = \sum_{d=0}^{d_{opt}} f(d) \tag{6.7}$$

represented by the shaded area in Figure 6.4a.

ORM assumes that the RD of all cache blocks are known by oracle. Unfortunately, this is impossible in practice. Therefore, next we provide two practical models applicable to recent insertion policies that address cache thrashing.

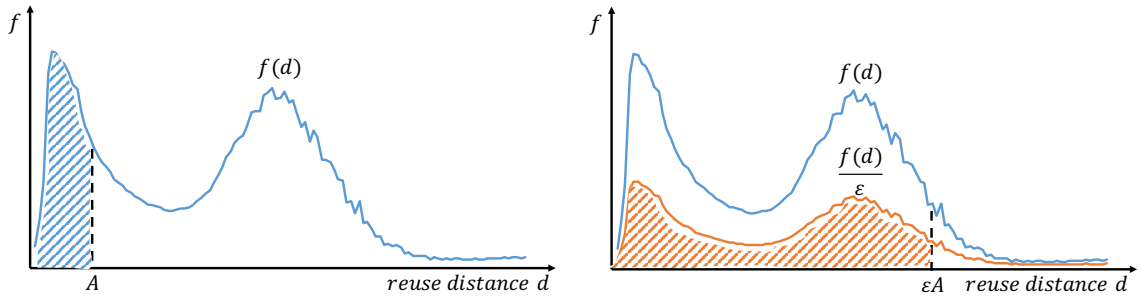
6.3.4 Equal Block Model (EBM)

EBM targets insertion policies that statistically keep a fraction of the working set in the cache and do not differentiate the reuse behavior of missed blocks, such as the bimodal insertion policies, BIP and BRRIP, which insert one out of every ε missed blocks with high priority and the other $\varepsilon - 1$ blocks with low priority. EBM aims to answer the following question: *what should* ε *be in order to maximize hit rate using the bimodal insertion policy?*

Consider a workload with RDD $f(d)$ for a cache set S, shown in Figure 6.6a. Using the traditional LRU replacement policy, all missed blocks are inserted with high priority at the MRU position. Only blocks with RD smaller than the cache associativity will receive hit, thus the total number of hits for LRU replacement policy is given by

$$S = \sum_{d=0}^{A-1} f(d) \tag{6.8}$$

represented by the shaded area shown in Figure 6.6a.



(a) (b)

Figure 6.6: EBM.

Inserting all missed blocks at the MRU position may maximize the number of hits for LRU-friendly workloads, however, this is rarely true for thrashing workloads, whose hit rate can be increased by using the bimodal insertion policies. To see why, the RDD of the blocks inserted with high priority are plotted, shown by the orange curve in Figure 6.6b. Since the blocks inserted with high priority are statistically selected, their RDD function is approximately $\frac{f(d)}{\varepsilon}$. As discussed in Section 6.3.2, for a block l_k with RD d

$$\cdots l_k \underbrace{[l_1][l_2]\cdots[l_d]}_{d} l_k \cdots,$$

its ERD becomes $d' = \lceil \frac{d}{\varepsilon} \rceil$ after applying the bimodal insertion policies. As long as d' is smaller than the cache associativity A

$$d' = \lceil \frac{d}{\varepsilon} \rceil < A \Rightarrow d < \varepsilon A \tag{6.9}$$

the next reference to block l_k will hit. In other words, for any block on the orange curve shown in Figure 6.6b that has an RD smaller than εA, the next reference to the block will hit. Thus, the total number of hits when applying the bimodal insertion policies is represented by the shaded orange area in Figure 6.6b, given as follows:

$$S(\varepsilon) = \sum_{d=0}^{\varepsilon A-1} \frac{f(d)}{\varepsilon} = \frac{1}{\varepsilon} \sum_{d=0}^{\varepsilon A-1} f(d) \tag{6.10}$$

If this shaded orange area is larger than the shaded blue area in Figure 6.6a, the bimodal insertion policies can increase the total number of hits and reduce thrashing. The number of hits $S(\varepsilon)$ is a discrete function of ε and it is maximized when its derivative equals to zero

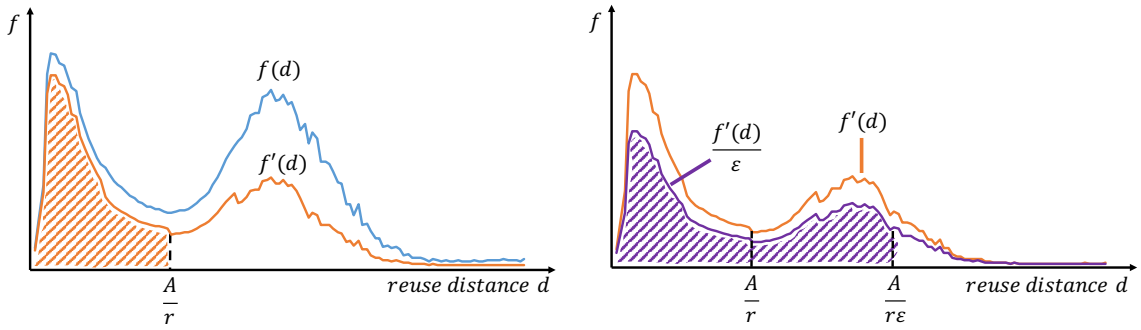
$$S'(\varepsilon) = \frac{S(\varepsilon+1) - S(\varepsilon)}{\varepsilon+1-\varepsilon} = 0 \Rightarrow S(\varepsilon+1) - S(\varepsilon) = 0 \tag{6.11}$$

which gives the optimal ε that maximizes hit rate.

6.3.5 Reuse Differentiating Model (RDM)

RDM targets insertion policies that exploit some prediction mechanisms to differentiate high-reuse blocks from low-reuse blocks, such as SHiP and EAF. These insertion policies insert the predicted high-reuse blocks with high priority and the predicted low-reuse blocks with low priority. However, as discussed in Section 6.2 they cannot fully address thrashing. The reason is that their prediction mechanisms are not ideal and can mispredict. Thus, the predicted high-reuse blocks are actually a mixture of both high-reuse and low-reuse blocks. Figure 6.7a shows the RDD of the predicted high-reuse blocks, the orange curve $f'(d)$, as well as the original RDD of all blocks, the blue curve

$f(d)$. The orange curve covers both short and long reuse distances, but has a higher portion of high-reuse blocks than the blue curve.



(a) (b)

Figure 6.7: RDM.

Assume block l_k is a predicted high-reuse block and has an RD of d

$$\cdots l_k \underbrace{[l_1][l_2]\cdots[l_d]}_{d} l_k \cdots.$$

Further assume that the fraction of predicted high-reuse blocks is r. With the reuse prediction mechanisms, the access pattern of the blocks inserted with high priority is as follows:

$$\cdots l_k \underbrace{[l_{i1}][l_{i2}]\cdots[l_{idr}]}_{d'=dr} l_k \cdots,$$

where $l_{i1}, l_{i2}, \cdots, l_{idr}$ are the predicted high-reuse blocks, a subset of blocks $l_1, l_2, \cdots, l_d$. The ERD becomes $d' = dr$ if the possibility of referencing any block in the working set is equally likely at any time. As long as

$$d' = dr < A \Rightarrow d < \frac{A}{r} \tag{6.12}$$

the next reference to block l_k will hit. Thus, the total number of hits for the traditional insertion policies that exploit reuse prediction mechanisms is represented by the orange shaded area in Figure 6.7a, given by

$$S = \sum_{d=0}^{\frac{A}{r}} f'(d) \tag{6.13}$$

If the orange curve has a thrashing RDD, which is the case for *cactusADM*, *hmmer*, *mcf*, and *sphinx3* shown in Figure 6.2, inserting all predicted high-reuse blocks with high priority can still

cause thrashing. To address this problem, the bimodal insertion policies, BIP and BRRIP, can be applied to the blocks on the orange curve and keep only a fraction of the predicted high-reuse blocks in the cache. According to EBM, the total number of hits would become

$$S(\varepsilon) = \sum_{d=0}^{\frac{A}{r\varepsilon}} \frac{f'(d)}{\varepsilon} = \frac{1}{\varepsilon} \sum_{d=0}^{\frac{A}{r\varepsilon}} f'(d) \tag{6.14}$$

represented by the purple area in Figure 6.7b. If this shaded purple area is larger than the shaded orange area in Figure 6.7a, thrashing is reduced. The optimal ε that maximizes the total number of hits is given by

$$S'(\varepsilon) = \frac{S(\varepsilon+1) - S(\varepsilon)}{\varepsilon+1-\varepsilon} = 0 \Rightarrow S(\varepsilon+1) - S(\varepsilon) = 0 \tag{6.15}$$

