Analysis and Optimizations for Modern Processors' Branch Target Buffer and Cache Memory

by

Kaveh Jokar Deris
M.Sc., Iran University of Science and Technology, 2003
B.Sc., Amirkabir University of Technology (Tehran Polytechnic), 2001

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Kaveh Jokar Deris, 2008
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Branch Target Buffer and Cache Memory Analysis and Optimizations for Modern Processors

by

Kaveh Jokar Deris
M.Sc., Iran University of Sci. and Tech., 2003
B.Sc., Amirkabir University (Tehran Polytechnic), 2001

Supervisory Committee

Dr. Amirali Baniasadi, Supervisior
Department of Electrical and Computer Engineering

Dr. Nikitas J. Dimopoulos, Departmental Member
Department of Electrical and Computer Engineering

Dr. Mihai Sima , Departmental Member
Department of Electrical and Computer Engineering

Dr. Jianping Pan, Outside Member
Department of Computer Science

Dr. Tor M. Aamodt, External Member
Department of Electrical and Computer Engineering
University of British Columbia

Supervisory Committee
Dr. Amirali Baniasadi, Department of Electrical and Computer Engineering
Supervisor

Dr. Nikitas J. Dimopoulos, Department of Electrical and Computer Engineering
Departmental Member

Dr. Mihai Sima, Department of Electrical and Computer Engineering
Departmental Member

Dr. Jianping Pan, Department of Computer Science
Outside Member

Dr. Tor M. Aamodt, Department of Electrical and Computer Engineering, UBC
External Member

Abstract

Microprocessor architecture has changed significantly since Intel Corporation developed the first commercial computer chip in the 1970s. The modern processors are much smaller and more powerful than their predecessors. Yet, in the mobile computing era the market demands for smaller, faster, cooler, and more power-efficient CPUs that could provide greater performance-per-watt results.

In this dissertation, we address some of the shortcomings in conventional microprocessor designs and discuss possible means of alleviating them. First, we investigate the energy dissipation in Branch Target Buffer (BTB), a commonly present component in branch prediction unit. Our primary contribution is a speculative allocation technique to improve BTB energy consumption. In this technique, a new on-chip structure predicts the BTB activity and dynamically eliminates unnecessary accesses.

Next, we formulate a quantitative metric to analyze the trade-off between processor energy efficiency and cache energy consumption. We investigate the upper

bound energy and latency budget available for alternative data and instruction cache enhancements.

This dissertation concludes with a novel approach to increase processors' performance by reducing data cache miss rate. We employ a speculative technique to bridge the performance gap between the common Least Recently Used (LRU) replacement algorithm and the optimal replacement policy. We evaluate the non-optimal decisions made by the LRU algorithm and provide a taxonomy of mistakes, which will aid to identify and avoid similar decisions in future incidents.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables vii

List of Figures viii

1 Introduction 1

2 Background and Related Research Work 6

2.1 Introduction ... 6
2.2 Cache Memory Essentials ... 7
2.3 Cache Enhancements for Miss-rate and Miss-penalty Reduction ... 11
2.4 Branch Prediction Essentials ... 17
2.5 Branch Predictor Energy and Performance Improvements ... 22

3 Design and Analysis of an Energy-aware Branch Target Buffer 26

3.1 Introduction ... 27
3.2 Speculative BTB Allocation in Embedded Processors ... 28
3.3 Speculative BTB Allocation in High-Performance Processors ... 39

4 Cache Complexity Analysis for Modern Processors 47

4.1 Introduction ... 48
4.2 Energy Budget Formulation ... 49
4.3 High-Performance Processors ... 53
4.4 Embedded Processors ... 67

5 Reducing Non-optimal LRU Decisions in Chip Multiprocessors **74**

5.1 Introduction ... 75
5.2 H-Blocks ... 77
5.3 P-Blocks ... 83
5.4 Avoiding LRU Non-optimal Decisions ... 85
5.5 Methodology ... 89
5.6 Results ... 90
5.7 History Table Implementation ... 98
5.8 Cache miss rate reduction with MLHT ... 102

6 Conclusions **104**

6.1 Summary of contributions ... 104
6.2 Future work ... 106

Bibliography **108**

A Simulation Tools **113**

A.1 SimpleScalar Simulator ... 113
A.2 Wattch Simulator ... 114
A.3 SESC: SuperESCalar Simulator ... 117
A.4 Cacti Tool Set ... 117

B Benchmarks **119**

B.1 SPEC CPU2000 Benchmarks ... 119
B.2 MiBench Embedded Benchmark ... 121
B.3 SPLASH-2 multi-processor Benchmark ... 122

C Dynamic Branch Predictors **123**

C.1 Branch Predictors' Structure ... 123

List of Tables

3.1 The subset of MiBench benchmarks studied and their BLC frequency 30
3.2 Simulated processor configuration 34
3.3 Energy consumed per access by the branch predictor units and the BLC-filter. 35
3.4 Energy consumed per access for branch predictor units and the BLC-filter. 41
4.1 Simulated processor configurations 53
4.2 Cache organizations and their relative size 54
4.3 Processor configuration used in this study 67
5.1 Tagged Blocks in the cache 86
5.2 Configuration of each core in our CMP model 89
5.3 Suite of SPLASH-2 Benchmark used 90
A.1 Common processor hardware units and the type of model used by Wattch 116
A.2 Comparison between modeled and reported power breakdown for the Pentium Pro® 116
A.3 Comparison between modeled and reported power breakdown for the Alpha 21264 116
B.1 SPEC 2000 Integer Benchmarks 120
B.2 SPEC 2000 Floating Point Benchmarks 120
B.3 MiBench Benchmarks 121
B.4 SPLASH-2 Benchmarks 122

List of Figures

2.1 The three portions of an address in set-associative and direct-mapped caches. 9
2.2 Values inside each state indicate that state's saturating counter value following the direction prediction output. Arrows shows the transactions to the next states after the branch outcomes are resolved. 20
3.1 BTB's energy consumption share in the branch predictor unit. 28
3.2 Total processor energy per access breakdown. Branch Predictor consumes 5-10% of total CPU energy. 29
3.3 BLC interval frequency 31
3.4 The BLC-Filter architecture. 32
3.5 (a) Branchless cycle predictor lookup. (b) Branchless cycle predictor update. 33
3.6 Average accuracy & coverage achieved for different BLC-filter configurations and for the MiBench benchmarks studied here. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters. 37
3.7 Average performance slowdown and Processor total energy reduction for different BLC-filter configurations. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters. 38
3.8 Average energy delay squared product and BTB energy reduction for different BLC-filter configurations. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters. 38
3.9 BLC frequency for different applications and for different execution bandwidths 40

3.10 BLC filter inserted in fetch stage 40
3.11 Average accuracy and coverage achieved for different BLC-filter configurations 42
3.12 Average BTB energy reduction and performance loss for different BLC-filter configurations 42
3.13 Average BTB energy reduction and performance loss for different branch predictor and BTB size. 44
3.14 Average BLC-Filter accuracy and covergae for different branch predictor and BTB size. 45
3.15 The BTB energy reduction for processors with different execution bandwidths. 45
4.1 Remainder energy over delay prediction for alternative (a) data cache and (b) instruction cache configurations using equation (4.6). 54
4.2 Energy budgets for alternative data cache configurations using the ED^2 metric. 56
4.3 Relative data cache run time energy consumption compared to the energy budget per application run time for each application and cache organization. (The 100% line shows the energy budget limit) 57
4.4 Average percentage of L1 data cache run time energy consumption compared to the energy budget per application run time for each processor. (The 100% line shows the energy-budget limit.) 58
4.5 Total processor energy consumption using reduced sized L1 data cache configurations. 59
4.6 The entire bar reports energy budget available to an ideal data cache for 4-way and 8-way processors. The lower part of each bar shows L1 data cache energy consumption for the reference cache. (Values more than 70 joules are not reported.) 60
4.7 Hit latency impact on performance gap between non-realistic and realistic data caches. 61
4.8 Energy budgets per application run time for alternative instruction cache configurations using the ED^2 metric. 62

4.9 Relative instruction cache run time energy consumption compared to the energy budget per application run time for each application and cache organization. The 100% line shows the energy budget limit. (Values more than 700% are not reported.) 63

4.10 Average percentage of L1 instruction cache run time energy consumption compared to the energy budget per application run time for each processor. The 100% line shows the energy-budget limit (Values more than 800% are not reported.) 64

4.11 Total processor energy consumption using reduced sized L1 instruction cache configurations. 64

4.12 The entire bar reports energy budget per application run time available to an ideal instruction cache for 4-way and 8-way processors. The lower part of each bar shows the L1 instruction cache run time energy consumption for the reference cache. (The values more than 4 (part a) and 3 (part b) joules are not shown) 65

4.13 Hit latency impact on performance gap between non-realistic and realistic instruction caches 66

4.14 Energy budgets for alternative data cache and instruction cache configurations. 70

4.15 Relative data and instruction cache energy consumption compared to the energy budget for each application and cache organization. (The 100% line shows the energy budget limit.) 71

4.16 Average percentage of L1 data and instruction cache energy consumption compared to the energy-budget for selected processor. (The 100% line shows the energy budget limit.) 71

4.17 The entire bar reports energy budget available to an ideal cache. The lower part of each bar shows the cache energy consumption for the reference cache. (Values more than 70 mjoules are not reported.) 72

4.18 Hit latency impact on performance gap between non-realistic and realistic data and instruction caches. 73

5.1 (a) Simple example of LRU replacement policy. Each row shows an access to the imaginary cache set presented in this illustration. (b) Same example illustrated with asterisk notation to save space. 78

5.2 (a) Block ‘A’ is recalled in third row. Due to other blocks access pattern two scenarios can happen in third row; Scenario 1 (above the dashed line): block ‘A’ is not an H-block since other blocks are referenced after ‘A’ evicted; Scenario 2 (below the line): underlined block ‘A’ is a simple example of an H-block since ‘D’ and ‘C’ have not been accessed. (b) H-blocks (unlike live blocks) are not restricted to references immediately after eviction. 79

5.3 AC is reset but RC increments on every replacement 80

5.4 History table and the fields recorded for each evicted block 81

5.5 Our H-Block detection algorithm in simple pseudo code 82

5.6 H-Blocks and L-Blocks comparison 82

5.7 A simple example to explain P-Blocks. We assume ‘A’, ‘B’, ‘C’, ‘D’ and ‘E’ are all mapped to the same set in a 4-way associative data cache. So long the program runs in the WHILE loop, ‘A’, ‘B’, ‘C’ and ‘D’ are loaded calculating T’s initial value. ‘E’ will be loaded in the inner FOR loop and will be referenced exactly 7 times before eventually being replaced by ‘D’........ 83

5.8 (a) LRU policy evicts the blocks on the order of access which results in five misses per while loop iteration (b) Early eviction of block ‘E’ as P-Block results in two misses per each while loop iteration... 84

5.9 Our modeled CMP system 90

5.10 NOD distribution. For each benchmark the left bar represents the LRU replacement policy and the right bar is when the SRA is used with a 16k entries history table. 91

5.11 H-Blocks prediction accuracy and coverage achieved by different history table configurations and for the Splash 2 benchmarks studied here. For each benchmark bars from left to right report for tables of size 512 to 64k entries. 93

5.12 P-Block prediction accuracy and coverage achieved by different history table configurations, and for the Splash 2 benchmarks studied here. For each benchmark bars from left to right report for tables of size 512 to 64k entries. 94

5.13 Decreasing cache block replacement by using the SRA cache management technique. For each benchmark the bars from left to right report for tables of size 512 to 64k entries........ 95
5.14 Average H-Blocks prediction coverage and accuracy for splash-2 benchmarks achieved by different history table configurations. Each line represents a different size of data cache memory per core. 96
5.15 Average H-Blocks miss rate reduction for splash-2 benchmarks achieved by different history table configurations. Each line represents different size of data cache memory per core. 97
5.16 The information stored in history table entries. 98
5.17 Modified version of LRU style history table (MLHT) diagram 100
5.18 H-Block prediction accuracy and coverage achieved by different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries. 101
5.19 P-Block prediction accuracy and coverage achieved by different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries. 102
5.20 Miss rate reduction for different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries. 103
A.1 Simulator Structure 113
A.2 SimpleScalar statistical output file........ 114
A.3 The overall structure of Wattch 115
C.1 Local branch predictors (left) Bimodal branch predictor (right) 2-level local history branch predictor 123
C.2 Global history branch predictor 125
C.3 Gshare Global history predictor........ 125
C.4 Combined branch predictor........ 126

Acknowledgments

My research work would not be complete without the help and support from many people at the University of Victoria. I would like to dedicate this page to thank those who have been most influential along my way.

First and foremost, I would like to thank my supervisor Dr. Amirali Baniasadi. I cannot thank Amirali enough for giving me the opportunity, and providing invaluable advice, guidance and support. Amirali's support throughout the years made me strong enough to confront ups and downs in my PhD program. I would like to thank him for so many things he taught me about research and life.

I would also like to thank my committee for their input and feedbacks on my research work: Professors Nikitas J. Dimopoulos, Mihai Sima, Jianping Pan, Kin F. Li, Sudhakar Ganti and Tor M. Aamodt.

I am so grateful to Dean Dr. Devor for his help and financial support which gave me the second chance when I needed it most to deliver what I started.

I would like to also thank my peers and friends in LAPIS research group. I spent most of my time with them during my research work and we share too many memories to remember. Special thanks go to Farshad Khunjush, Ehsan Atoofian, Scott Miller and Solmaz Khezerloo. Their help, encouragement and in-dept explanations to as many questions as I had for them smooth my path around all sorts of problems.

Finally, I thank my family. I am mostly grateful to my father, Abbas, for giving me the confidence and encouragements to achieve unreachable, and truly support me to continue my studies to the highest level I desire. And to my mother, Mahnaz, whose long fight with cancer taught me how to bear with unwanted problems and never give-up.

Dedicated to:

my parents, Mahnaz and Gholam Abbas,

and my sisters, Tina, Tiam and Tara

Chapter 1

Introduction

Moore's Law, which first appeared in 1965, truly envisioned the future of semiconductor industry, stating that the number of transistors on a chip will double about every two years. Despite many advancements made by industry and academia to keep this trend for many years, the ever increasing demand for faster, smaller and more energy efficient computing devices still challenges the processor designers to overcome many difficulties in this roadmap.

For many years scaling the transistor size has been the driving force behind integrated circuit industry including memory, microprocessors, and graphics processors to deliver higher speed and improve overall performance. The design constraints were dominated by decreasing the feature size to achieve a higher clock speed, dealing with more complex feature sets and finally cutting down the growing thermal envelopes and power dissipation.

More recently, the designers tend to maintain or slightly change the operating clock frequency so that the thermal and power dissipation remain within the limits; however, the demand for increasing performance urges more tasks to be done per clock cycle. This trend moved the processors toward employing more than one core and introduced multi-core processing.

State-of-the-art multi-core processors integrate billions of nano-scale transistors in a limited die area to include more functionality and deliver higher computational power. Such complexity comes with extensive design issues such as transistor density, leakage energy waste, resource sharing, interconnect latency, etc. These open problems require innovative ideas to keep the multi-core processors' performance and energy dissipation on track with semiconductor technology road map [1].

The energy efficiency is of particular interest for mobile computing. (i.e. laptop and palmtop computers and many other portable devices). The same hardware trend mentioned earlier drives embedded processors toward multi-core per chip development. Indeed embedded applications are a natural fit and execute faster with multi-core technology, if the task can be partitioned between different processors. Yet, the fact that embedded processors often run under limited battery life and energy efficiency (in addition to computing power) is a determining factor in their design.