6.3.6 Protecting High-Reuse Blocks

For mixed workloads with both high-reuse and low-reuse blocks, cache pollution can happen when low-reuse blocks evict high-reuse blocks out of the cache, hurting performance. As discussed in EBM, for a block l_k with an RD of d, to make sure that the second reference to block l_k hits, the ERD of l_k after applying the bimodal insertion policies should satisfy Equation (6.9).

This means that up to $A-1$ blocks are allowed to be inserted with high priority between the two adjacent accesses to l_k. If these $A-1$ blocks are largely low-reuse, they could evict the high-reuse blocks already present in the cache set, causing misses when these high-reuse blocks are referenced again. To prevent the high-reuse blocks from being evicted, the number of blocks inserted with high priority should be reduced. This can be realized by allocating fewer ways in the set for low-reuse blocks. In EBM, all A ways are allocated to low-reuse blocks. To protect high-reuse blocks, the modified EBM allocates fewer ways, represented by A' ($A' \leqslant A$), to low-reuse blocks. A' is called the *reserved associativity*. With this modification, to make sure the second reference to l_k hits, the ERD, d' should satisfy

$$d' = \lceil \frac{d}{\varepsilon} \rceil < A' \Rightarrow \varepsilon > \frac{d}{A'} \tag{6.16}$$

meaning that ε needs to be larger in order to protect high-reuse blocks.

6.3.7 Sensitivity to Cache Size and Associativity

The analytic models show that the optimal fraction is related to cache size and associativity. In this section, the sensitivity of the optimal fraction to cache size and associativity is discussed using EBM. Suppose that the cache size is increased by α times to αC, where α is a positive integer. The number of sets will increase by α times. For a given workload, the number of blocks mapped to a given set S will decrease by approximately α times. Since RD is measured using the number of unique blocks mapped to the same set, the RD will decrease by approximately α times. In addition, the number of references with the same RD will also decrease by approximately α times. Thus, the RDD for a given set S becomes approximately $\frac{f(\alpha d)}{\alpha}$, which is lower and left-pushed compared with the original $f(d)$. The total number of hits becomes

$$S(\varepsilon) = \sum_{d=0}^{\varepsilon A-1} \frac{f(\alpha d)}{\alpha \varepsilon} \tag{6.17}$$

which is also lower and left-pushed compared with the original $S(\varepsilon)$. Thus, *the optimal* ε *that maximizes the number of hits decreases as cache size is increased, meaning that more blocks will be inserted with high priority*. The above analysis also applies when the cache size is decreased, in which case α is a positive real number smaller than 1 and ε *increases as cache size decreases*.

To see how cache associativity affects the optimal ε, suppose that the cache size is fixed and the associativity is increased by β times to βA, where β is a positive integer. The number of sets will decrease by β times, thus the number of blocks mapped to a given set S will increase by β times. The RD will increase by approximately β times. In addition, the number of references with the same RD will also increase by approximately β times. Thus, the RDD for a given set S becomes $\beta f(\frac{d}{\beta})$. The total number of hits becomes

$$S(\varepsilon) = \sum_{d=0}^{\varepsilon\beta A-1} \frac{\beta f(\frac{d}{\beta})}{\varepsilon} = \beta \sum_{d=0}^{\varepsilon A-1} \frac{f(d)}{\varepsilon} \tag{6.18}$$

which has similar form as the original $S(\varepsilon)$. Although the new RDD, $\beta f(\frac{d}{\beta})$ is taller and right-pushed compared with the original $f(d)$, this effect on ε is offset by the increase of the associativity.

Thus, *the number of hits, as well as the optimal* ε*, is insensitive to changes of cache associativity*. The analysis also applies when the associativity decreases.

6.4 Dynamic Insertion Throttling

The analytical models have demonstrated that the optimal fraction of the working set that should be kept in the cache depends on the RDD of the workloads. Since different workloads have different RDDs, this optimal fraction varies as workload behavior changes and should be carefully controlled. This work proposes dynamic insertion throttling (DIT) to determine the optimal fraction at runtime. Two approaches are proposed to implement DIT. The first approach, DIT-RD, samples cache accesses to approximate the RDD and then uses that information to determine the optimal fraction based on the proposed analytical models. The second approach, DIT-SM, uses a simple set sampling based approach to determine the optimal fraction from a set of possible values when no RDD information is available.

6.4.1 DIT-RD

DIT-RD uses the RD sampling technique described in [22] to obtain an approximation of RDD at runtime. The technique samples a fraction of the total cache sets and uses their RDD to approximate the RDD of the application. The sampling sets can be randomly selected. It has been shown that sampling only $1/64$ of the total sets is sufficient to capture the behavior of an application [22, 32, 39, 59]. To get the RD of cache references, a FIFO is attached to each of the sampling sets. The addresses of all the cache references to each sampling set are inserted to its attached FIFO. Each new access to a sampling set compares its address with the set's FIFO entries to determine its RD. Since the RD in this work only measures unique references, a valid bit is added to each FIFO entry to invalidate previously inserted duplicate addresses. An RD counter array is then used to store the sampled RDs and obtain the RDD. The i^{th} counter in the array stores the number of references with an RD of i. Prior work [22] has shown that an array of 256 16-bit RD counters is sufficient to get an accurate RDD sample for most applications and the hardware overhead is

manageable. If any of the 256 RD counters saturates, the entire RD counter array is cleared and RDD sampling is restarted. Clearing the RD counter array makes it possible to capture the reuse behavior changes in the workload and get up-to-date RDD.

With the RDD information available at runtime, DIT-RD applies the analytical models to obtain the optimal fraction. When applying EBM, Equation (6.10) is applied to the sampled RDD to find the ε that maximizes the total number of hits. When applying RDM, two RD counter arrays are used to store the RDDs of both the predicted high-reuse blocks and the predicted low-reuse blocks. Equation (6.14) is applied to the RDD of the high-reuse blocks to determine the optimal ε. The fraction of the predicted high-reuse blocks, r, in Equation (6.14) can be easily obtained by two counters. If the sampled RDD indicates that the workload is LRU-friendly and r is low, it is likely that the insertion policies predict too few high-reuse blocks and the cache is under-utilized. In this case, Equation (6.14) is applied to the RDD of the low-reuse blocks to determine what fraction of the low-reuse blocks should also be inserted with high priority in order to remove cache under-utilization.

When deriving the optimal ε using Equation (6.10) and Equation (6.14), Equation (6.16) is also applied to adjust the ε to protect high-reuse blocks. The reserved associativity A' in Equation (6.16) is determined by subtracting the number of high-reuse blocks from the cache associativity A. A block is regarded as high-reuse if it is reused after insertion. A' typically ranges from 1 to 8 when the cache associativity is 16. In fact, our evaluations show that always fixing A' at 1 actually performs better.