This dissertation addresses both energy and performance design constraints in modern processors. In the first part, we propose our energy optimization technique for the Branch Target Buffer (BTB). BTB is a major energy consuming structure often used by branch predictors in single and multi-core processors. Exploiting the BTB improves performance by making early identification of target addresses (for control flow instructions) possible. To achieve this, at fetch, modern processors access BTB for all instructions to find the branch/jump target address as soon as possible. This aggressive approach helps performance, but it is inefficient from the energy point of view. This is due to the fact that control flow instructions (conditional or unconditional branches)

account for less than 25% of the fetched instructions [16]. Therefore, many BTB accesses consume energy and produce heat but do not contribute to performance.

The second part of this dissertation starts with studying the cache complexity. We formulate a quantitative methodology to analyze different cache improvement techniques and evaluate their efficiency from energy point of view. Then we concluded from the results and propose an optimized cache replacement algorithm for chip multi-processors which improves average memory access time by reducing cache miss rate. The new cache management prevents the shortcomings in the widely used LRU (Least Recently Used) policy at the expense of auxiliary hardware overhead. The improved cache management introduces high data availability without increasing the associativity. This is particularly beneficial for memory intensive workloads which run under the pressure of fast processor clock cycles and have a working set greater than the available cache size.

The rest of this dissertation is organized into following five chapters:

Chapter 2: Background and Related Research Work

Chapter 2 provides an overview of the cache memory and branch predictor units. Those are the two crucial components used in almost all modern processors and are the main focus of this dissertation. In this chapter we first describe the key technology and standards for each unit and then review some of the main constraints and design challenges involved. Finally we present the related studies addressing the issues in past.

Chapter 3: Design and Analysis of an Energy-aware Branch Target Buffer

Chapter 3 introduces an energy-aware method to identify and eliminate unnecessary BTB accesses. Our technique relies on a simple energy efficient structure, referred to as the BLC-filter, to identify cycles where there is no control flow instruction among those

fetched, at least one cycle in advance. By identifying such cycles and eliminating unnecessary BTB accesses we reduce BTB energy dissipation (and therefore power density). Exploiting BLC-filter on an embedded processor we eliminate half of unnecessary BTB accesses and reduce total processor energy consumption by 3% with a negligible performance lost.

Chapter 4: Cache Complexity Analysis for Modern Processors

Chapter 4 presents how cache complexity impacts energy and performance in high performance processors. Moreover, in this work we estimate cache energy budget for modern processors and calculate energy and latency break-even points for realistic and ideal cache organizations. We calculate these break-even points for embedded and high-performance processors and for different applications. We show that design efforts made to reduce the cache miss rate are only justifiable if the associated latency and energy overhead remain below the calculated break-even points. We also study alternative cache configurations for different processors and investigate if such alternatives would improve energy-efficiency.

Chapter 5: Reducing Non-optimal LRU Decisions in Chip Multiprocessors

In Chapter 5, we analyze the LRU cache management in chip multi-processors and account for non-optimal decisions (NODs) made by the replacement algorithm. Our speculative technique reduces a significant amount of the undesirable decisions in a conventional LRU policy used in CMP processors and closes the gap between the LRU and Belady's theoretical optimal replacement policy [45].

Exploiting Speculative Replacement Algorithm (SRA) on a quad core chip multiprocessor could reduce on average 7% of L1 data cache miss rate while using about 1k entries history table.

Chapter 6: Conclusions

Chapter 6 summaries the key achievements in this dissertation and suggests the possibilities for further improvements in future research.

Chapter 2

Background and Related Research Work

This chapter provides the required background to readers unfamiliar with the concepts of computer architecture which will be discussed in this dissertation. The chapter is organized into two major subsections, which discuss the basics of cache memories and branch prediction. After a preliminary introduction in each section we review some of the techniques and approaches offered in recent literature to improve performance or reduce energy consumption.

2.1 Introduction

Modern processors are optimized in many ways to execute a larger number of instructions in less time. Pipelining, out-of-order execution, multithreading, and parallel computing are just a few of many recent enhancements introduced by computer architects to expand computers' throughput and processing speed. The ultimate optimization goal for all of these design advancements is to increase a continuous stream of data and instructions to the processing units and execute as many applications as possible in the shortest time with minimum energy consumption. In order to achieve this goal in the field

of computer architecture, processor designers create, modify, extend or customize existing components or their interactions.

In this chapter, we present a background review of cache memories and branch predictors, which are two fundamental blocks commonly present in almost all modern processors, and are the main focus of this dissertation. We also present some of the enhancements in this area that have been published for these two blocks in recent literature.

2.2 Cache Memory Essentials

Computer programmers always have a demand for an unlimited source of fast memory. Unfortunately, this ideal is not feasible with current technologies. In order to reduce the access time to a virtually unlimited number of memory locations, computer designers have suggested the concept of *memory hierarchy* based on the principle of locality [20].

The importance of memory hierarchy becomes more obvious as the gap between processor and memory performance increases. Memory hierarchy consists of multiple levels of memory with different sizes and speeds. The fastest types of memory are more expensive to implement per byte and thus are usually smaller. The design goal is to provide a memory system with a cost as low as the cheapest level of memory and with a speed as fast as the fastest level. Each level of faster memory usually contains a subset of data which can also be found in the slower memory level below.

Cache is the first level of memory hierarchy and it is accessed when the data or instruction requested by the CPU is not found in the internal registers. If the requested data is found in the cache, it is called a *cache hit*. The requested data will be passed to the

CPU, and the memory access time (*hit delay*) will be a small portion of the time which would otherwise be spent to retrieve the data from the main memory. If data is not found in the cache it is called a *cache miss*. In this case, a fixed-size *block* of data is fetched from the lower level of memory hierarchy (*e.g.* main memory) and placed in the cache for quick access in likely future reuses. The time spent to retrieve the data from the lower level of memory hierarchy is called the *miss penalty*. This time would be saved as long as the data resides in the cache.

A cache memory which holds instructions for possible reuse is called *instruction cache* memory. Instruction caches are highly useful if the application code has an iterative behavior. *Data cache* maintains data values which are fetched from data memory. The different characteristics of instruction and data caches result in slightly different considerations in cache design.

2.2.1 Cache Memory Organization

Data is stored in the cache in fixed-size collection of bytes called a *block*. Cache organization defines where to store a block of data. If only one location for each block of data exists in the cache, the cache organization is called *direct mapped*. If the data block can be placed anywhere in the cache, the term *fully associative* is used. If data blocks are restricted to certain sets of places in the cache, the cache organization is *set associative*.

A *set* is a group of blocks in the cache which coexist in a single cache entry. If there are *n* blocks in a set, the cache organization is called *n-way set associative.* The cache capacity can be calculated as the product of associativity, the number of entries (sets) and the size of a block in bytes.

2.2.2 Cache Addressing

In order to reference a data value in the cache, the CPU sends the address of data to the cache. The provided address has three sections. The first part which includes the least significant bits of the address indicates the location of the byte in the data block. This portion of the data address is called *block offset* (Figure 2.1). The block offset size depends on the number of bytes in the data block.

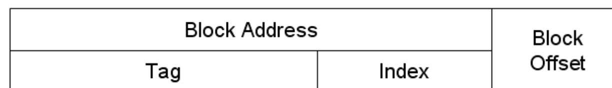

Figure 2.1: The three portions of an address in set-associative and direct-mapped caches.

The second portion of the data address is called the *index* and it points to the cache entry where the data content is stored. The size of index depends on the number of entries in a direct mapped cache or the number of sets in a set associative cache.

The rest of data address is called the *address tag* and is stored in the cache along with data content. In a cache access, the address tag of the requested data is compared with all the address tags stored in the same set, and if there is a match the data value is retrieved.

2.2.3 Cache Replacement Policies

In a set associative cache organization, when all possible locations to store a data block in the set contain other data; the *cache replacement policy* decides which block should be discarded. The *victim block* is evicted from the cache to open room for the arriving data block. Obviously, in the direct mapped caches there is only one possible location to store the block of data and no replacement strategy is needed. In the case of fully associative

caches, the new data can be placed in any location depending on the insertion or replacement algorithm.

There are three primary strategies to select a victim block for replacement:

- *Random*: candidate blocks are randomly selected to be evicted from the cache. This strategy will uniformly allocate data blocks in a set.
- *Least-recently Used (LRU)*: This strategy reduces the chance of throwing out information that will be needed soon: it gives recently accessed blocks a higher priority to stay in the cache. Therefore, LRU relies on the principle of locality (*i.e.* the recently used data are likely to be used in near future) and victimizes the cache blocks which have least recently been accessed.
- *First in, First out (FIFO)*: FIFO simplifies LRU operation by assuming that the oldest block that has entered the set is the least likely to be accessed again.

LRU is the most widely used replacement strategy in most computers' cache memory; however, LRU implementation will become increasingly difficult as the associativity and the number of blocks to track increases.

2.2.4 Cache Miss Classification

An ideal cache memory would maintain all requested data from the CPU and would always return a cache hit. Unfortunately, due to existing limitations, real cache structures do not perform perfectly. There are several reasons a cache miss might occur. These reasons include:

- *Compulsory misses*: The first access to a block of data will result in a cache miss unless other techniques such as prefetching (fetching data earlier than they are needed. Prefetching is introduced later in this chapter) are used. Compulsory misses are also called *first-reference misses* or *cold-start misses.*
- *Conflict misses*: In a set associative or direct mapped cache organization, if too many blocks are mapped to the same set, a data block might be discarded due to lack of space in the set. Future references to the evicted block will result in a cache miss. Conflicted misses are also called *collision misses* or *interference misses*.
- *Capacity misses*: If the cache capacity can not hold all the data referenced during the program execution, capacity misses will occur due to blocks being discarded and are later retrieved.
- *Coherence misses*: This type of miss occurs in a multiprocessor system with local caches sharing data in main memory. If one processor changes its local copy, a cache coherence mechanism invalidates the shared data in other processors in order to maintain coherence between the shared data. This invalidation of data causes a miss in other processors which reference the same data block.

2.3 Cache Enhancements for Miss-rate and Miss-penalty Reduction

Cache performance improves when fewer accesses to cache return a cache miss. Computer researchers have investigated many design alternatives to reduce the cache miss rate while maintaining cache hit time. Changes in the cache structure can reduce

cache misses. For instance, conflict misses can be reduced with higher associativity. A larger cache will reduce capacity misses whereas compulsory misses would be reduced if block size is increased. These changes could also increase hit time or miss penalty. In the following subsections, we present some techniques from recent literature which reduce cache miss rates and balance other parameters to make whole systems work faster.

2.3.1 Victim Cache

One way to reduce the cache miss rate is to store what has recently been discarded from cache in a buffer. If the evicted blocks are referenced soon after their replacement, they can be easily retrieved from the buffer.

This recycling buffer should be a small, fully associative cache located on the main cache refill path. In case of a miss the victim block will be stored in this *victim cache* and the cache also is checked to see if it contains the missing block. In [44], it was shown that, depending on the program, a small four-entry victim cache effectively reduces one-quarter of the misses in a 4KB direct-mapped data cache.

2.3.2 Non-Blocking Cache

In out-of-order processors, multiple instructions can be executed and completed in a single clock cycle. For such architectures, there is a chance that independent instructions in the pipeline each require access to the same section of memory. Therefore, a cache miss should be prevented from stalling memory access for other instructions. The *non-blocking caches* provide the benefits of such a scheme to other instructions. Data caches with the non-blocking technique continue to supply hit data during a miss. This

optimization is called "*hit under miss*" and reduces the miss penalty by remaining attentive to CPU requests.

A slightly different version of non-blocking caches allows multiple cache misses from different requests. In such cases cache memory continues to serve other instructions despite multiple outstanding misses. Such a caching system will employ complex controllers with multiple memory ports. In [29], Farkas and Joupi demonstrated that a significant number of hits are serviced under multiple misses for an 8KB direct-mapped data cache.

2.3.3 Hardware and Software Prefetching

Another technique to reduce miss rate and penalty is to fetch a memory block before it is requested by the processor. Both data and instruction can be *prefetched* either *dynamically* during program runtime or *statically* at compile time. In the first approach dedicated hardware will dynamically speculate future requests to memory and prefetch them directly into the cache or into the external buffer. In a simple implementation for dynamic instruction prefetching, the processor fetches two blocks on a cache miss. The requested block is placed in the cache and the second block is placed in a stream buffer. If the requested block is found in the stream buffer, the original cache request will be canceled and the instruction block is retrieved from the stream buffer. A similar approach can be implemented for data blocks.

In [48], Palacharla and Kessler showed that for a scientific suite of applications an eight entry stream buffer, implemented for dynamic data and instruction prefetching,

could capture consecutively 50% and 70% of all data and instruction misses from a processor with two 64KB 4-way set associative caches.

In an alternative static approach, the compiler inserts a special prefetching instruction in the compiled code to prefetch the data before it is needed. The prefetched data can be placed either in cache or in prefetch registers. Data and instruction prefetching is only valid for non-blocking caches, which means that the data cache should be able to continue supplying data and instructions while waiting for the prefetch data to arrive.

2.3.4 Trace Cache

In out-of-order architectures multi instructions are often executed in parallel. In order to facilitate sufficient instructions for such computers, several instructions should be fetched every cycle. *Trace cache* is a technique to improve hit rate in instruction caches. In this technique cache blocks are not limited to spatial locality. Instead, a sequence of instructions, including taken branches, are loaded into instruction cache blocks.

The trace cache blocks contain a dynamic trace of executed instructions rather than static sequences of instructions as placed in the memory. Hence, the trace cache address mapping mechanism is more complicated than normal caching. However, in case of low spatial locality the trace cache benefits are more observable when compared with conventional caches with long block size. When there is a low spatial locality, taken branches would change the program flow by jumping to close distances. As a result, the continuous sequence of instructions occupying the cache block space would be wasted, causing multiple accesses to the instruction cache.

Apart from implementation difficulties, the downside of trace caches is that they store the same instruction in multiple locations. In [17] trace cache is discussed in more detail. Modern processors, such as Intel Pentium 4 (NetBurst Microarchitecture), exploit trace caching in their instruction cache.

2.3.5 Way-Prediction

Way prediction is an energy-efficient technique which also improves caches latency. In this method, a small predictor is laid inside the set-associative cache in order to speculate the next way or block within the set, which is likely to be referenced in the following cache access. The prediction saves the time it takes to set the multiplexer's selection bits and also saves energy by comparing only one address tag. If the first tag does not match, then other tags in the cache set are compared. This introduces a longer latency. Simulations performed by Albonesy [11] suggest that set way prediction is about 85% accurate, which means that it degrades energy and cache latency for 85% of instructions in the pipeline.

2.3.6 Improving Cache Replacement Algorithm

Many researchers from both industry and academia have examined cache replacement management for different workloads in both single and multi-core processors. In set associative caches, cache blocks are not referenced and reused similarly. So, in the case of a cache conflict, selecting the victim block to be replaced is challenging. The commonly used Least Recently Used (LRU) replacement policy does not cope with the blocks' access pattern, which results in non-optimal replacement decisions and unnecessary cache misses. Based on this observation, new optimizations have been suggested to improve cache management for particular workloads or in general.

Early eviction of blocks which are unlikely to be referenced in the near future can improve victim selection in LRU policies. Maintaining such blocks in the cache occupies the cache space without contributing to cache performance. Lai et al., [2] proposed early substitution of these blocks with prefetch data. In their simulation they eliminated a large number of the memory stalls and achieved a 62% speedup.

Smart insertion policy is another mechanism to improve cache replacement algorithms. In this technique, LRU is modified to insert the incoming cache blocks with low reuse pattern in the LRU position. Qureshi et al., [38] showed that for a memory-intensive workload this technique on average reduces 21% of misses for a 1MB 16-way L2 data cache.

2.4 Branch Prediction Essentials

Control instructions such as jumps and branches are frequently found in program code and could potentially stall the instruction fetching stream. This problem is more obvious in high performance processors, in which several instructions are often fetched per cycle. Thus, almost all modern processors employ speculative methods in order to keep a continuous stream of instructions in the pipeline.

A *branch predictor* is a hardware block commonly used in modern processors to speculate the outcome of conditional branch instructions and the target address of the next fetching instructions. While predicting a conditional branch might help to fetch more instructions into the pipeline and possibly to execute more instructions in parallel, a branch misprediction may cause a deep execution of program code from a wrong path which wastes processor resources, energy and time.

Studies have shown that branch instructions' behavior is highly predictable [53]. It was shown that branch instructions tend to repeat their past behavior. Also, one branch instruction might correlate with other branch instructions. Therefore, the knowledge of a sequence of branches' outcome can be used to correctly predict the subsequence occurrence of the branch instruction and correlated branches.

Based on this observation, branch predictors use the history of the conditional branch instructions to predict the outcome of future branches. The main challenge in branch prediction is to find the correct connection between the collected history of branch outcomes, and future occurrences of the same or correlated branch instructions. In this section we review some of the branch predictor essentials. More details about branch predictor's structures can be found in the appendix C and [20].

2.4.1 Branch Target Prediction

In a pipeline processor, after fetching the current instruction, the fetch engine must be aware of the next address so it can fetch the following instruction into the pipeline. In order to achieve this, the fetch engine requires the op-code information about the instruction which is already fetched but not decoded. If the instruction is a non-branch or a not-taken branch the next address is the offset of one instruction added to the current PC (Program Counter). However, if the current instruction is a taken branch or a jump to a different location in the code, finding the next PC address becomes more complicated and depends on the branch addressing mode. In offset addressing mode, the offset value would be added to the current PC to produce the next fetching address. In contrast, in indirect addressing mode the fetch engine obtains the next address from the content of a register. This procedure can increase the fetch penalty and delay fetching until the decode stage.

A branch prediction cache that stores predicted addresses for next instructions after a branch can facilitate target address resolution in fetch engines and reduce branch penalties. This address buffer is called the *Branch-Target Buffer (BTB)* or *Branch-Target Cache*.

A branch-target buffer has a structure similar to the cache hardware. Every BTB entry consists of both the current PC and the predicted next PC. At the fetch stage, the instruction PC is fetched and looked up in the BTB. This process occurs at least one cycle before the fetching instruction is identified at the decode stage. If the sought after PC is found in the BTB, it indicates that the instruction currently being fetched is a branch. In this case the BTB will return the predicted PC of the next instruction. If the PC of the

current instruction is not found in the BTB, the current instruction is treated as a non-branch instruction and the fetch engine will fetch the next sequential instruction. This strategy will not introduce a branch delay in the pipeline, provided that the branch instruction PC is found in the BTB and the next PC is predicted correctly.

2.4.2 Branch Direction Prediction

The conditional branch instructions do not always cause a change in program flow. The condition of the branch instruction will determine whether the branch is *taken,* and the flow of instruction fetch should be redirected; or whether it is *not-taken,* and next consecutive instruction should be fetched normally. However, the branch condition will not be resolved until the execution stage. The goal of the branch direction predictor is to allow processors to resolve the outcome of the branch instruction early, thus preventing any stalls in instruction fetching. The effectiveness of a branch predictor scheme depends on its accuracy and the cost of predicting a branch direction incorrectly (*branch misprediction penalty*).

Branch outcome changes with program behavior during runtime. An elaborate branch predictor can accurately evaluate the history of past correlated branches and extract the future pattern of arriving branches' outcome. In the following section, we review the basics of direction prediction with a simple 2-bit history based predictor. Appendix C presents more detail on commonly used branch predictor schemes.

The simplest scheme for speculating a branch direction is to use a *branch-prediction buffer* or *branch history table*. In this scheme, the branch history table is a small memory which is indexed by the least significant bits in the branch instruction

address. Each entry in this table consists of a simple 2-bit saturating counter. When a branch instruction conditional outcome is revealed at the pipeline backend to be a taken branch, the corresponding counter in the history table will increment (until saturated). On the contrary, if the conditional branch instruction is not-taken, the saturating counter is decremented. For future branches, the history table will predict the branch outcome based on the most significant bit of saturating counter value. Figure 2.2 shows the state diagram used for predicting the next branch direction according to the corresponding 2-bit saturating counter value.

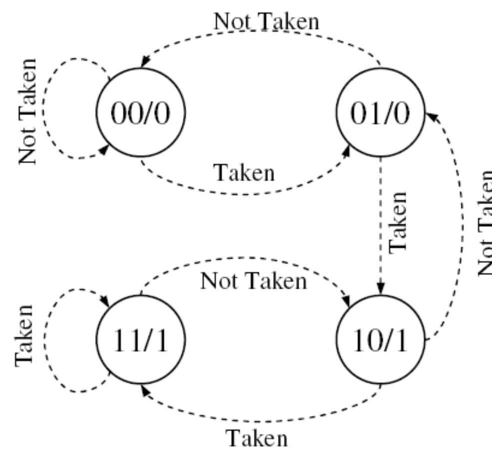

Figure 2.2: Values inside each state indicate that state's saturating counter value following the direction prediction output. Arrows shows the transactions to the next states after the branch outcomes are resolved.

The described direction predictor includes required steps in branch prediction procedure; *i.e. history table lookup*, and *update*. We discuss these two steps in the following section.

History Table Lookup

Branch predictors are accessed in early stages of the pipeline to speculate the direction of a branch. Depending on the prediction mechanism, required steps are taken to look up the address of branch instruction (or other attributes) in the predictor structure and BTB. The

results are a speculative direction, and the target address for the next instruction that will be used by the fetch engine.

In a simple 2-bit predictor, the address of the arriving branch instruction is used to index the history table. Then, the content of the looked up entry (*i.e.* the counter value) is retrieved to predict the branch direction.

History Table Update

After executing the conditional branch instruction in the pipeline, the true (non-speculative) outcome of the branch condition is resolved. At this point, the branch predictor is accessed again in order to update the history table. If the predicted outcome of the conditional branch was different from the actual resolved value, a branch misprediction penalty should be paid. In this case, all subsequent instructions fetched after the mispredicted branch are flushed and the state of pipeline before the branch misprediction is recovered.

The branch misprediction penalty might stall the pipeline for a number of cycles depending on the pipeline structure, the type of predictor and strategy used for recovering from misprediction. Therefore highly accurate predictors are essential for high performance processors.

2.4.3 High Performance Branch Prediction Deficiency

While using highly accurate branch predictors can significantly improve processor performance, such predictors suffer from implementation difficulties. Latency and energy

consumptions are the two major deficiencies in such predictors [14], [15]. These two aspects are discussed in the following sections.

Timing Overhead

Usually accurate branch predictors are associated with a large hardware footprint. Accessing a large storage table or employing multiple stages in the branch predictor introduces high timing overhead and may require a few cycles to complete branch prediction. This prediction delay disrupts the fetching process and affects processor performance. Accordingly, highly accurate predictors might not be able to improve overall processor performance, if they come with high timing overhead.

Energy Overhead

In regards to energy consumption, a highly accurate predictor may save energy by avoiding misprediction penalties and saving that wasted energy for recovering from a wrong execution direction. However, the accurate branch predictor should balance between accuracy and the energy overhead. If accessing a high performance predictor consumes more energy than that it saves by preventing branch penalties, a less accurate predictor might be more beneficial for the overall processor energy consumption.

2.5 Branch Predictor Energy and Performance Improvements

Improving branch predictor accuracy will speed up processor execution time and save overall energy. It has been shown in previous studies [37] that branch predictors consume a considerable portion of energy in a conventional processor. Also, due to frequent access, this block can easily become a hotspot in the processor chip and cause thermal

issues. In this section, we review some of the literature addressing the inefficiencies in branch predictors and branch target buffers.

2.5.1 Neural Network Methods for Branch Prediction

In recent years, designers have proposed highly accurate branch predictors based on neural networks theory. In *perceptron* predictor [13], a simple neural network, the perceptron, has replaced the commonly used 2-bit counters in dynamic branch predictors. In perceptron predictor the hardware resources scale up linearly with history length, rather than exponentially. Hence, the predictor is capable of using long history lengths in its calculations, which results in more accurate branch prediction. In an implementation of this predictor by Jimenez and Lin [12], the Perceptron branch predictor outperformed other well-known branch predictors and improved processor performance by 5.7% over the McFarling hybrid predictor.

Optimized GEometric History Length (O-GEHL) branch predictor [6] is another neural network based predictor that employs long global history lengths in its direction prediction. The O-GEHL predictor features several tables (*e.g.* 8 tables) which are indexed through functions of global history and the branch address. The O-GEHL predictor effectively captures correlation between recent and old branches and speculates the branch outcome through the addition of the predictions read on the predictor tables.

The downsides of the neural network methods for branch prediction are their extensive computation and hardware storage requirements. As a result, such predictors often introduce excessive latency and energy dissipation, which makes their implementation impractical.

2.5.2 Static Energy-aware Branch Prediction

This technique proposes an energy efficient branch prediction technique based on a compiler hint mechanism, which filters unnecessary accesses to branch predictor blocks. The technique requires a hint instruction to be inserted statically during compile time. The hint instruction anticipates some static information about the upcoming branches and reduces hardware involvement during run-time.

This method, when combined with known dynamic branch predictors, can reduce energy consumption with almost no performance degradation. In [41], Monchiero et al., implemented this prediction methodology for VLIW processors and observed an average 93% access reduction to the branch predictor which saved 9% of total processor energy.

2.5.3 Dynamic Energy-aware Branch Prediction

There are several techniques proposed to reduce the branch predictor and BTB power dynamically during program runtime. Unlike static techniques, these techniques do not require recompilation of the applications or changes to the hardware structure.

Parikh et al. [16] reduced energy consumption of branch predictor and BTB by introducing Banking and Prediction Probe Detector (PPD). Banking reduces the active portion of predictor block, and PPD filters unnecessary accesses to the branch predictor and BTB. PPD aims at reducing the energy dissipated during predictor lookups. PPD identifies instances in which a cache line has no conditional branches, so that a lookup in the predictor buffer can be avoided. Also, PPD identifies instances when a cache line has no control-flow instruction at all, so that the BTB lookup can be eliminated. The Parikh

and coworker's results showed that PPD can reduce 31% of local predictor energy and 3% of overall processor energy dissipation.

Baniasadi and Moshovos introduced Branch Predictor Prediction (BPP) [3] and Selective Predictor Access (SEPAS) [4] to reduce branch predictor energy consumption. BPP stores information regarding the sub-predictors accessed by the most recent branch instructions executed. This information is used to avoid accessing all three underlying structures. SEPAS selectively accesses a small filter to avoid unnecessary lookups or updates to the branch predictor.

Huang et al., [10] used profiling to resize large BTB structures whenever reducing size does not impact the BTB miss rate. They exploit the fact that many BTB entries are underutilized and suggest adaptive BTB technique to reduce energy consumption. Huang et al. demonstrate that adaptive BTB can save between 20 to 70 percent of energy spent in the branch predictor.

Hu et al., reduced leakage energy in direction predictors [63]. They show that as the branch predictors' structure grow in size, the leakage energy consumption dominates overall predictor energy. Hu et al. propose decay technique which deactivate (turn off) predictor entries or address buffer lines if they have not been used in a long time. They claim that their results reduce BTB energy by 90% and the branch predictor by 40-60%.

Chapter 3

Design and Analysis of an Energy-aware Branch Target Buffer

This chapter employs a speculative resource allocation technique to design an energy-aware Branch Target Buffer. Speculative resource allocation is a special form of dynamic resource allocation which uses prediction methods to dynamically allocate or resize the targeted hardware. The direct advantage of this technique over dynamic resource allocation is the ability to decide about resource allocation in advance.

We evaluate speculative resource allocation by applying the technique to save energy in the branch target buffer (BTB) in both an embedded processor and an out-of-order superscalar processor. More details about BTB and branch predictors' structure are explained in chapter 2 and appendix C.

The work presented here was published in the Proceeding of SAC2006, The 21st ACM International Symposium on Applied Computing [33], and also presented in workshop on Unique Chips and Systems (UCAS-2) in conjunction with IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006)[27], and published in the special edition of Elsevier's Computers & Electrical Engineering journal.

3.1 Introduction

The energy consumption trend in modern processors has increased exponentially since 1990 [58]. Today modern processors are designed under tight power and energy consumption constraints. This is not only a concern for embedded processors using batteries, but also for servers and cluster processors which have to avoid over-heating and expensive packaging costs.

In preceding chapter of this dissertation we investigated static and dynamic techniques to reduce energy in the branch predictor. Resource usage could be customized to save energy. In this method, units with high energy demands are temporarily shut down when their usage has no or very little contribution to the processor performance.

The objective of this Chapter is to use speculative allocation techniques to reduce branch target buffer (BTB) energy consumption without harming accuracy and hence overall performance. BTB is a major energy consuming structure in the processor and is often used by branch predictors for target address prediction. We target the BTB due to the following: First, conventional processor designs access the BTB aggressively and frequently. This requires using multi-ported structures and can result in high temperatures (possibly resulting in faults) and higher leakage [46], [30]. Second, BTB is an energy hungry structure and consumes a considerable share of the branch predictor unit's energy budget.

This chapter is organized in two main sections. In the first section, section 3.2, we introduce and analyze our speculative technique in the embedded processor space. Next,

in section 3.3, we repeat our suggested technique for high performance superscalar processors. Each section is organized to present the following subsections: The speculation technique structure, methodology and results, prediction accuracy and coverage and finally energy-performance tradeoffs.

3.2 Speculative BTB Allocation in Embedded Processors

Embedded processors often perform under resource constraints. It is due to such restrictions that designers use simple in-order architectures in embedded processors. As such, fetching a small number of instructions every cycle provides enough work to keep the pipeline busy [57].

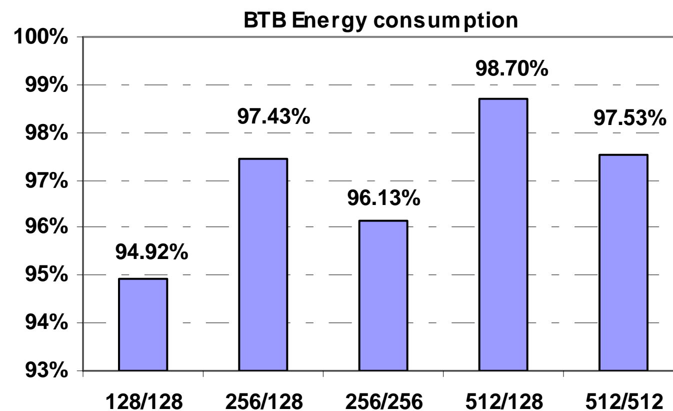

Figure 3.1: BTB's energy consumption share in the branch predictor unit.

While currently exploited branch predictors are already consuming considerable energy [37], their consumption is expected to grow as embedded processors seek higher performance and exploit more resources. In figure 3.1 we have shown the breakdown of energy per access for major units in two selected processor architectures used in this

study. As shown in the figure, and consistent with previous studies, 5 to 10% of processor total energy is consumed in the branch prediction unit.