Although the analytical models are based on the RDD of a single set, getting an RDD for each cache set will incur exorbitant hardware overhead. DIT-RD uses a single RDD for all the cache sets and obtains a single optimal fraction for the entire cache. The evaluations show that this still gives satisfying results.

When applied to multi-core processors, each processor maintains its own RDD sampler and applies the analytical models independently, similar to [31].

The drawback of DIT-RD is that the RD sampler and the control logic incur hardware overhead, which is evaluated in detail in [22]. Section 6.4.2 proposes DIT-SM, a low-cost approach to find

the optimal ε.

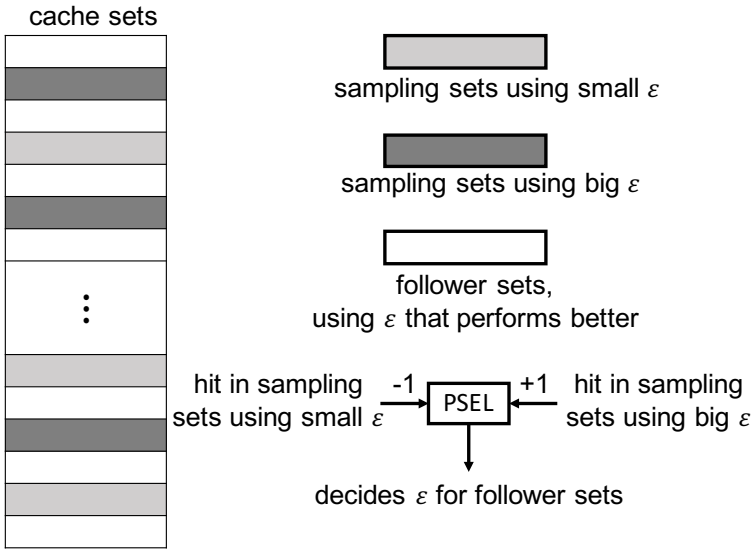


Figure 6.8: DIT-SM using set sampling.

6.4.2 DIT-SM

DIT-SM requires negligible additional hardware. Instead of applying the analytical models, DIT-SM dynamically samples a predefined set of possible fractions and selects the best performing one as the optimal fraction. It uses an idea similar to set dueling [59]. The total cache sets are divided into sampling sets and follower sets. In set dueling, half of the sampling sets are dedicated to the LRU policy ($\varepsilon = 1$) and the other half are dedicated to the BIP policy ($\varepsilon = 32$ *or* 64). Different from set dueling, the ε for the sampling sets are not fixed and can dynamically change in DIT-SM. Half of the sampling sets use a big ε, and the other half use a small ε, shown in Figure 6.8. The follower sets use the ε from the half of the sampling sets that perform better.

The performance of the two types of the sampling sets are evaluated periodically using the policy selector (PSEL) counter. Upon hit to the sampling sets using the big ε, the PSEL counter is incremented by 1, and upon hit to the sampling sets using the small ε, the PSEL counter is decremented by 1. A miss counter counts the number of misses to both sampling sets. When the counter reaches a preset threshold, the evaluation is triggered. The preset threshold should guarantee that enough misses occur to the sampling sets so that the performance difference between the big

Table 6.1: Possible ε values used by DIT-SM.

	possible values
ε	1, 2, 4, 8, 16, 32, 64, 128, 256

and small ε parameters is noticeable. Evaluations show that a miss counter threshold of 65536 gives satisfying results.

When the evaluation is triggered, the PSEL counter is compared with its initial value $PSEL_{init}$. If $PSEL \geqslant PSEL_{init}$, the big ε performs better. The ε of the follower sets is set to the big ε, and both the big ε and the small ε are incremented. Otherwise, the small ε performs better. The ε of the follower sets is set to the small ε, and both the big ε and the small ε are decremented. After every evaluation, PSEL is reset to $PSEL_{init}$. The set of predefined possible ε is shown in Table 6.1. The rule of selecting the possible ε is that they should cover the optimal ε of a wide range of applications and the performance difference between two adjacent values should be noticeable. When ε is incremented or decremented, ε is set to the next adjacent larger or smaller value.

The insertion policies that use reuse prediction mechanisms, e.g., SHiP and EAF, can predict either too many or too few blocks as high priority. When too many blocks are inserted with high priority, thrashing can still happen. In this case, DIT-SM keeps a fraction of the predicted high-reuse blocks in the cache and inserts all predicted low-reuse blocks with low priority. When too few blocks are inserted with high priority, the cache suffers from under-utilization. In this case, DIT-SM inserts a fraction of the predicted low-reuse blocks with high priority as well in addition to inserting all predicted high-reuse blocks with high priority.

The additional hardware needed by DIT-SM are only a few counters and some simple control logic, which are negligible. In multi-core processors, each processor maintains its own sampling sets and optimal ε [31].

6.5 Experimental Methodology

DIT is evaluated using the simulation framework, CMP$im [30], released by the First JILP Workshop on Computer Architecture Competitions [2]. The CMP$im simulation framework models a 4-way out-of-order processor with a 128-entry reorder buffer and a three-level cache hierarchy,

Table 6.2: Configuration of the experimental processor.

L1 I-Cache	32KB, 4-way, Private, 1 cycle
L1 D-Cache	32KB, 8-way, Private, 1 cycle
L2 Cache	256KB, 8-way, Private, 10 cycles
LLC	1MB per-core, 16-way, Shared, 30 cycles
MSHR	32 outstanding misses
Main Memory	32 outstanding requests, 200 cycles

Table 6.3: Categories of the selected SPEC CPU2006 benchmarks used for evaluation.

LRU-friendly	astar bzip2 gromacs wrf zeusmp dealII
Thrashing	cactusADM gcc h264ref hmmer libquantum mcf sphinx3
Insensitive	bwaves milc lbm

shown in Table 6.2. The three-level cache hierarchy is based on an Intel Core i7 system. The L1 and L2 caches use the LRU replacement policy. The LLC is evaluated using BRRIP, SHiP, and EAF with DIT-RD and DIT-SM applied.

The evaluations use sequential and multi-programmed workloads, constructed from 16 applications from the SPEC CPU2006 benchmarks. All the SPEC CPU2006 benchmarks are grouped into three categories - LRU-friendly, thrashing, and insensitive using a single-core processor with 1MB LLC. Thrashing applications are those whose performance increases if only a fraction of the working set is kept in the cache. LRU-fiendly applications are those whose performance degrades if only a fraction of the working set is kept in the cache. Insensitive applications are those whose performance almost does not change when the fraction of the working set kept in the cache is varied. All the thrashing applications, six LRU-friendly applications, and three insensitive applications are selected, shown in Table 6.3.

A shared LLC with four cores is evaluated using 100 multi-programmed workloads constructed from these 16 applications. The 100 multi-programmed workloads are grouped into four categories: homogeneous, LRU-friendly, thrashing, and mix. Homogeneous workloads consist of four identical applications. Thrashing/LRU-friendly workloads consist of only thrashing/LRU-friendly applications. Mix workloads consist of a mixture of thrashing, LRU-friendly, and insensitive applications. Traces are collected using PinPoints [53]. SimPoint [67] is used to select one simulation point for each application. Each application is run for 2 billion instructions. The first 1 billion instructions

are used to warm up the cache, and statistics are collected for the second 1 billion instructions. For multi-programmed workloads, if the end of the trace is reached, the simulator rewinds the trace until the slowest thread finishes execution.