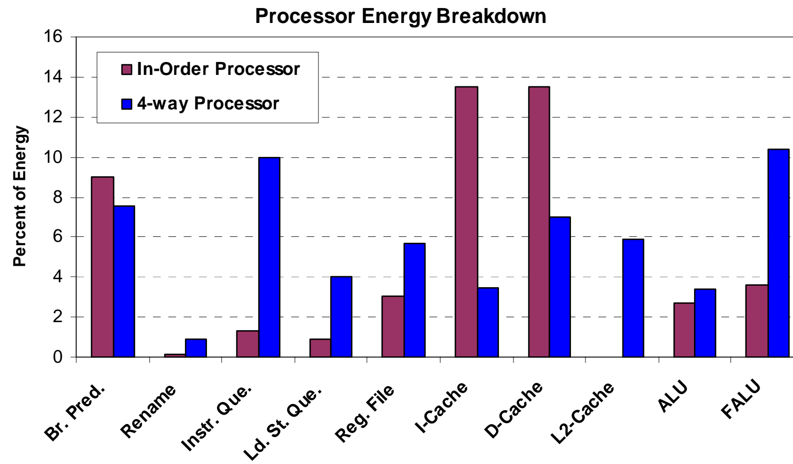

Figure 3.2: Total processor energy per access breakdown. Branch Predictor consumes 5-10% of total CPU energy.

A considerable share of predictor energy is consumed by the BTB. In figure 3.2 we report the percentage of predictor energy, consumed by the BTB in a processor similar to Intel XScale as measured by Cacti [51] tool which is enclosed in Wattch power model and our performance simulator. We report for different configurations (BTB sizes/predictor) to cover both currently used predictors and those likely to be used in future embedded processors. As presented the BTB is a major contributor to the overall predictor energy consumption.

Modern embedded processors use BTB to maintain steady instruction flow in the pipeline front-end [57]. The processor accesses the BTB to find the target address of the next instruction, possibly a branch. Unfortunately accessing BTB every cycle is not energy efficient. While only taken branch instructions benefit from accessing the BTB, the BTB structure is accessed for every instruction. These extra accesses result in energy dissipation without contributing to performance. By identifying occasions where

accessing the BTB does not contribute to performance we can avoid these extra accesses. This will reduce energy dissipation without harming performance.

Table 3.1: The subset of MiBench benchmarks studied and their BLC frequency

Program	BLC	Program	BLC
Blowfish Encode	87 %	*Blowfish Decode*	87 %
CRC	80 %	*Dijkstra*	82 %
D Jpeg	94 %	*C Jpeg*	85 %
GSM Toast	94 %	*Lame*	92 %
Rijndael Encode	94 %	*SHA*	93 %
String Search	78 %	*Susan Smoothing*	92 %

Consistent with previous study [42], our study of an in-order embedded processor [33] shows that embedded applications may have branch frequency anywhere from 5% to 30%. Considering the fact that modern embedded processors (with narrow pipeline) fetch very few instructions each cycle, chances are, quite often, there is no branch instruction among those fetched. We refer to such cycles as *branchless cycles* (or *BLC*). A *branch cycle* (or *BC*) is a fetch cycle where there is at least one branch instruction among those fetched.

To investigate possible energy reduction opportunities we report how often BLCs occur in each benchmark. Table 3.1 reports the percentage of BLCs for a representative subset of MiBench benchmarks [42] used in our study. We select twelve different types of applications, including automotive and industrial control (*Susan Smoothing*), consumer devices (*JPEG, Lame*), office automation (*String Search*), networking (*Blowfish, SHA, CRC, Dijkstra*), security (*Blowfish, Rijndael, SHA)*, and telecommunications (*CRC, GSM Toast*). On average, 88% of cycles are BLCs. In other words, 88% of the time, we could avoid accessing the BTB without losing performance.

Because it would require an enormous number of simulations and take a lot of space to present the results, we could not perform our study on a complete benchmark suite. Our selected benchmarks include applications with diverse number of control instructions. More details about these benchmarks are presented in appendix B.

Quite often a number of consecutive cycles may be branchless. We refer to these periods as branchless intervals. To provide better insight in figure 3.3 we report how often different branchless intervals occur. We report for both an in-order embedded processor and a more advanced future embedded processor with bandwidth of two. As reported, more than 90% of branchless intervals take less than 10 cycles. As the bandwidth of the processor increase often there would be shorter intervals between branch cycles as the likelihood of fetching a branch increases.

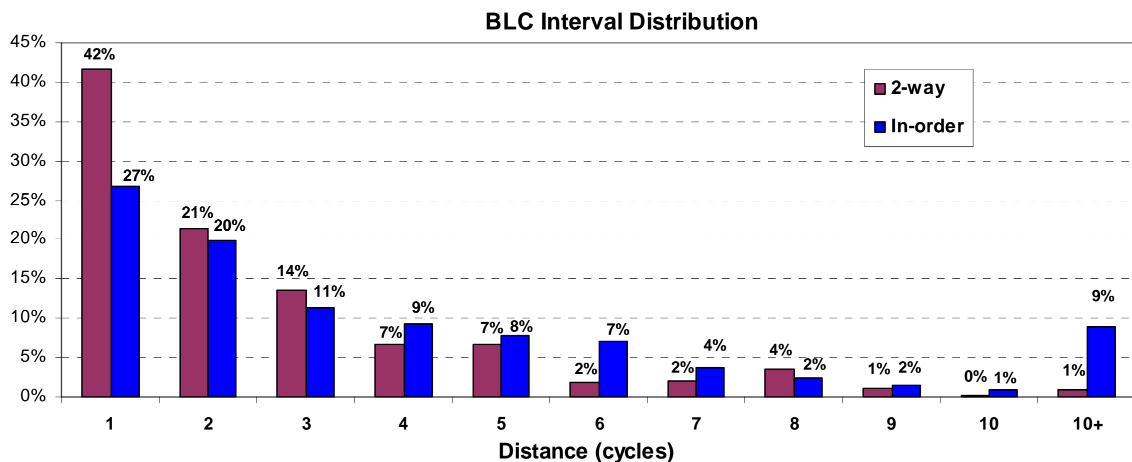

Figure 3.3: BLC interval frequency.

3.2.1 Branchless Cycle Predictor (BLCP)

We conclude from the previous section that there is an opportunity in the embedded space to reduce BTB energy consumption by avoiding unnecessary BTB accesses. In this section we propose a simple, highly accurate and energy efficient predictor to identify such accesses. By identifying a BLC accurately and at least one cycle in advance, we

avoid unnecessary BTB accesses. Depending on the number of instructions fetched during the predicted BLC, we avoid a number of unnecessary BTB accesses. Figure 3.4 shows our proposed BLC predictor architecture. We also refer to our BLC predictor as the BLC-filter, since it filters out unnecessary accesses on branchless cycles.

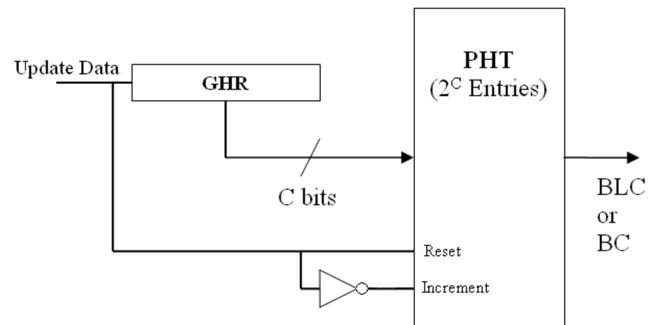

Figure 3.4: The BLC-Filter architecture.

The BLC-filter is a history-based predictor which consists of two major parts: a small Global History Shift Register (GHR) and a Prediction History Table (PHT). The PHT size is decided by the GHR size. GHR is used to record the history of BCs and BLCs. Throughout this chapter we refer to the length of this register as *GHR-size*. The bigger the GHR-size is, the more we know about past history.

BLC prediction is done every cycle. GHR records the most recent branch or branchless cycles. We represent every BLC in GHR with a zero and every BC with a one. The GHR value is used to access an entry in the PHT. Every PHT-entry has a saturating counter with the saturating value of *Sat*.

We probe the predictor every cycle. If the counter associated with the most recent GHR value is saturated we assume that the following cycle will be a BLC and avoid accessing the BTB (see figure 3.5 (a)).

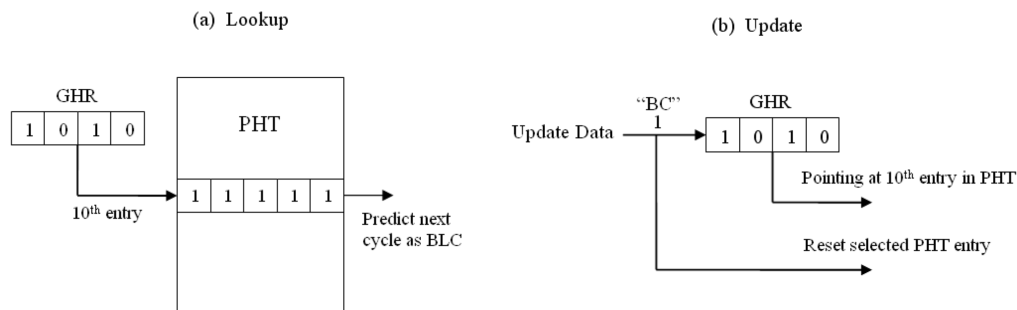

Figure 3.5: (a) Branchless cycle predictor lookup. (b) Branchless cycle predictor update.

We update the BLC-filter every cycle and as soon as we know whether there has been a branch among the instructions fetched in the most recent fetch cycle. A BC results in updating the GHR with a one, where a BLC results in shifting in a zero in the GHR. We use the GHR to access the PHT entry to update. We increment the associated PHT counter if the latest group of fetched instructions does not include any branches. We reset the associated counter if there is at least one branch among those fetched.

As presented in figure 3.5 (b), we update the BLC-filter every cycle. One way to maintain the filter as accurate as possible is to use decode-based information. Accordingly, at decode we check if there has been any branch instruction among those decoded. We update the predictor's entry associated with the history at the time the branch instruction was fetched. Finding the entry is done by shifting left the most recent history by 2 bits (our fetch latency is 2 cycles).

BLC-filter's configuration is similar to gshare branch predictors. Gshare predictors have high prediction accuracy by taking the advantage of program counter value (PC) in probing the prediction history table. PC is highly correlated with branch instructions that can help identifying branch cycles as well. However, the PC value is not used in BLC predictor for two reasons: first, a fetch cycle might include more than one instruction which causes the uncertainty about which instruction PC should be used.

Second, the PC value usually changes when the update value is ready at the decode stage which could introduce bubbling in the pipeline.

3.2.2 Methodology and Energy Reduction Results

We used programs from the MiBench embedded benchmark suite compiled for the Intel Xscale-like architecture using the Simplescalar v3.0 tool set [9]. Details about exploited simulators and tool sets can be found in appendix A. Table 3.1 reports the subset of MiBench benchmarks we used in our study. We used GNU's gcc compiler. We simulated the complete benchmark or half a billion instructions, whichever comes earlier. We detail the base processor model in table 3.2.

Table 3.2: Simulated processor configuration

Processor Core	
Instruction Window	RUU= 8; LSQ=8
Issue Fetch	1 Instruction per Cycle;
Issue Width	2 Instruction per Cycle; 1 integer, 1 FP
Miss Pred. Penalty	6 cycle
Fetch Buffer	8 entries
Functional Units	1 Int ALU, 1 Int mult/div 1 FP ALU, 1 FP mlt/div, 1 mem port
Memory Hierarchy	
L1 D-cache Size	32 KB, 32-way, 32B blocks, wr bk
L1 I-cache Size	32 KB, 32-way, 32B blocks, wr bk
L1 latency	1 cycle
L2	N/A
Memory latency	32 cycles
D-TLB/I-TLB Size	128/128-entry, fully assoc., 30-cycle miss
Branch Prediction	
BTB	128-entry, 1-way
Direction Predictor	bimodal predictor, 128 entries
Return-address-stack	N/A

To evaluate our technique, we used a modified version of Wattch tool set [8]. We report both accuracy (i.e., how often we accurately predict a BLC) and coverage (i.e., what percentage of BLCs are accurately identified).

Provided that a sufficient number of BLCs are accurately identified, BLCP can potentially reduce BTB energy consumption. However, it introduces extra energy overhead and can increase overall energy if the necessary behavior is not there.

We used CACTI [51] to estimate the energy overhead associated with BLCP. We report the relative energy consumed per access for variable sizes of BLCP (our selected BLCP configuration is discussed in section 3.2.4) and other structures used by the branch predictor in table 3.3. Numbers reflect the energy consumed by each structure compared to the energy consumed by a branch predictor equipped with 128-entry bimodal predictor and a direct-mapped 128-entry BTB. As reported the overhead of the 8-entry BLCP filter is far less than the energy consumed by the BTB. Nonetheless, we take into account this overhead in our study.

Table 3.3: Energy consumed per access by the branch predictor units and the BLC-filter.

Modeled Unit	Size	Percentage
BLCP	2x2 to 64x6	< 4 %
BTB	128 x 1-way	94.9 %
Bimodal Dir. Predictor	128 entries 2- bits	3.4 %

3.2.3 Predictor Accuracy and Coverage

To measure the performance of BLC Predictor we use the concepts of accuracy and coverage. These two concepts are often used to evaluate any binary classifiers. Coverage (also called sensitivity) is the portion of BLC cycles detected truly (true positives) of all the BLC cycles exist (true positives and false negatives). Whereas, accuracy define as the portion of BLC cycles detected truly (true positives) of all the detected BLC cycles (true positives and false positives).

In figure 3.6, we report prediction accuracy and coverage for BLCP. In 3.6 (a), we report average accuracy for different GHR sizes (i.e., 1 to 6) and saturating counters (i.e., 2 to 6-bit counters). As reported, we predict BLC cycles with an accuracy varying from 91% to 99.6% for different BLC-filter configurations.

Note that mistaking a BLC for a BC does not harm performance as it only results in unnecessary BTB access. However, mispredicting a BC as a BLC can result in late target address identification, and it comes with an extra cycle penalty in our study. This is due to the fact that the required information to update the filter would be available one cycle later and at the decode stage.

In 3.6 (b), we report the percentage of the identified BLCs. We identify between 25% and 81 % of BLCs for different BLCP configurations. We conclude from figure 3.6 that with a fixed GHR size, increasing the saturation threshold will generally increase accuracy but reduce coverage. With a fixed counter size, increasing the GHR size does not always improve accuracy and coverage. While a small GHR size results in recording very little history, larger GHR sizes may result in mapping small repeating patterns to different BLC-filter entries which may result in a longer BLCP learning period.

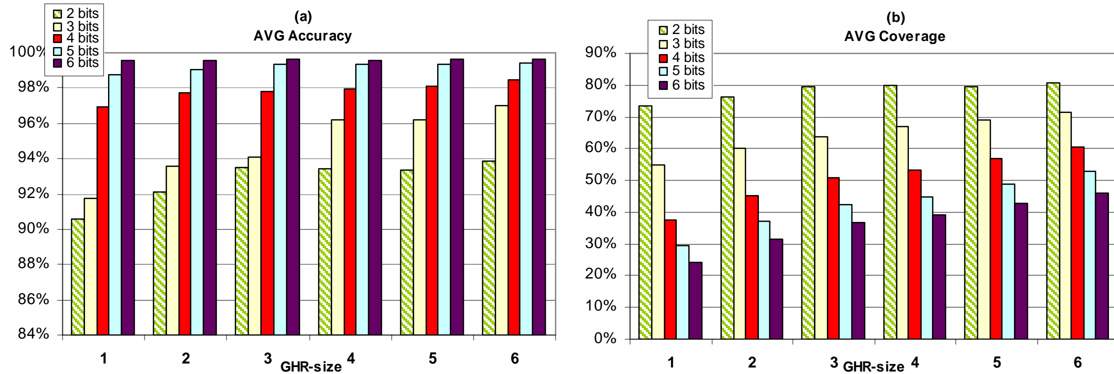

Figure 3.6: Average accuracy and coverage achieved for different BLC-filter configurations and for the MiBench benchmarks studied here. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters.