6.6 Results

6.6.1 Single-Core Results

Figure 6.9 shows the single-threaded performance, instructions per cycle (IPC), when applying DIT-RD and DIT-SM to BRRIP, SHiP, and EAF. *cactusADM*, *hmmer*, *mcf*, and *sphinx3* have significant thrashing. On average, DIT-RD/DIT-SM improves the performance of thrashing applications by 3%/3%, 6%/6%, and 7%/7%, respectively, over DRRIP, SHiP, and DEAF. For *gcc* and *zeusmp*, SHiP inserts too few blocks with high priority and causes cache under-utilization. DIT-RD/DIT-SM is able to compensate that by keeping a fraction of the predicted low-reuse blocks in the cache as well and improves performance over SHiP by 18%/15% and 14%/14%, respectively, for *gcc* and *zeusmp*. For the majority of the other LRU-friendly and insensitive applications, DIT-RD/DIT-SM does not hurt performance and can improve performance for some of them. When all applications are considered, DIT-RD/DIT-SM improves performance by 1%/1%, 3%/2%, and 2%/2%, respectively, over DRRIP, SHiP, and DEAF on average.

6.6.2 Multi-Core Results

Figure 6.10 shows the average multi-programmed performance, throughput, when applying DIT-RD and DIT-SM to BRRIP, SHiP, and EAF. DIT is most effective for thrashing workloads and mix workloads that consist of LRU-friendly, thrashing, and insensitive applications. On average, DIT-RD/DIT-SM improves the throughput of thrashing workloads by 6%/7%, 4%/7%, and 5%/6%, respectively, and the throughput of mix workloads by 2%/1%/, 5%/3%, and 5%/5%, respectively, over DRRIP, SHiP, and DEAF. For mix workloads, DIT is able to significantly reduce the cache occupancy of insensitive applications that do not benefit from larger cache space, leaving more space for LRU-friendly and thrashing applications whose performance benefits from larger cache.

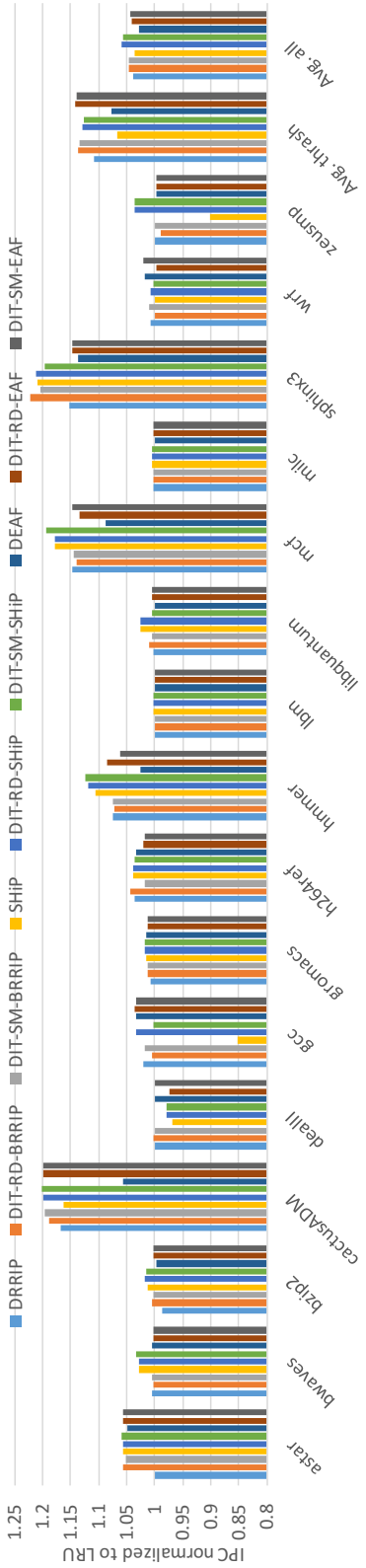


Figure 6.9: Normalized IPC to LRU for single-threaded applications.

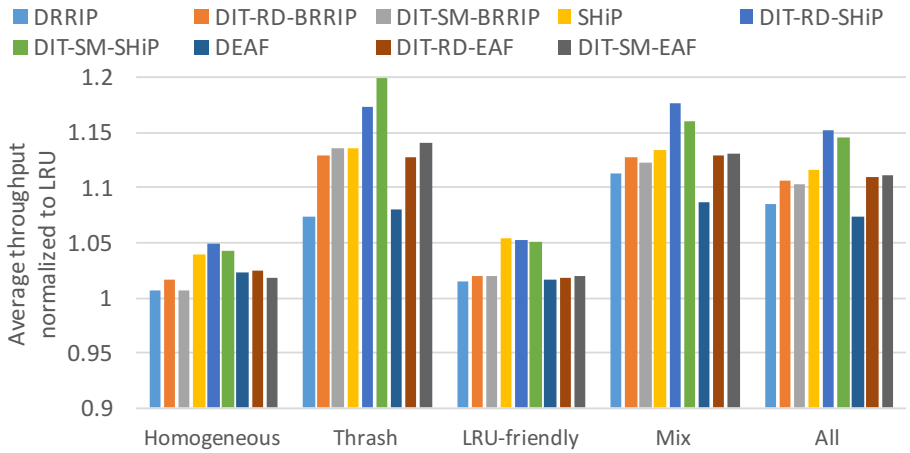


Figure 6.10: Average throughput normalized to LRU for 100 multi-programmed workloads, grouped by categories.

On average, DIT does not hurt the performance of LRU-friendly workloads and can improve the performance of homogeneous workloads. When all the multi-programmed workloads are considered together, DIT-RD/DIT-SM improves throughput by 3%/2%, 4%/3%, and 3%/4%, respectively, over DRRIP, SHiP, and DEAF on average. The performance improvements over DRRIP, SHiP, and DEAF for all the 100 workloads are shown in Figure 6.11. DIT is able to improve performance for roughly 75% of the 100 workloads. For the majority of the rest 25% of the workloads, performance degradation is less than 5%.

Besides throughput, weighted speedup (WS) [31, 69] and harmonic mean fairness (HMF) [31, 48] are also evaluated for multi-programmed workloads. Table 6.4 shows the improvements of WS and HMF by DIT. We conclude that DIT improves both performance and fairness compared to DRRIP, SHiP, and DEAF.

6.6.3 Validating the Analytical Models

EBM and RDM are validated by comparing the ε they obtain dynamically with the ε obtained by static profiling. BRRIP and EAF are selected to validate EBM and RDM, respectively. The dynamic ε for the models are obtained by running DIT-RD-BRRIP and DIT-RD-EAF. The static profiling simulates BRRIP and EAF multiple times with different ε parameters in Table 6.1. In each simulation, a single ε is used throughout the entire run. The ε that maximizes performance is

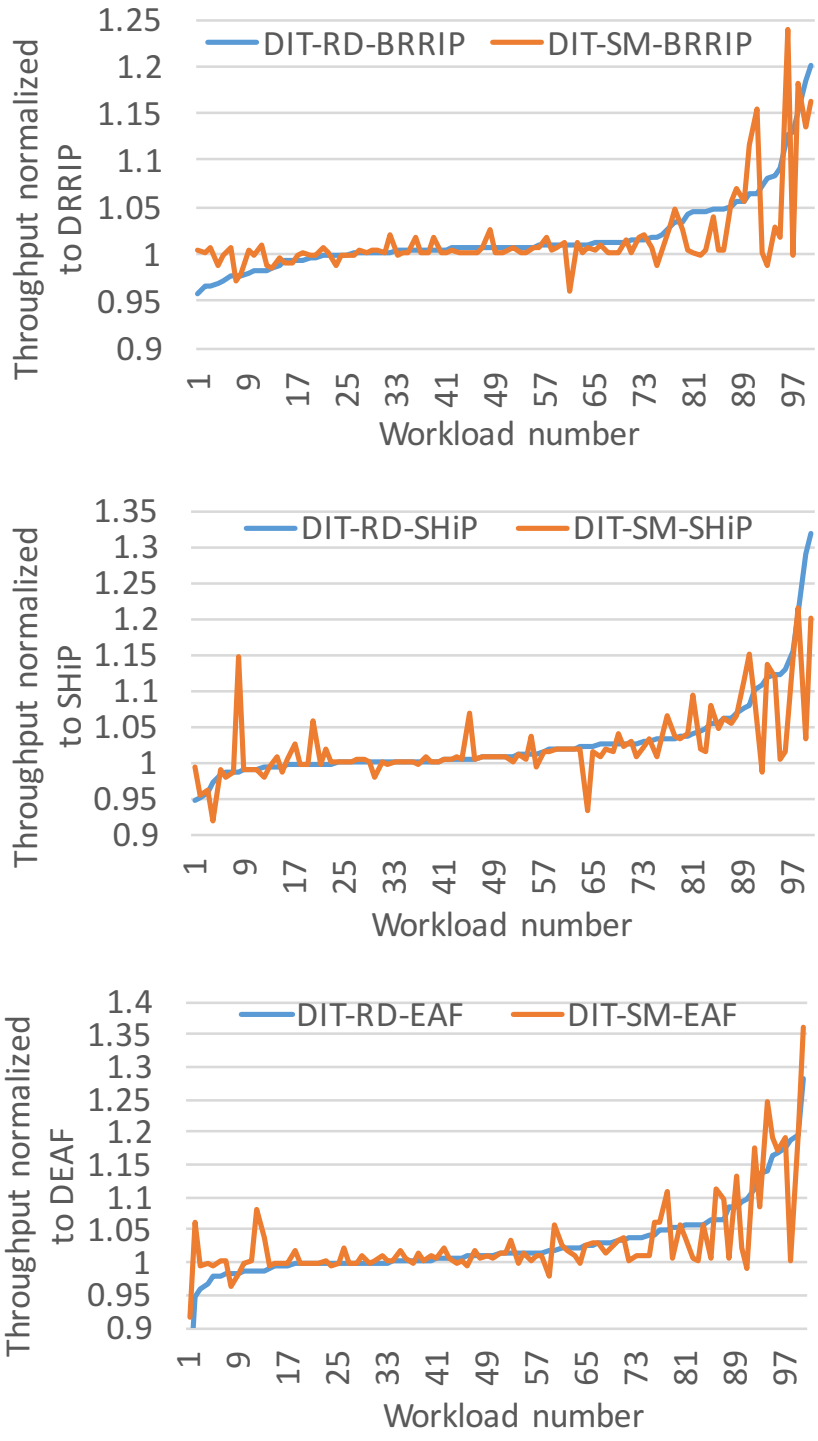


Figure 6.11: Normalized throughput of DIT-RD and DIT-SM for 100 multi-programmed workloads.

Table 6.4: Average percentage improvements (DIT-RD/SM-BRRIP over DRRIP, DIT-RD/SM-SHiP over SHiP, and DIT-RD/SM-EAF over DEAF) of WS and HMF by DIT for thrashing workloads and all the 100 multi-programmed workloads.

	WS		HMF	
Policies	**Thrash**	**All**	**Thrash**	**All**
DIT-RD-BRRIP	5%	2%	4%	2%
DIT-SM-BRRIP	4%	2%	4%	2%
DIT-RD-SHiP	5%	4%	3%	2%
DIT-SM-SHiP	6%	3%	4%	3%
DIT-RD-EAF	3%	3%	2%	3%
DIT-SM-EAF	3%	3%	3%	3%

selected as an approximation of the optimal ε.

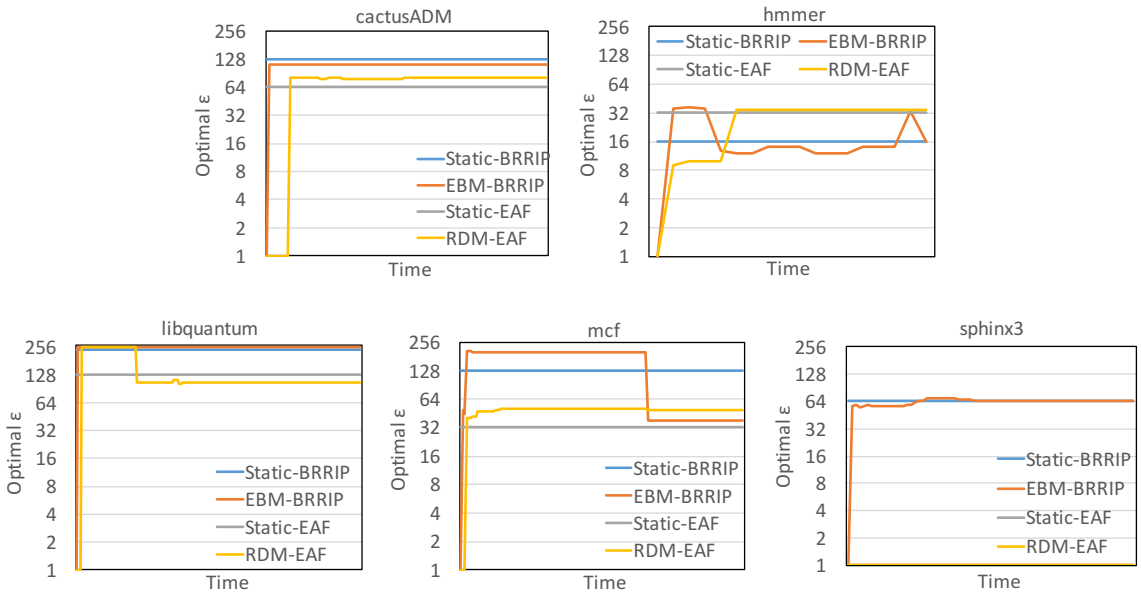


Figure 6.12: Comparison of the best ε obtained by static profiling and by the analytical models. EBM is applied to the BRRIP insertion policy and RDM is applied to the EAF insertion policy.

Figure 6.12 shows the comparison of the dynamic ε obtained via the models and the static ε obtained via profiling for five selected thrashing workloads. EAF does not cause cache under-utilization for these workloads. The dynamic ε obtained by the models changes with time as program executes. ε is initialized to 1 at the beginning of program execution. The results show that DIT is able to adapt to program behavior changes by changing ε, such as *hmmer*, *libquantum*, and *mcf*, whereas, BRRIP and EAF always use a fixed ε. Generally, the dynamic ε obtained via the models matches the ε obtained via static profiling. Some big gaps between the models and static

Table 6.5: The optimal ε obtained via static profiling using BRRIP for different LLC sizes with cache associativity fixed at 16.