3.2.4 Energy and Performance Tradeoff

We measure processor speed (performance) using the number of committed instructions per cycle (IPC). We report average processor energy reduction and performance loss (the arithmetic mean value of performance slowdown for the studied benchmarks) in figure 3.7. We use the same set of parameters used in figure 3.6. For a fixed GHR-size larger saturation threshold comes with lower performance loss and more energy savings. Note that small number of saturating bits (e.g., 2-bit counters) results in high performance slowdown (IPC reduction) which also increase total energy consumption. Larger GHR sizes also come with high overhead as they require larger PHT tables.

In figure 3.8, we report total processor energy delay squared product (ED^2) and average BTB energy reduction. BTB energy reduction varies between 21% and 75% for different BLC-filter configurations. For a fixed counter size, increasing the GHR-size almost always improves BTB energy reduction. Considering total processor's energy delay squared product (figure 3.8(a)), we find a BLC-filter with a GHR-size of three which uses 6-bit saturating counters the most efficient one. As explained earlier, BLC-filters with lower GHR-size reduce the processor performance and increase the energy delay squared product (figure 3.8 (a)). The figure also shows, using GHR-size larger

than three bits will increase hardware overhead without resulting in major improvement in energy reduction. Using our selected configuration (GHR-size 3-bit with 6-bit saturating counter), on average, we reduce BTB and total processor energy consumption by 32% and 3.3% respectively with an average performance cost of 0.2%.

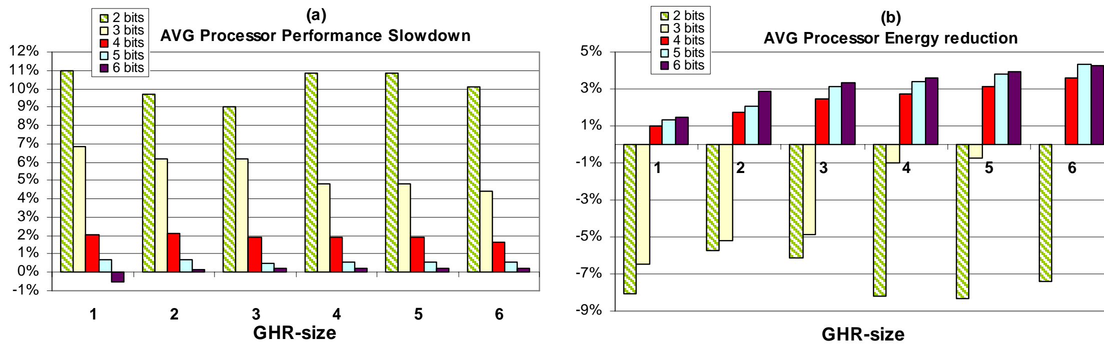

Figure 3.7: Average performance slowdown and Processor total energy reduction for different BLC-filter configurations. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters.

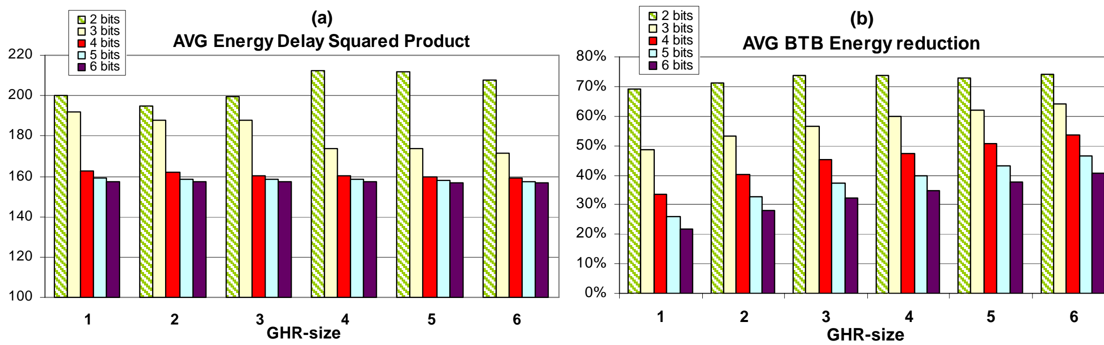

Figure 3.8: Average energy delay squared product and BTB energy reduction for different BLC-filter configurations. For each GHR-size (x-axis) bars from left to right report for 2 to 6-bit saturating counters.

3.3 Speculative BTB Allocation in High-Performance Processors

We apply the previously explained technique to a high-performance modern superscalar processor [33]. Aggressive instruction fetch mechanism in superscalar architecture facilitates executing as many instructions as possible in parallel in the pipeline. The number of fetched instructions varies during program runtime depending on program behavior and processor configuration. This motivates investigating BTB access elimination during branchless cycles in the superscalar space.

Fetching high number of instructions per cycle has two consequences. First, as the number of instructions fetched per cycle increases so does the energy savings potential. This is due to the fact that more instructions are fetched during a branchless cycle. Therefore, correctly detecting a branchless cycle reduces more energy compared to the single instruction per cycle fetching scenario exploited in a typical embedded processor. Second, the average gap between branch cycles decreases. This could make our techniques less desirable in the high-performance space as there will be fewer branchless cycles and fewer opportunities to reduce energy. Figure 3.9, reports the BLC frequency for processors with different execution bandwidths and for the SPEC CPU2000 used in this work. As presented, BLC frequency is less comparing to that reported in table 3.1 for embedded processors.

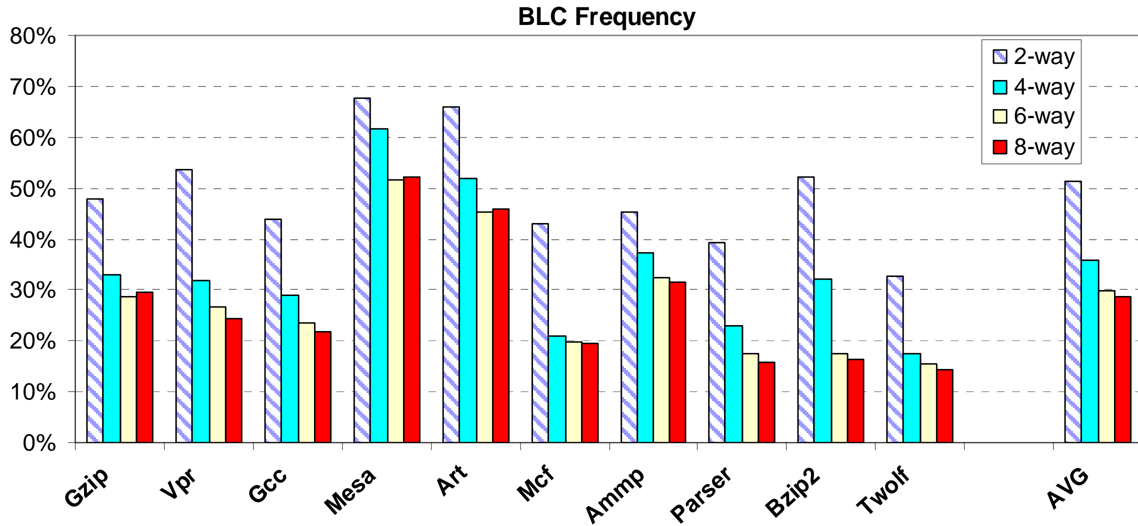

Figure 3.9: BLC frequency for different applications and for different execution bandwidths

We used an alternative configuration for the predictor presented in figure 3.4, to predict BLCs at least one cycle before they occur. Figure 3.10, shows the block diagram including the filter used to eliminate unnecessary BTB accesses for a high-performance processor.

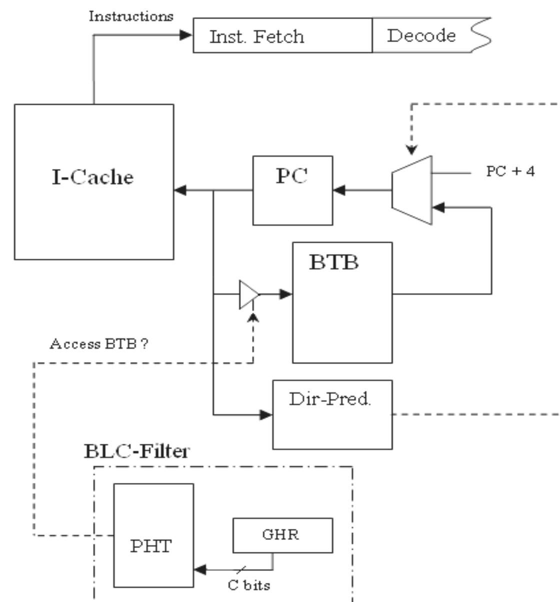

Figure 3.10: BLC filter inserted in fetch stage

The major difference between this filter and the one used in the embedded processor is that when a branch cycle is detected, the appropriate counter in pattern history table is decremented rather than being reset. Resetting the saturating counters demands that a certain pattern repeatedly being followed by a BLC before BLC filter could identify the branchless cycle (saturating counters start counting from zero). This will increase the BLC filter's confidence but limits the prediction coverage only to repeatable patterns. However, BLC frequency in high performance processors (with wider execution bandwidth) does not provide as many repeatable opportunities as in in-order processors. Therefore, BLC predictor in high performance processors decrements the saturating counters on branch cycles and predicts branchless cycles with lower confidence.

3.3.1 Energy Reduction Results

We used the previously explained methodology (see section 3.1.2) to evaluate our technique. We used a MIPS-like architecture to simulate the high performance processor. The configuration of the branch prediction unit components and their relative energy consumption, which is computed using Cacti tool, is reported in table 3.4.

Table 3.4: Energy consumed per access for branch predictor units and the BLC-filter.

Modeled Unit	Size	Percentage
BLC-Filter	2x2 to 64x4	< 0.65 %
BTB	512 x 4-way	60.2 %
Comb. dir. Predictor	32k x 3 (LHT) 32k x 2 (GHT) 32k x 2 (sel.)	33.9 %
RAS	64 (entry)	5.6 %

3.3.2 Predictor Accuracy and Coverage

Similar to the section 3.2.3, we repeat our study for the SPEC'2K applications studied in this work. In figure 3.11 (a) we report average prediction accuracy for different GHR sizes (i.e., 1 to 6) and saturating counters (i.e., 2, 3 and 4-bit counters). As reported we predict BLC cycles with an accuracy varying from 74% to 88% for different filter configurations.

In 3.11 (b) we report the BLC-filter's average coverage. We identify between 25% and 61% of BLCs for different filter configurations.

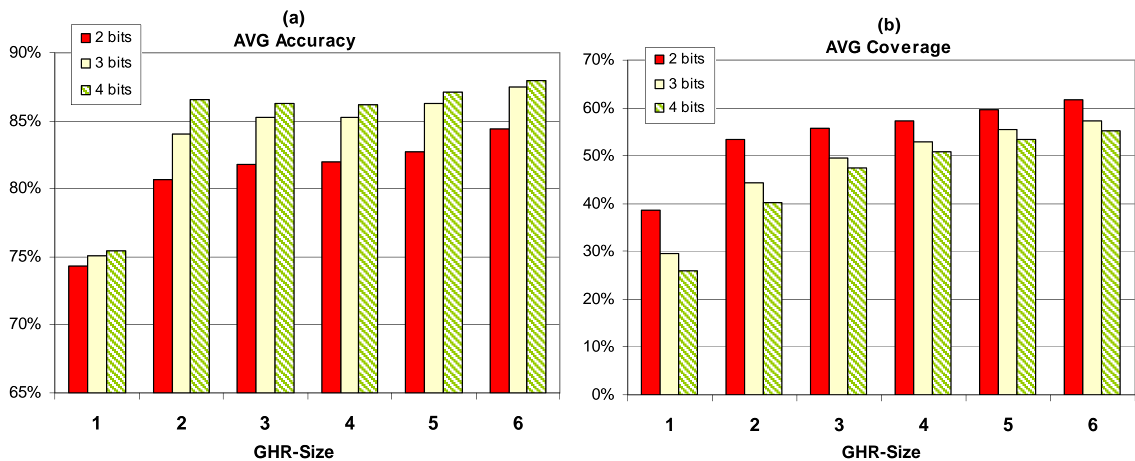

Figure 3.11: Average accuracy and coverage achieved for different BLC-filter configurations

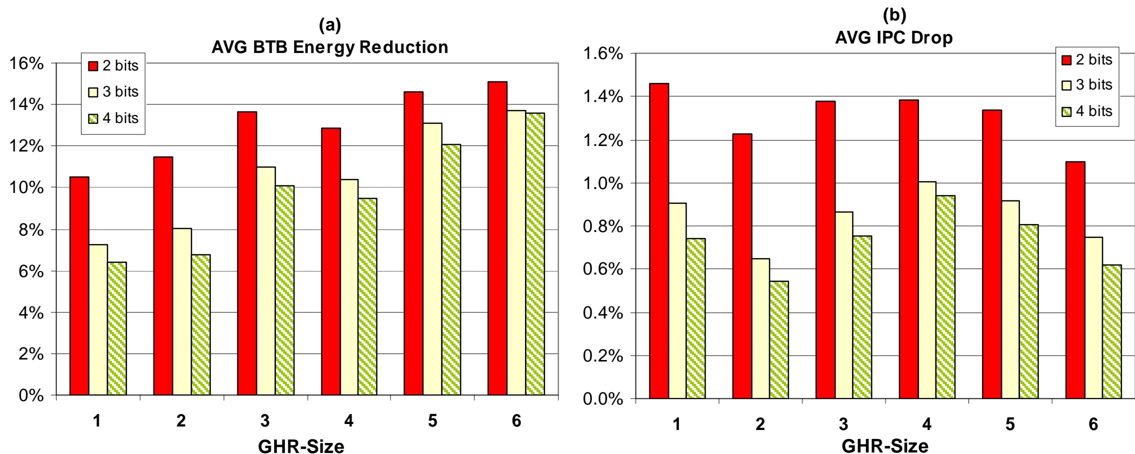

Figure 3.12: Average BTB energy reduction and performance loss for different BLC-filter configurations

3.3.3 Energy and Performance Tradeoff

We report BTB energy reduction and processor performance loss in figure 3.12. We use the same set of parameters used in figure 3.11. BTB energy reduction varies between 6 and 15% for different BLC-filter configurations. Average performance loss is below 1.5% for all configurations. For a fixed GHR-size larger saturation threshold comes with lower performance loss but at the expense of reduced energy savings. For a fixed counter size, increasing the GHR-size almost always improves BTB energy reduction. The only exception is when we increase the size from three to four. We conclude from figure 3.12 that the best GHR-size or history length to consider in predicting application behavior is different from one application to another. This is consistent with other studies [28].

3.3.4 Sensitivity Analysis

We also discuss how BLC predictor performs for different processors. We report the BTB energy reduction and performance cost when BLC-filter is used in processors with different branch predictors and processors with different execution bandwidths.

Branch Predictor Configuration

To investigate how the BLC predictor impacts energy and performance for different branch predictor configurations, we study the following combined branch predictor [49] configurations: a) 2k-entry gshare, bimodal & selector BTB: 256-entry, 4-way b) 4k-entry gshare, bimodal & selector BTB: 512-entry, 4-way c) 8k-entry gshare, bimodal & selector BTB: 512-entry, 4-way and d) 32k-entry gshare, bimodal & selector BTB: 512-entry, 4-way. The gshare predictor used in all predictors uses 6-bit history.