	Optimal ε			
Benchmarks	**0.5MB**	**1MB**	**2MB**	**4MB**
astar	8	2	1	1
cactusADM	256	256	256	4
gcc	16	4	4	8
gromacs	2	1	1	1
h264ref	32	8	1	1
hmmer	32	16	4	128
mcf	64	64	32	16
sphinx3	128	128	32	2

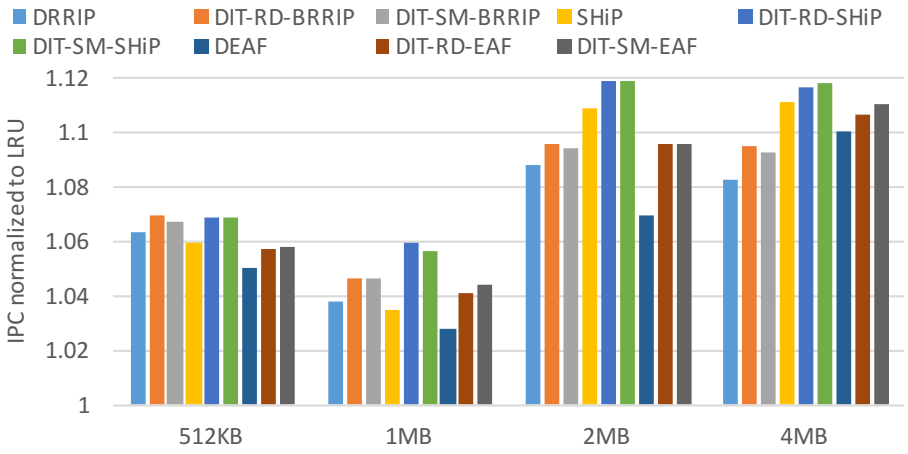
profiling are due to static profiling's inability to adapt to program behavior changes.

6.6.4 Sensitivity to Cache Size

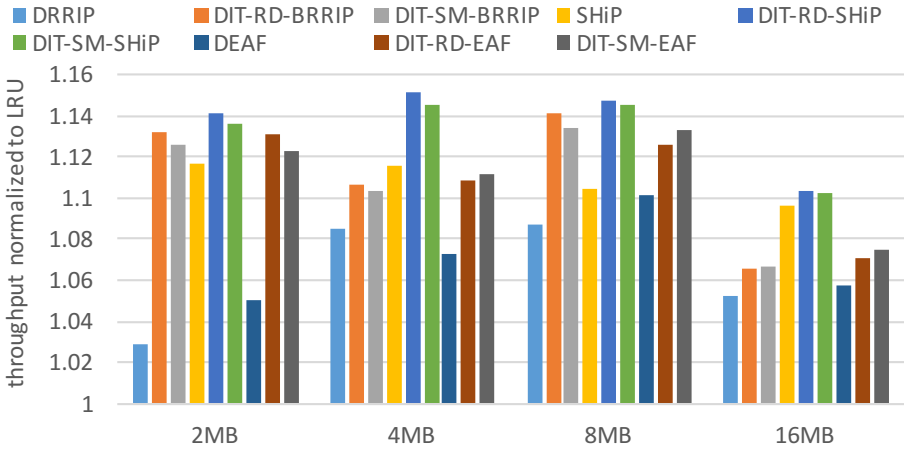
To study the sensitivity of the optimal fraction to cache size, the LLC size is varied from 512KB/core to 4MB/core with the associativity fixed at 16. The optimal fractions for these LLC sizes are determined by simulating the BRRIP insertion policy with all the ε values shown in Table 6.1. The optimal fraction is given by the ε that performs best. The study was done for single-threaded applications only.

Table 6.5 shows the optimal ε for a number of selected applications under different LLC sizes. As predicted by the EBM model, larger cache suffers less from thrashing and prefers a larger fraction of the working set to be kept in the cache, which is validated by the simulation results. Generally, as the LLC size increases, many thrashing workloads become less thrashing or LRU-friendly and the optimal ε decreases, meaning that more blocks are kept in the cache.

Figure 6.13 shows the performance of DIT under different LLC sizes. DIT improves performance over prior insertion policies for all four LLC sizes. DIT is more effective for smaller caches that suffer more from thrashing. As the LLC size increases to 4MB/core, thrashing is significantly reduced and the improvements by DIT also become smaller.



(a) single-core



(b) 4-core

Figure 6.13: Average performance over LRU under different LLC sizes.

Table 6.6: The optimal ε obtained via static profiling using BRRIP for different LLC associativities with LLC size fixed at 1MB/core.

	Optimal ε		
Benchmarks	**8-way**	**16-way**	**32-way**
astar	2	2	2
cactusADM	256	256	192
gcc	4	4	4
h264ref	8	8	8
hmmer	16	16	16
mcf	64	64	64
sphinx3	128	128	128

6.6.5 Sensitivity to Cache Associativity

To study the sensitivity of the optimal fraction to cache associativity, the associativity of the LLC is varied from 8 to 32 with the size fixed at 1MB/core. The optimal fractions for these associativities are determined by simulating the BRRIP insertion policy with all the ε values shown in Table 6.1. Table 6.6 shows the optimal ε for a number of selected applications under different LLC associativities. As predicted by the EBM model, the optimal ε is insensitive to the changes of associativity, which is validated by the simulations. For most workloads, the optimal ε stays almost unchanged as associativity varies, except for *cactusADM*, whose optimal ε decreases when the associativity reaches 32. The reason is that *cactusADM* suffers more from conflict misses, which becomes less severe when the LLC associativity reaches 32. Thus, the workload becomes less thrashing and the optimal ε decreases.

6.6.6 Interaction with Prefetching

Hardware prefetching is commonly used to improve cache performance. To study the interaction of DIT with prefetching, the prefetcher in the CMP$im simulator is turned on. According to [31], CMP$im uses a per-core stream prefetcher similar to [72]. Figure 6.14 shows the average performance of all the single-threaded and multi-programmed workloads under the different insertion policies when the prefetcher is turned on. The results show that both DIT-RD and DIT-SM are still effective in the presence of prefetching. On average, DIT improves single-threaded performance by up to 1% across all the 16 selected SPEC CPU2006 applications, and multi-programmed perfor-

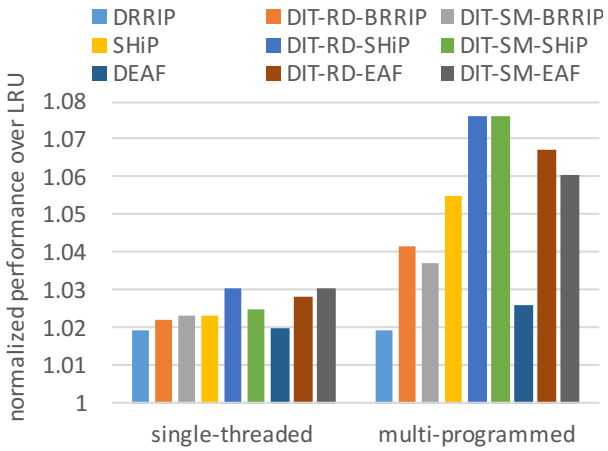


Figure 6.14: Average results when prefetch is truned on.

mance by up to 4% across all the 100 multi-programmed workloads, over prior insertion policies - DRRIP, SHiP, and DEAF.

Chapter 7

Conclusions and Future Work

7.1 Dissertation Summary

This dissertation identifies the problem that the datapath and cache resources in modern superscalar processors are inefficiently utilized, leading to inferior performance and energy efficiency. The one-size-fits-all design approach employed by the traditional superscalar processors allocates a fixed amount of resources for all applications at all times to deliver the best overall performance, which is not always energy efficient, because both the application behavior and the use scenario are changing all the time and the demand for processor resources is also changing accordingly. Caches in modern processors also suffer from inefficient utilization. In set-associative L1 caches, the non-matching data ways are accessed unnecessarily, which causes energy wastes. For memory intensive workloads, caches often suffer from thrashing, but prior insertion policies that manages the LLC took ad hoc approaches and failed to fully address this problem, limiting the performance impact.