Here we assume a GHR size of six bits and 3-bit saturating counters. This requires using a 64-entry PHT where each entry is a 3-bit counter (192 bits total). We select this configuration as it results in considerable energy reduction while maintaining performance within acceptable limits.

In figure 3.13, we report performance and BTB energy reduction for the four configurations. As reported variations in predictor and BTB size have very little impact on our results. "Mesa" and "art" show higher BTB energy reduction compared to other benchmarks. This is consistent with the data reported in figure 3.9 where both benchmarks show higher number of BLCs compared to other applications.

Note that in the case of "mesa", we witness a small (less than 0.1%) performance improvement. Our study shows mesa achieves better performance as the result of better prediction for indirect jump instructions. "Art", on the other hand, shows the highest performance loss among all applications. This can be explained by the fact that the BLC predictor does not predict BLCs occurring in "art" as accurately as other applications.

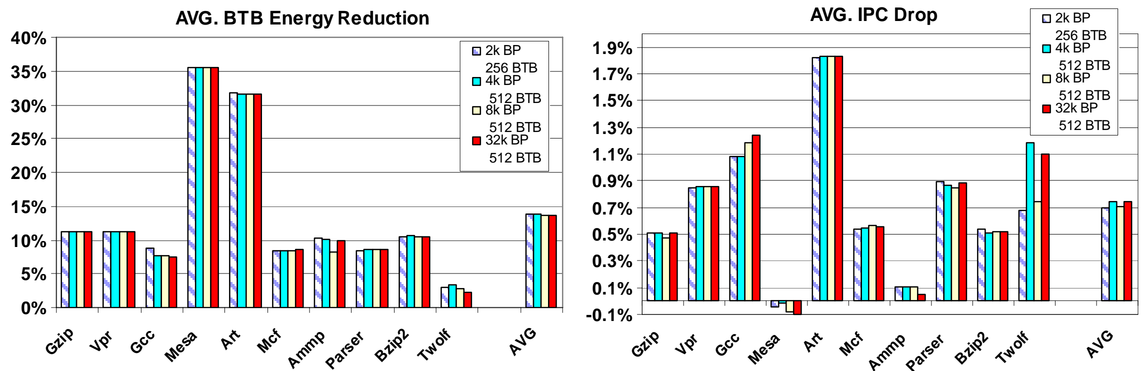

Figure 3.13: Average BTB energy reduction and performance loss for different branch predictor and BTB size.

In figure 3.14, we investigate the impact of predictor and BTB size variations on BLC predictor's accuracy and coverage. Results show that the impact of varying alternative configurations on BLC predictor's performance, are very little.

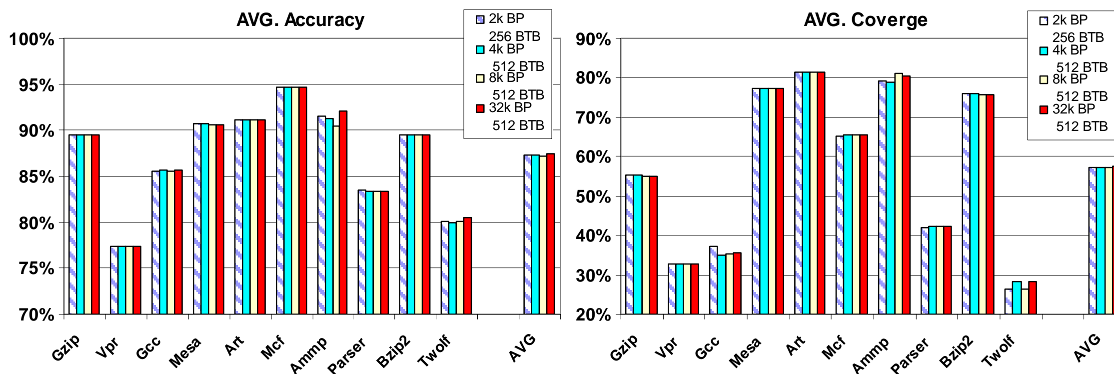

Figure 3.14: Average BLC-Filter accuracy and covergae for different branch predictor and BTB size.

Execution Bandwidth

In figure 3.15 we report the BTB energy reduction for four different processor pipeline execution bandwidths. To provide insight we also report maximum energy reduction possible as achieved by a perfect BLC predictor (Oracle). While the entire bar shows maximum energy reduction possible, the lower part in each bar reports energy reduction achieved by a 64-entry BLC-filter using 3-bit counters.

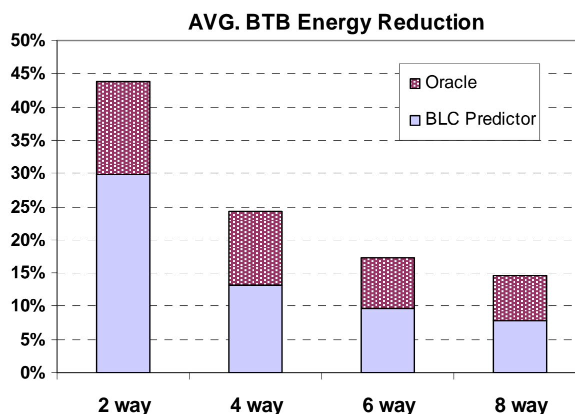

Figure 3.15: The BTB energy reduction for processors with different execution bandwidths.

In a narrow pipeline, e.g. the 2-way processor, fewer instructions are fetched each cycle. With fewer fetched instructions, there is lower chance of having a control

flow instruction among those fetched. Hence, BLCs are more frequent. Consequently, there is higher chance for the BTB energy reduction by using BLC predictor. This explains why the BTB energy reduction is higher for the 2-way processor. As the processor bandwidth increases so does the number of fetched instructions. Therefore, BLCs become less frequent and so does the BTB energy reduction achieved by BLC predictor.

Figure 3.15 shows that the speculative BTB allocation using BLC predictor eliminates more than half of the unnecessary BTB accesses across all execution bandwidths. Note that this comes with negligible performance cost.

Chapter 4

Cache Complexity Analysis for Modern Processors

Amdahl's law expresses the performance return for a fractional enhancement. However, the exceeded energy dissipation due to the enhancements and its limiting perspective seems to be missing. This chapter evaluates the energy efficiency of modern processor's cache structure. We study the impacts of cache complexity on the overall processor energy and performance. Using a quantitative approach, a technique which evaluates the usefulness of an enhancement in the cache, we show that design efforts made to reduce cache miss rate are only justifiable from the energy and performance point of view if associated latency and energy overhead remain below the calculated break-even-point. By formulating energy and latency limitations, we investigate the upper bound budget available for alternative data and instruction cache enhancements.

The work presented here was published in the proceeding of (CCECE 2007), IEEE Canadian Conference on Electrical and Computer Engineering [34], and the workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA 2006) in conjunction with 15th IEEE Inte'l Conference on Parallel Architectures and Compilation Techniques (PACT-2006)[35]. This work also appeared in ACM SIGARCH Newsletters of Computer Architecture News [31].

4.1 Introduction

On-chip caches represent a considerable fraction of the total energy dissipation of processors. Cache energy dissipation already accounts for 25% of the total energy dissipation in DEC 21164 [25] and 43% in SA-110 [22] respectively. As processors continue using larger caches, the cache energy consumption will become even a bigger concern. Exploiting more complex caches in the future may become necessary as the memory-processor performance gap continues to increase. It is due to this widening gap that previous studies have suggested a variety of techniques including alternative replacement policies [43], [26], [64], [55], [40] and prefetching techniques [56], [36], [62]. While such techniques have improved memory performance, they have resulted in higher cache complexity.

Simple caches used in conventional processors often offer hit penalties about one to three cycles [20]. However, as modern processors continue to rely on slower wires and higher clock frequencies, on-chip caches may have to settle for longer access times and higher hit penalties to accommodate faster clocks and larger structures. Accordingly, in the quest for improving cache hit rates, designers must take into account not only hit rates, but also the delay and energy overhead associated with their techniques.

We study cache complexity and show that simply focusing on increasing the cache hit rate without considering the penalties in delay and energy associated with more complex cache structures could be misguiding. We study both energy and latency break-even points for several cache configurations used. We use the ED^2 (energy-delay-square product) metric as a voltage independent metric [61] and the ED metric (energy-delay product) to find energy break-even points. In this quantitative approach we assume that

changing the cache configuration does not require any structural change in other processor components. We estimate maximum energy available to an ideal cache with zero miss rate and no hit penalties. We also estimate the maximum hit penalty affordable in an ideal cache with zero cache miss rate.

The rest of this chapter is organized as follows. First, in Section 4.2, we present energy budget formulations. Next, in section 4.3, we presents energy and latency effectiveness of cache configurations in high performance superscalar processors. We repeated our calculation for an embedded architecture in section 4.4. In each of the last two sections we report our methodology, results and cache complexity tradeoffs.

4.2 Energy Budget Formulation

An important aspect of any design project is to decide how chip real-estate should be distributed among different processor components. The resource distribution would ultimately decide the energy and area share of each component. An important factor in distributing processor resources among components is the final contribution of the component to the overall performance. Therefore it is important to have an accurate estimation regarding how changing design parameters for each processor section impacts overall energy and performance.

Accordingly, one of the goals of this chapter is to study how variations in data and instruction cache complexity impact the overall performance and energy consumption. In this section we provide the formulation needed to estimate how cache energy impacts overall energy efficiency. To study energy efficiency, we explore how changing the

cache organization impacts energy and performance. We also investigate how much energy could be invested in the cache and still maintain the overall energy efficiency.

The formulation presented in this section will be used in following section in both high performance and embedded processors. In our formulation, each processor initial configuration is represented by $Config_{ref,}$. Modified processor configurations (presumably enhanced configuration) are represented by $Config_{new}$. We define the break-even energy budget as the amount of energy a new cache configuration can consume and still maintain the same ED^2 or ED as the reference configuration.

We choose these two metrics as they represent a simplified form of a more general cost function for hardware circuit optimization:

$$F_c = (E/E_0) \times (D/D_0)^{\eta} \qquad 0 \le \eta < +\infty \qquad (4.1)$$

where D is the critical path delay through the circuit, E is the energy dissipation per cycle, D_0 and E_0 are the lower bound values that can be achieved through the logic and circuit fine tuning for a fix supply voltage. The parameter η (Eta) is a notion of hardware intensity, which can relate to power supply in energy-efficient designs [61]. In this study we consider $\eta = 2$ for high performance processors and $\eta = 1$ for in-order embedded processors.

Consistent with previous work [61], we choose this particular form of the cost function (4.1) because its relative delay changes to relative energy changes has the property:

$$\frac{\partial F_c}{\partial D} \Big/ \frac{\partial F_c}{\partial E} = \eta \frac{E}{D} \qquad (4.2)$$

which is a common language in circuit and architecture communities and is used in previous research works [61].

We first study energy budget using the ED^2 metric:

$$E_{total_new} \times D^2_{total_new} = E_{total_ref} \times D^2_{total_ref} \tag{4.3}$$

We first study the data cache. We assume that the total energy (E_{total}) is the sum of the energy spent in the L1 data cache (E_{Data_cache}) and the energy spent in the remainder of the processor $E_{remainder}$. After substituting this into (4.3) we have:

$$E_{Data_cache_budget} = \frac{(E_{remainder_ref} + E_{Data_cache_ref}) \times D^2_{total_ref}}{D^2_{total_new}} - E_{remainder_new} \tag{4.4-a}$$

$E_{Data_cache_budget}$ is the energy available to the new cache to consume while maintaining the same ED^2. It should be noted that if the new cache's energy consumption exceeds the energy budget, the new cache would not be energy efficient as it would not maintain ED^2.

Applying the same calculations to the L1 instruction cache (E_{Inst_cache}), instruction cache budget could be estimated using the following:

$$E_{Inst_cache_budget} = \frac{(E_{remainder_ref} + E_{Inst_cache_ref}) \times D^2_{total_ref}}{D^2_{total_new}} - E_{remainder_new} \tag{4.4-b}$$

Using the ED metric, equation 4.4 can be rewritten as:

$$E_{Data_cache_budget} = \frac{(E_{remainder_ref} + E_{Data_cache_ref}) \times D_{total_ref}}{D_{total_new}} - E_{remainder_new} \tag{4.5-a}$$

and

$$E_{Inst_cache_budget} = \frac{(E_{remainder_ref} + E_{Inst_cache_ref}) \times D_{total_ref}}{D_{total_new}} - E_{remainder_new} \tag{4.5-b}$$

Estimating the cache energy budget in the presence of an energy model is feasible using simulation tools such as simplescalar [9] and wattch [8]. In the absence of a cache power model, (*e.g.,* when modeling a perfect cache) accurate estimation of $E_{remainder_new}$ is

impossible. Under such circumstances (and similar to previous study [37]), we assume that the average energy per cycle used by the non-cache portion of the reference processor is equal to the average energy spent per cycle in the non-cache portion of the new processor:

$$\frac{E_{remainder_ref}}{D_{total_ref}} = \frac{E_{remainder_new}}{D_{total_new}} \tag{4.6}$$

The validity of this assumption is explored in section 4.3. Substituting (4.6) into (4.4) and (4.5), we will have:

Assuming the ED^2 metric

$$E_{Data_cache_budget} = \frac{(E_{remainder_ref} + E_{Cache_ref}) \times D^2_{total_ref}}{D^2_{total_new}} - \frac{E_{remainder_ref} \times D_{total_new}}{D_{total_ref}} \tag{4.7-a}$$

$$E_{Inst_cache_budget} = \frac{(E_{remainder_ref} + E_{Cache_ref}) \times D^2_{total_ref}}{D^2_{total_new}} - \frac{E_{remainder_ref} \times D_{total_new}}{D_{total_ref}} \tag{4.7-b}$$

Rewriting for ED metric:

$$E_{Data_cache_budget} = \frac{(E_{remainder_ref} + E_{Cache_ref}) \times D_{total_ref}}{D_{total_new}} - \frac{E_{remainder_ref} \times D_{total_new}}{D_{total_ref}} \tag{4.8-a}$$

$$E_{Inst_cache_budget} = \frac{(E_{remainder_ref} + E_{Cache_ref}) \times D_{total_ref}}{D_{total_new}} - \frac{E_{remainder_ref} \times D_{total_new}}{D_{total_ref}} \tag{4.8-b}$$

4.3 High-Performance Processors

In table 4.1, we present the configurations of two high performance processors used in this study. The table also includes cache organizations used by each processor. We used a modified version of simplescalar [9] and wattch [8] simulators to estimate performance and energy. We used a suite of integer and floating point benchmarks from SPEC2000. The benchmarks are compiled for the alpha instruction set. Simulations run for 500 million instructions.

Table 4.1: Simulated processor configurations

Processor	8-way	4-way
Issue width	8	4
Clock rate	2 GHz	2 GHz
Instruction Cache	64KB 4-way	16KB 2-way
I$ Hit Latency	3	3
Data Cache	64KB 4-way	16KB 2-way
D$ Hit Latency	3	3
TLB entries (I/D)	16/32 4-way	16/32 4-way
L2 cache	1 MB 4-way	256KB 4-way
L2$ Hit Latency	10	10
Branch Predictor	32K x 2/32K x 2/ 32K x 6	4K x 2/4K x 2/ 1K x 13
Issue Window	128	80
Memory Latancy	250	150

To investigate how cache complexity impacts overall processor energy and performance we study a number of alternative cache configurations. We double the number of cache entries, block size and cache associativity to model such alternatives. We also examine a combination of configuration changes. In table 4.2 we report the new cache configurations, their relative size compared to the original cache size and the abbreviations used through this study.

Table 4.2: Cache organizations and their relative size

Cache Configuration	Abbr.	Size	Cache Configuration	Abbr.	Size
Original Cache	C_{orig}	x1	Original Cache	H_{orig}	x1
Doubled Cache Entries	C_s	x2	Halved Cache Entries	H_s	x½
Doubled Block-size Cache	C_b	x2	Halved Block-size Cache	H_b	x½
Doubled Associativity Cache	C_a	x2	Halved Associativity Cache	H_a	x½
Doubled Entries and Associativity	C_{sa}	x4	Halved Entries and Associativity	H_{sa}	x¼
Doubled Entries, Associativity and Block-size	C_{sab}	x8	Halved Entries, Associativity and Block-size	H_{sab}	x⅛

We assume that cache size variations investigated in this study do not impact cache latency. Therefore, for more realistic scenarios where larger caches have longer latencies, the reported energy budget and energy consumption could be viewed as budget upper bound and consumption lower bound.

Remainder Energy Estimation

In figure 4.1 we have evaluated equation (4.6) accuracy by predicting the remainder energy for a processor with new configuration ($E_{remainder_new}$) and comparing the estimated energy over delay in the new configuration with the reference configuration. Figure 4.1 shows the results for data and instruction caches when we estimate the remainder energy over delay for SPEC2000 benchmarks run on a 4-way processor.

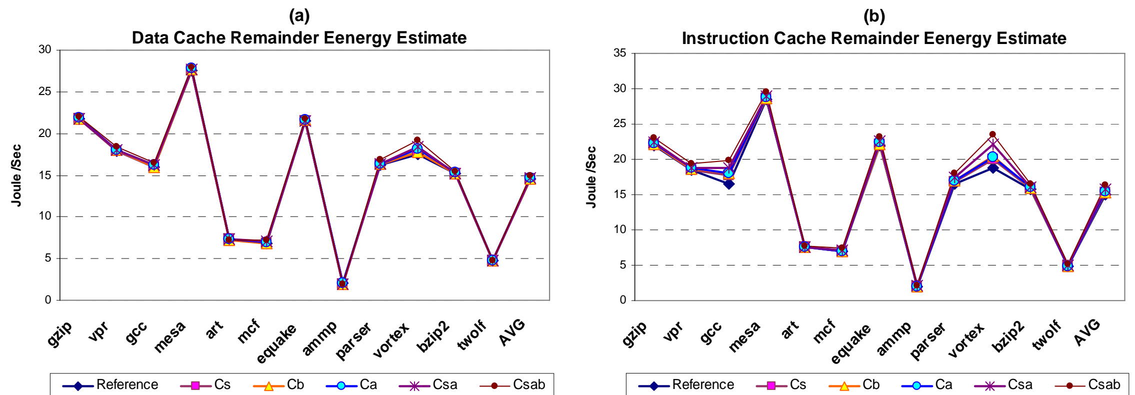

Figure 4.1: Remainder energy over delay prediction for alternative (a) data cache and (b) instruction cache configurations using equation (4.6).

As shown in the figure the difference is higher for Instruction caches (especially when the changes are quite significant *e.g. Csab*) but the closeness of energy over delay in different configurations is consistent with equation (4.6).

Results

We investigate L1 data cache energy efficiency for the 4- and 8-way processors reported in table 4.1. In section 4.3.2 we repeat our evaluation for instruction cache configurations. In each section, we analyze different aspects of the cache energy dissipation in a separate subsection. We study energy efficiency for alternative cache configurations and report the energy budget for each alternative. We compare the estimated cache energy consumptions with energy budgets. We also estimate the upper bound for cache energy consumption. We estimate the maximum cache energy budget available to different processors under an ideal data cache with zero miss rate and no hit penalty. We investigate cache latency break-even point. We compare a realistic cache with a non-realistic cache with zero miss rate and non-zero hit penalties.

4.3.1 L1 Data Cache: Energy Analysis

Using equations (4.4-a) and (4.4-b), we calculate $E_{Data_cache\text{-}budget}$ for new cache configurations and report the results using ED^2. For each processor, the original cache configuration, reported in table 4.1, is presented as $Config_{ref}$. We also examine different cache structures listed in table 4.2.

Energy Budget

Figure 4.2 reports energy budget (per application run time) per benchmarks for each processor configuration. Different curves represent different cache structures introduced in table 4.2. The calculated energy budget varies from processor to processor and from one application to another. The figure shows that in most cases a processor with a cache having twice number of cache entries, twice of a block size and twice associativity, *Csab*, has the highest cache energy budget. This is due to the fact that *Csab* provides better performance improvements compared to other configurations.

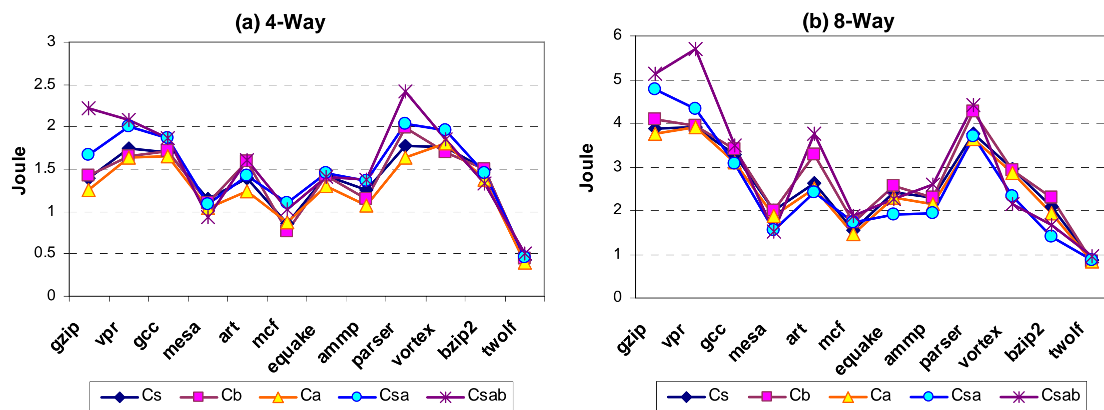

Figure 4.2: Energy budgets for alternative data cache configurations using the ED^2 metric.

Increasing the cache structure size does not always result in a higher energy budget. In figure 4.2(b) the 8-way processor, for example, a processor with twice number of cache entries and assoicativity, *Csa, has* a lower budget for almost all benchmarks when compared to *Cb.* Accordingly, for this processor, replacing *Cb* with *Csa*, could increase energy excessively without improving performance considerably.

We witness larger energy budgets for *gzip, vpr, vortex* and *parser*. Our study shows that for all these benchmarks, the overall performance is more sensitive to cache organization compared to others. Some applications (*e.g.*, *ammp* and *art*) do not benefit

from larger caches. This could be explained by the large number of compulsory cache missies for these applications. Note that by exploiting more complex caches we reduce the number of conflict and capacity misses but cannot remove compulsory misses.

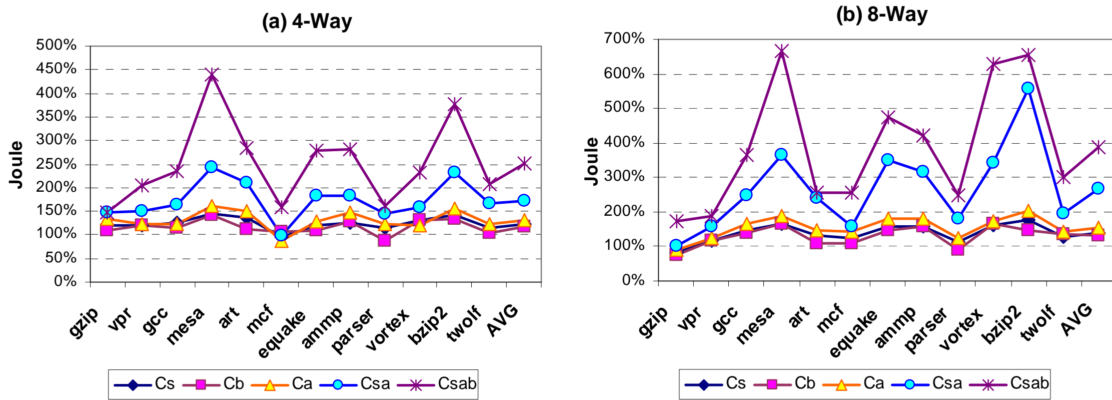

Figure 4.3: Relative data cache run time energy consumption compared to the energy budget per application run time for each application and cache organization. (The 100% line shows the energy budget limit)

While figure 4.2 reports the energy budget available to each processor's data cache configuration it does not report energy consumption. In figure 4.3 we report the percentage of energy budget consumed by each of the cache organizations. The 100% line represents the calculated break-even energy budget for each cache structure. Energy consumptions exceeding this line indicate too much energy consumption, *i.e.,* intuitively, the energy spent is not worth the performance improvement gained. Under such circumstances, the reference cache structure is a better choice. As presented, none of the cache enlargements remain within the cache energy budget limit. The only acceptable cache enhancement is *Cb* (where block size is doubled) for *gzip* and *parser* in the 8-way and *parser* for the 4-way processor. For both processors and all applications *Csab* exceeds the energy budget limit.

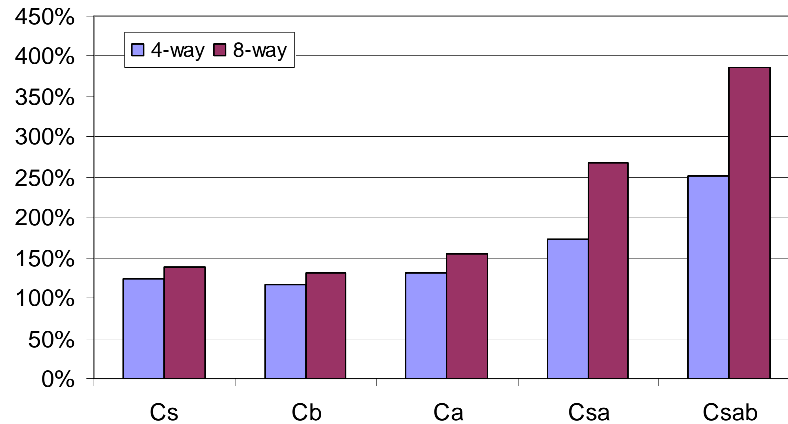

Figure 4.4: Average percentage of L1 data cache run time energy consumption compared to the energy budget per application run time for each processor. (The 100% line shows the energy-budget limit.)

In figure 4.4, we report the average share of energy budget consumed by each cache structure in each processor. As reported exploiting complex caches does not payoff since the average energy consumption exceeds the energy budget limit.

Data Cache Energy Analysis for Smaller Data Caches

In this section we investigate how using smaller caches impacts energy efficiency for the processors studied in this work. While using a smaller cache results in less energy directly consumed by the cache it could increase the overall energy consumption due to possible increases in the cache miss rate. This in turn could result in an increase in the number of accesses to higher levels of memory hierarchy and therefore the overall energy consumption.

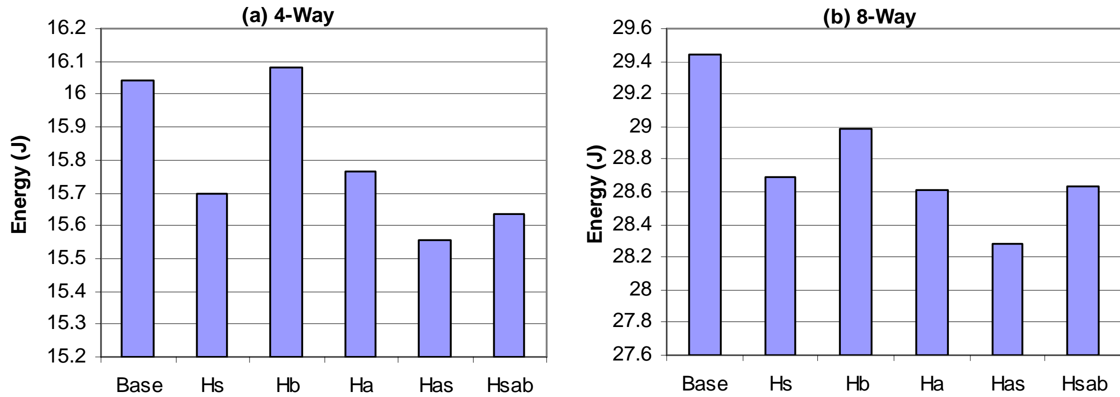

Figure 4.5: Total processor energy consumption using reduced sized L1 data cache configurations.

We use the smaller cache configurations reported in table 4.2. As reported in figure 4.5, often and except for halved sized block size in the 4-way processor, smaller cache configurations result in less total energy consumption. In other words, energy savings achieved by using more simple cache structures often exceeds the extra energy caused by possible increases in data cache miss rate.

Data Cache Energy Budget Upper Bound

To provide better insight we study an ideal data cache and report energy budget upper bound for the processors used in this study. The simulated ideal cache represents a non-realistic cache with zero cache miss and no hit latency. Because there is no power model for an ideal cache, we estimated the $E_{reminder\text{-}new}$ using formula (4.7-a).

In figure 4.6, we report the maximum energy budget calculated for each processor configuration using an ideal cache. The lower part of each bar represents the original processor's cache energy consumption as a portion of the maximum energy budget. The upper part of each bar indicates how much more energy could be invested in the cache to

achieve an ideal cache. On average, both processors spend about 8% of their maximum energy budget.

Among all applications, *art* and *ammp* can use more energy to achieve an ideal cache. Note that in the ideal cache modeled here we assume that all cache misses (including compulsory misses) are removed. Therefore applications such as *ammp* and *art* with large data footprints can potentially benefit more than others by using an ideal cache.

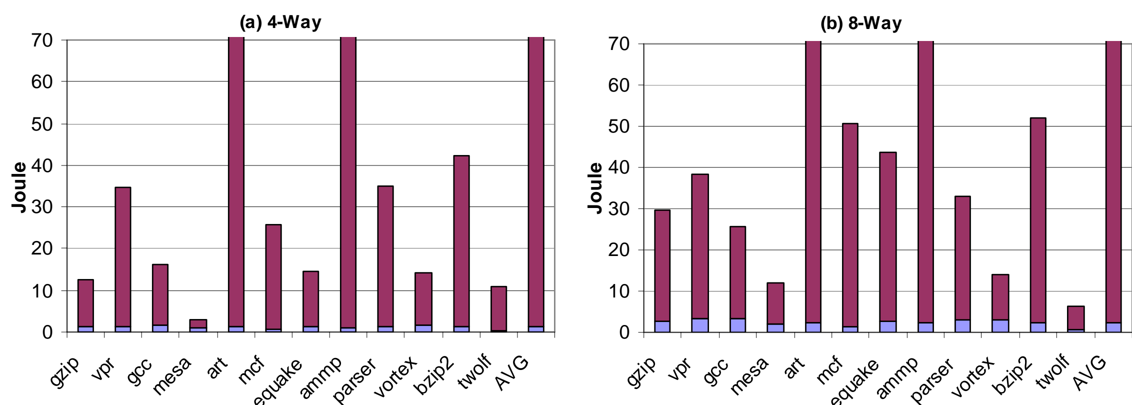

Figure 4.6: The entire bar reports energy budget available to an ideal data cache for 4-way and 8-way processors. The lower part of each bar shows L1 data cache energy consumption for the reference cache. (Values more than 70 joules are not reported.)

Latency Break-even Point

Improving hit rate could increase hit penalty as it may require exploiting larger and more complex caches. While achieving higher hit rates is desirable, it is important to maintain low hit penalty. To provide better insight, we investigate latency break-even points for the processors studied here and for an ideal data cache.