This dissertation aims to improve processor performance and energy efficiency via more efficient utilization of datapath and cache resources. It demonstrates that dynamically allocating datapath resources based on application needs and use scenarios significantly improves processor energy efficiency, and that managing cache resource utilization via more efficient methods can lead to dramatic energy savings and performance improvement. It proposes an adaptive processor that dynamically changes the active datapath resources according to application behavior and use scenarios to improve energy efficiency. When applied to front-end throttling, the adaptive proces-

sor achieves average improvements of 28%, 28%, and 32% for energy, energy-delay product, and energy-delay-squared product, respectively, over all selected benchmarks on an 8-way superscalar processor. When applied to dynamic core scaling, the adaptive processor saves an additional 20% of a full-size cores energy on average and achieves an average of 46% further energy reduction at the minimum operating voltage. It also proposes the ETL and DIT techniques to reduce cache power consumption and improve cache performance. ETL is able to remove the majority of non-matching data way accesses and reduce read energy by 68% on average on a 4-way set-associative L1 instruction cache. DIT improves performance of prior state-of-the-art insertion policies, DRRIP, SHiP, and DEAF, by 7%, 7%, and 6%, respectively, for thrashing workloads, and by 3%, 4%, and 4%, respectively, for 100 mixed workloads in a 4-core configuration. All these proposed techniques are orthogonal to each other and can be applied together. The contributions of these proposed techniques are summarized as follows.

Adaptive front-end throttling. This technique dynamically adjusts the front-end instruction delivery bandwidth using software profiling or a runtime hardware controller to optimize arbitrary target metrics, being performance, energy, or any trade-offs between them, as user scenario and program behavior change. Adaptive front-end throttling is orthogonal to, and can even leverage, most existing techniques, providing even greater savings. The new architecture is implemented at the register transfer level (RTL), and circuit-level synthesis and simulation are used to accurately analyze the area, delay, and power overhead of the throttling technique and resulting energy savings. Evaluation results show that this technique incurs negligible overhead and provides significant energy savings.

Dynamic core scaling. Instead of scaling voltage, DCS dynamically adjusts the active superscalar datapath resources and tries to ensure that programs run at a given percentage of their maximum speed while minimizing energy consumption. A hardware controller is proposed to effectively manage performance-energy trade-off using DCS. Since DCS does not rely on voltage scaling, it can be combined with DVFS to achieve greater energy savings. To effectively manage performance-energy trade-off using a combination of DCS and DVFS, an oracle controller is proposed to demonstrate the optimal control strategy, and two practical controllers are proposed

for real implementations. The proposed hardware controllers effectively extend the performance-energy trade-off capabilities in superscalar processors.

Early tag lookup. ETL effectively removes the majority of the accesses to the non-matching data ways in L1 set-associative instruction caches, achieving significant read energy reduction comparable to the phased caches. ETL does not cause any performance penalty, and incurs insignificant hardware overhead and low design complexity. Narrower fetch width, larger BTB, better branch predictors, and memory dependence prediction help enhance the effectiveness of ETL.

Dynamic insertion throttling. This work demonstrates that maximizing performance for cache polluting and thrashing workloads requires an *optimal* fraction of the working set to be in the cache. Prior insertion policies cannot maximize performance as they determine this fraction empirically. This work provided a theoretical and practical analysis of insertion policies and determined that by inserting only a fraction of the blocks with a high priority, these policies effectively reduce the RD of blocks in the cache. It was also determined that these blocks receive hits *only if the ERD becomes smaller than the cache associativity*. Using this key insight, an analytical model is derived to show that it is possible to determine the optimal RD such that the fraction of blocks with ERD equal to or less than the cache associativity can be maximized. Two practical models applicable to recent policies when the oracle RD information is not available are provided and two simple mechanisms to determine the optimal fraction based on these models are proposed. The proposed mechanisms significantly improve performance compared to recent insertion policies over a wide range of applications and system configurations.

7.2 Future Work

The future work that are related to this dissertation are summarized as follows.

Adaptive processor. In DCS, both a simple hardware controller and a sophisticated one are proposed to dynamically manage resource allocation for performance-energy trade-off. Although the proposed hardware controllers are efficient, a small gap still exists between DCS-controller and DCS-potential. To extend this work in the future, more effective control algorithms can be

developed to further reduce this gap or even outperform DCS-potential.

Early tag lookup. The proposed ETL technique uses a *gselect* branch predictor. Commercial processors usually use more sophisticated branch predictors, which are usually pipelined and take multiple cycles to make one prediction. Future work can investigate how to apply the existing ETL technique with more sophisticated branch predictors and pipelined instruction fetch.

LLC management. Applications running on different cores usually have different demands on the shared LLC resource. In the traditional LLC, cache space is allocated based on demands. However, applications could issue many cache accesses but do not benefit from more cache space, for example, some applications may have scanning accesses that evict useful blocks. Future work can investigate cache management policies that allocate the LLC resources based on whether an application benefits from larger cache space rather than demands.

7.3 Related Publications

During my PhD study, I have authored and co-authored the following publications.

Wei Zhang, Hang Zhang, and John Lach. "Extending Performance-Energy Trade-off via Dynamic Core Scaling". In submission.

Wei Zhang, Samira Khan, and John Lach. "DIT-Cache: Understanding, Modeling, and Exploiting the Theory behind Insertion Policies to Improve Performance". In submission.

Wei Zhang, Hang Zhang, and John Lach. "Reducing Dynamic Energy of Set-Associative L1 Instruction Cache by Early Tag Lookup". In International Symposium on Low Power Electronics and Design, 2015. (**Best paper nominee.**)

Wei Zhang, Hang Zhang, and John Lach. "Dynamic Core Scaling: Trading Off Performance and Energy Beyond DVFS". In International Conference on Computer Design, 2015.

Wei Zhang, Hang Zhang, and John Lach. "Adaptive Front-End Throttling for Superscalar Processors". In International Symposium on Low Power Electronics and Design, 2014.

Hang Zhang, **Wei Zhang**, and John Lach. "A Low-Power Accuracy-Configurable Floating Point Multiplier", In International Conference on Computer Design, 2014.

Kenneth M. Zick, Meeta Srivastav, **Wei Zhang**, and Matthew French. "Sensing Nanosecond-Scale Voltage Attacks and Natural Transients in FPGAs". In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2013.

Bibliography

[1] HP labs. CACTI. http://www.hpl.hp.com/research/cacti/.

[2] Jilp workshop on computer architecture competitions. http://jilp.org/jwac-1/.

[3] Enhanced intel speedstep technology for the intel pentium m processor. `ftp://download.intel.com/design/network/papers/30117401.pdf`, March 2004.

[4] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. The reuse cache: Downsizing the shared last-level cache. In *MICRO*, 2013.

[5] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Archit. Code Optim.*, 9(4):38:1–38:19, January 2013.

[6] Juan L. Aragón, José González, and Antonio González. Power-aware control speculation through selective throttling. In *HPCA*, 2003.

[7] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *ISCA*, 2010.

[8] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *ISCA*, 2001.

[9] Amirali Baniasadi and Andreas Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *ISLPED*, 2001.

[10] A. Bardizbanyan, M. Sjalander, D. Whalley, and P. Larsson-Edefors. Speculative tag access for reduced energy dissipation in set-associative l1 data caches. In *ICCD*, 2013.

[11] Alen Bardizbanyan, Magnus Själander, David Whalley, and Per Larsson-Edefors. Reducing set-associative l1 data cache energy by early load data dependence detection (eld3). In *DATE*, 2014.

[12] Brannon Batson and T. N. Vijaykumar. Reactive-associative caches. In *PACT*, 2001.

[13] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[14] Alper Buyuktosunoglu, Tejas Karkhanis, David H. Albonesi, and Pradip Bose. Energy efficient co-adaptive instruction fetch and issue. In *ISCA*, 2003.

[15] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *HPCA*, 1996.

[16] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *PACT*, 2012.

[17] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *ISCA*, 2011.

[18] Jamison D. Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In *MICRO*, 1999.

[19] Jianwei Dai, Menglong Guan, and Lei Wang. Exploiting early tag access for reducing l1 data cache energy in embedded processors. *IEEE Trans. VLSI*, 2014.

[20] Subhasis Das, Tor M. Aamodt, and William J. Dally. Reuse distance-based probabilistic cache replacement. *ACM Trans. Archit. Code Optim.*, 12(4):33:1–33:22, October 2015.

[21] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, Oct 1974.

[22] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *MICRO*, 2012.

[23] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *ISCA*, 2001.

[24] N.B.M. Hajj, C. Polyckronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *ISLPED*, 1998.

[25] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *ISCA*, 2000.

[26] Houman Homayoun, Avesta Sasan, Jean-Luc Gaudiot, and Alex Veidenbaum. Reducing power in all major cam and sram-based processor units via centralized, dynamic resource size management. *IEEE Trans. VLSI*, 2011.

[27] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED*, 1999.

[28] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *ISCA*, 2007.

[29] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *ISCA*, 2002.

[30] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. of the 4th Workshop on Modeling, Benchmarking and Simulation*, 2008.

[31] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *PACT*, 2008.

[32] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ISCA*, 2010.

[33] Daniel A. Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *MICRO*, 2013.

[34] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, December 1999.

[35] Timothy Jones, Michael O'Boyle, Jaume Abella, and Antonio Gonzalez. Software directed issue queue power reduction. In *HPCA*, 2005.

[36] Tejas Karkhanis, James E. Smith, and Pradip Bose. Saving energy with just in time instruction delivery. In *ISLPED*, 2002.

[37] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, 2007.

[38] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. In *PACT*, 2010.

[39] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *MICRO*, 2010.

[40] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *MICRO*, 2012.

[41] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *MICRO*, 1997.

[42] Jinson Koppanalil, Prakash Ramrakhyani, Sameer Desai, Anu Vaidyanathan, and Eric Rotenberg. A case for dynamic pipeline scaling. In *CASES*, 2002.

[43] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, 2001.

[44] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. Optimal bypass monitor for high performance last-level caches. In *PACT*, 2012.

[45] Sheng Li, Jung Ho Ahn, Richard Strong, Jay Brockman, Dean Tullsen, and Norman Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and many-core architectures. In *MICRO*, 2009.

[46] Bin Lin, Arindam Mallik, Peter Dinda, Gokhan Memik, and Robert Dick. User- and process-driven dynamic voltage and frequency scaling. In *ISPASS*, 2009.

[47] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO*, 2008.

[48] Kun Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In *ISPASS*, 2001.

[49] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *ISCA*, 1998.

[50] Nikil Mehta, Brian Singer, Iris Bahar, Michael Leuchten, and Richard Weiss. Fetch halting on critical load misses. In *ICCD*, 2004.

[51] Amir Nahir, Manoj Dusanapudi, Shakti Kapoor, Kevin Reick, Wolfgang Roesner, Klaus-Dieter Schubert, Keith Sharp, and Greg Wetli. Post-silicon validation of the ibm power8 processor. In *DAC*, 2014.

[52] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. Reducing power consumption for high-associativity data caches in embedded processors. In *DATE*, 2003.

[53] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *MICRO*, 2004.

[54] Pavlos Petoumenos, Georgia Psychou, Stefanos Kaxiras, Juan Manuel Cebrian Gonzalez, and Juan Luis Aragon. MLP-aware instruction queue resizing: The key to power-efficient performance. In *ARCS*, 2010.

[55] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*, 2013.

[56] Thomas Piquet, Olivier Rochecouste, and André Seznec. Exploiting single-usage for effective memory management. In *ACSAC*, 2007.

[57] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Dynamic resizing of superscalar datapath components for energy efficiency. *IEEE Trans. Comput.*, 2006.

[58] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *MICRO*, 2001.

[59] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[60] Kaushik Rajan and Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO*, 2007.

[61] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 1, pages 154–163 vol.1, Aug 1996.

[62] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro*, 2012.

[63] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *MICRO*, 2010.

[64] Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. Tlc: A tag-less cache for reducing dynamic first level cache energy. In *MICRO*, 2013.

[65] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO*, 2002.

[66] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

[67] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[68] Hajime Shimada, Hideki Ando, and Toshio Shimada. Pipeline stage unification: A low-energy consumption technique for future mobile processors. In *ISLPED*, 2003.

[69] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[70] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. FreePDK: An open-source variation-aware design kit. In *MSE*, 2007.

[71] Weiyu Tang, Alexander V. Veidenbaum, Alexandru Nicolau, and Rajesh Gupta. Integrated i-cache way predictor and branch target buffer to reduce energy consumption. In *HPC*, 2002.

[72] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM J. Res. Dev.*, 46(1):5–25, January 2002.

[73] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *MICRO*, 1995.

[74] Osman S. Unsal, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-fetch: Compiler-enabled power-aware fetch throttling. In *IEEE CAL*, 2002.

[75] Huaping Wang, Yao Guo, I. Koren, and C.M. Krishna. Compiler-based adaptive fetch throttling for energy-efficiency. In *ISPASS*, 2006.

[76] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *MICRO*, 2011.

[77] Chia-Lin Yang and Chien-Hao Lee. Hotspot cache: Joint temporal and spatial locality exploitation for i-cache energy reduction. In *ISLPED*, 2004.

[78] Tadashi Yasufuku, Satoshi Iida, Hiroshi Fuketa, Koji Hirairi, Masahiro Nomura, Makoto Takamiya, and Takayasu Sakurai. Investigation of determinant factors of minimum operating voltage of logic gates in 65-nm cmos. In *ISLPED*, 2011.

[79] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *ICS*, 1993.

[80] Chuanjun Zhang, Frank Vahid, Jun Yang, and Walid Najjar. A way-halting cache for low-energy high-performance systems. *ACM Trans. Archit. Code Optim.*, 2005.

[81] Wei Zhang, Hang Zhang, and John Lach. Adaptive front-end throttling for superscalar processors. In *ISLPED*, 2014.

[82] Wei Zhang, Hang Zhang, and John Lach. Dynamic core scaling: Trading off performance and energy beyond dvfs. In *ICCD*, 2015.

[83] Wei Zhang, Hang Zhang, and John Lach. Reducing dynamic energy of set-associative l1 instruction cache by early tag lookup. In *ISLPED*, 2015.

[84] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Tag check elision. In *ISLPED*, 2014.

Acronym List

The acronyms used in this dissertation are collected here for easy reference.

ALU Arithmetic Logic Unit

BIP Bimodal Insertion Policy

BP Branch Predictor

BRRIP Bimodal Re-Reference Interval Prediction

BTB Branch Target Buffer

CAM Content Addressable Memory

CPI Cycles Per Instruction

DCS Dynamic Core Scaling

DIT Dynamic Insertion Throttling

DVFS Dynamic Voltage and Frequency Scaling

EAF Evicted Address Filter

EBM Equal Block Model

ED Energy Delay

ED^2 Energy Delay Square

ERD Effective Reuse Distance

ETL Early Tag Lookup

FET Front-End Throttling

HMF Harmonic Mean Fairness

ILP Instruction-Level Parallelism

IPC Instructions Per Cycle

IQ Issue Queue

LLC Last Level Cache

L1 Level one

L2 Level two

LRU Least Recently Used

LSQ Load Store Queue

MRU Most Recently Used

NNPC Next Next Program Counter

NPC Next Program Counter

ORM Oracle Reuse Model

PC Program Counter

PHT Pattern History Table

PSEL Policy Selector

RAM Random Access Memory

RD Reuse Distance

RDD Reuse Distance Distribution

RDM Reuse Differentiating Model

ROB Reorder Buffer

RTL Register Transfer Level

SHiP Signature-based Hit Predictor

SRAM Static Random Access Memory

TLB Translation Lookaside Buffer

WS Weighted Speedup