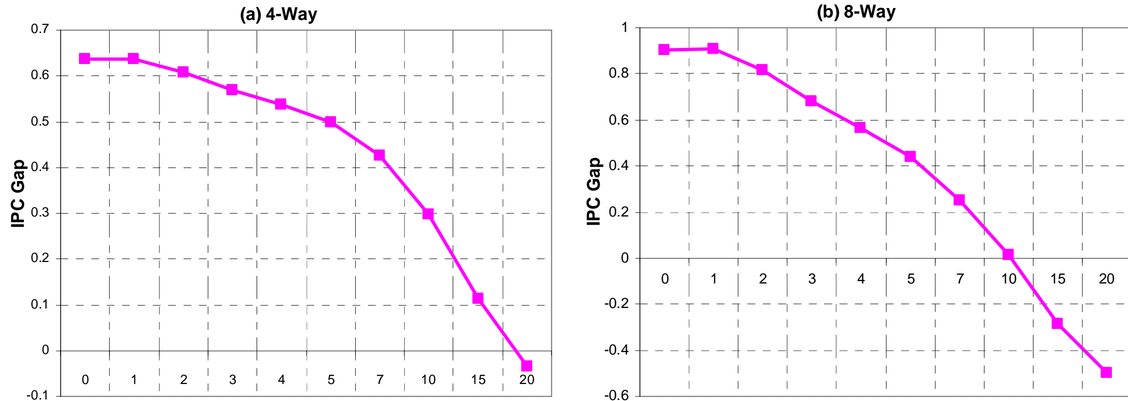

Figure 4.7: Hit latency impact on performance gap between non-realistic and realistic data caches.

In figure 4.7 we report the performance gap between a non-realistic cache with zero miss rate and a processor using the original non-ideal cache for the applications studied in this work. For the non-realistic processor we assume an L1 data cache with zero miss rate but different non-zero hit latencies.

The latency break-even point is the point (or hit penalty) where the gap reaches zero, i.e., a processor using a cache with zero miss rate performs similar to the processor using a realistic cache (i.e., a processor with a lower hit rate but a smaller hit penalty). Achieving zero miss rate is only justifiable if the associated hit penalty stays below the break-even point. The 4-way processor has a break-even point of 20 cycles while the 8-way processor has a break-even point of 10 cycles.

4.3.2 L1 Instruction Cache: Energy Analysis

Energy Budget

In figure 4.8 we report energy budget for the instruction cache using equation (4.4-b). As presented, for both processors, except for gcc and vortex, Csab (where block size, the

number of entries and cache associativity are doubled) has the lowest energy budget. This is different from the results obtained studying the data cache. Accordingly, for all applications but gcc and vortex, using much larger instruction cache structures (i.e., Csab) does not improve performance considerably but results in a substantial energy increase. Our study shows that both gcc and vortex will considerably benefit from larger instruction caches as their miss rate shows a greater drop compared to other applications.

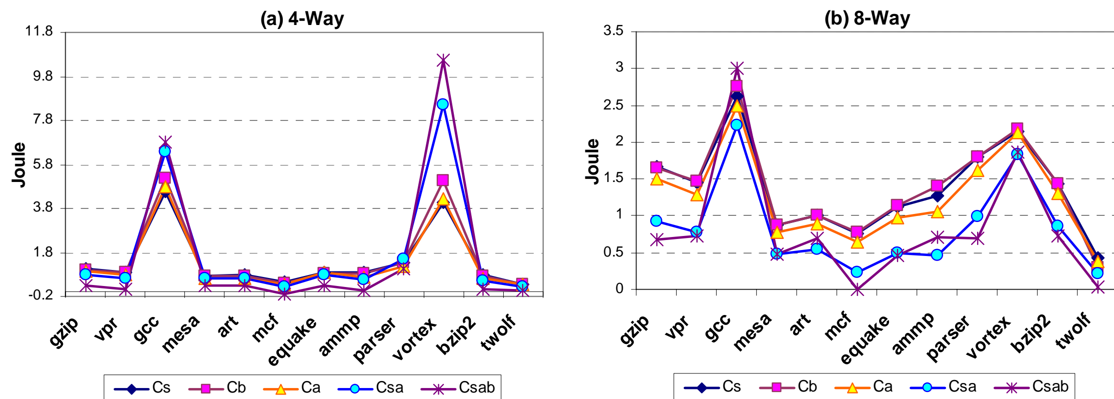

Figure 4.8: Energy budgets per application run time for alternative instruction cache configurations using the ED^2 metric.

In figure 4.8 (a) the 4-way modeled processor, for *mcf*, we witness a negative energy budget. Our study shows that for *mcf* the original instruction cache miss rate is very small (almost zero) and enlarging the instruction cache does not make the cache performance any better. Consequently, the performance improvement achieved by doubling the cache features (*Csab*) is not worth the extra energy. Accordingly our negative estimate indicates energy overinvestment in the instruction cache for *mcf*.

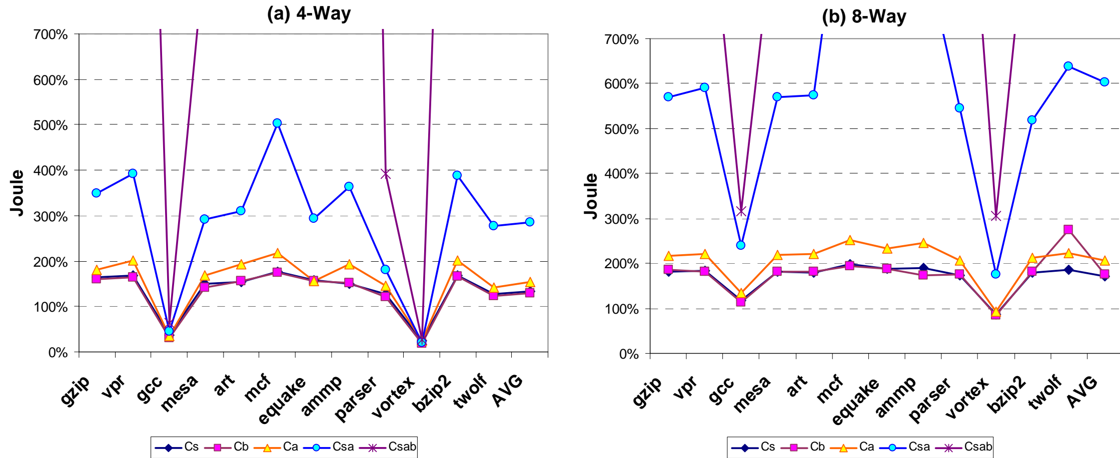

Figure 4.9: Relative instruction cache run time energy consumption compared to the energy budget per application run time for each application and cache organization. The 100% line shows the energy budget limit. (Values more than 700% are not reported.)

Note that *mcf* has a positive energy budget for the 8-way processor. This is due to the fact that in the 8-way processor, a more aggressive approach is exploited effectively increasing the number of instruction cache accesses for this application. As a result, the overall performance shows higher sensitivity to instruction cache size. In other words, an 8-way processor has enough number of back-end resources to take advantage of a more complex cache while the 4-way processor does not.

Figure 4.9 shows the percentage of the energy budget consumed by different instruction caches for different benchmarks and for each processor. The 100% line represents the energy budget limit based on the ED^2 term. Among all applications, *gcc* and *vortex* are the only one benefiting from more complex caches for the 4-way processor. The large values for relative energy consumption in figure 4.9 indicates that exploiting more complex cache structures for the application is not worthwhile. Average energy consumption compared to the available energy budget for each processor is reported in figure 4.10.

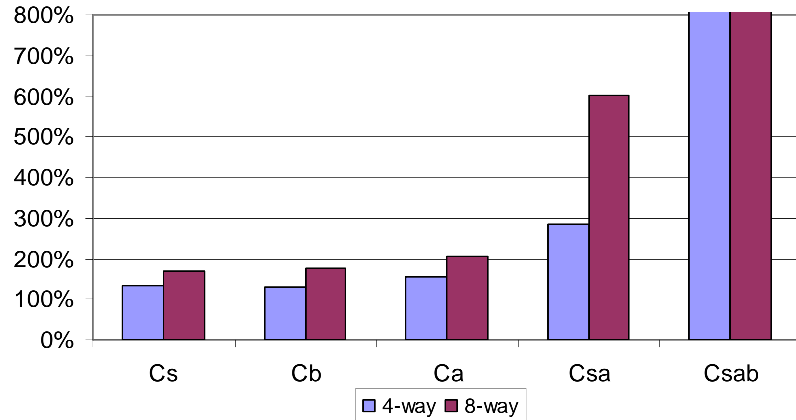

Figure 4.10: Average percentage of L1 instruction cache run time energy consumption compared to the energy budget per application run time for each processor. The 100% line shows the energy-budget limit (Values more than 800% are not reported.)

Cache Energy Analysis for Smaller Caches

Figure 4.11 shows overall processor energy consumption if smaller instruction caches (*i.e.,* organizations reported table 4.2) are used. Unlike results reported for data cache, with the exception of using the half size cache entries (*Hs*) and half size associativity (*Ha*) and their combination (*Hsa*) in the 4-way processor, processors using smaller cache configurations consume more overall energy.

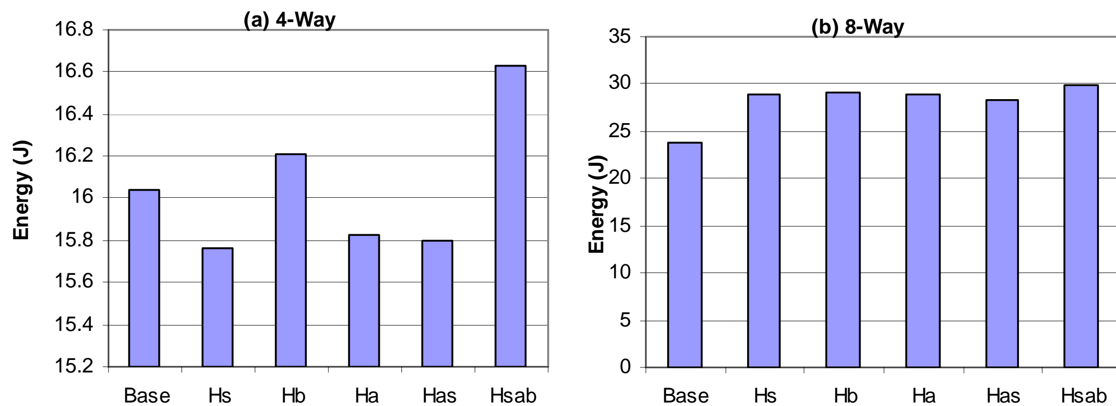

Figure 4.11: Total processor energy consumption using reduced sized L1 instruction cache configurations.

Cache Energy Budget Upper Bound

In figure 4.12 we report instruction cache energy upper bound estimated using equation (4.7-b) for an ideal cache. The estimated upper limit for energy budget is lower compared the upper limit estimated for the data cache. On average, instruction cache energy consumption is about 70% and 80% of the instruction cache upper energy limit for 4-way and 8-way processors respectively. This energy budget leftover for such cache structures, highly constrains future enhancements.

Note that the estimated upper energy limit for *gcc* and *vortex* is much higher compared to other applications for both processors. *Gcc* and *vortex* use the smallest percentage of their upper limit. The upper limit for these two applications is too high to report. The instruction cache miss rate for *gcc* and *vortex* improves significantly by using a more complex cache structure. Note that in the case of *art,* the reference cache's energy consumption matches the upper bound.

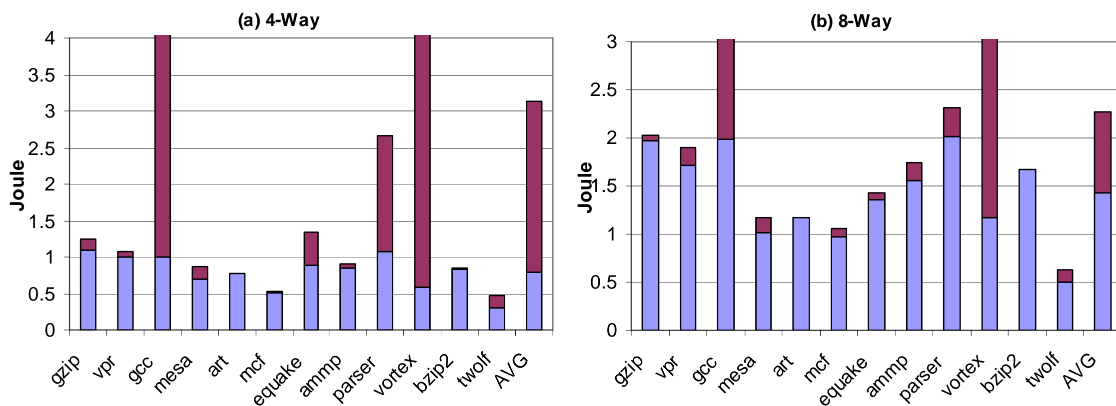

Figure 4.12: The entire bar reports energy budget per application run time available to an ideal instruction cache for 4-way and 8-way processors. The lower part of each bar shows the L1 instruction cache run time energy consumption for the reference cache. (The values more than 4 (part a) and 3 (part b) joules are not shown)

Latency Break-even Point

In figure 4.13 we report average performance gap between a processor using an ideal (i.e., with zero miss rate) instruction cache with different hit latencies and one using the original instruction cache for the applications used in this study.

The break-even point for both processors is below three cycles indicating that any effort in reducing the miss rate should not result in an increase in the hit latency.

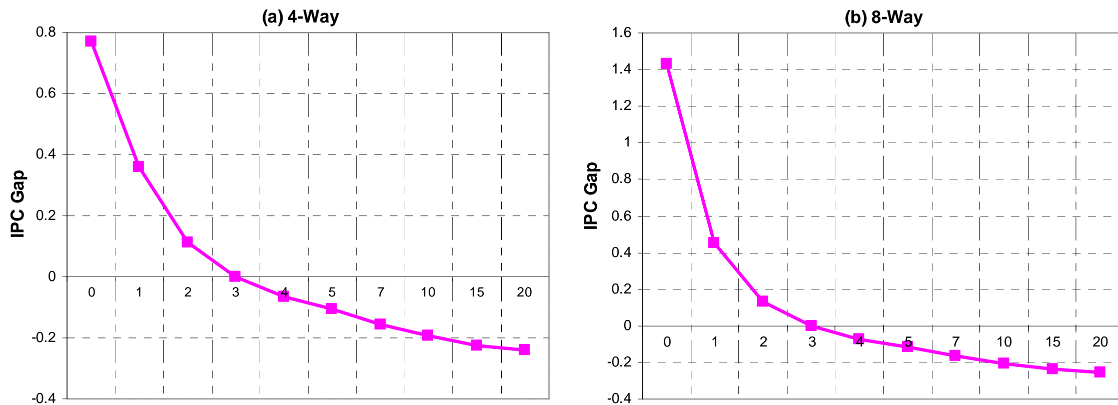

Figure 4.13: Hit latency impact on performance gap between non-realistic and realistic instruction caches

4.4 Embedded Processors

In table 4.3, we present the Xscale-like embedded processor configuration used in this study. We also report cache organizations used by each processor and assume that changing the cache configuration would not impact other processor components' designs. We also optimistically assume that variations in cache size do not impact cache latency. Therefore for more realistic scenarios where larger caches have longer latencies, the reported budget and energy consumption could be viewed as budget upper bound and consumption lower bound.

We used a modified version of Xtrem [19] simulator to estimate performance and energy. We used a subset of MiBench applications [42]. The benchmarks are compiled for the Arm instruction set. We simulated the complete benchmark or half a billion instructions, whichever comes earlier.

Table 4.3: Processor configuration used in this study

Processor Core	
Core Frequency	200 MHz
Miss Pred. Penalty	3 cycle
Write Buffer	8 entries
Functional Units	1 Int ALU, 1 Int mult/div 1 FP ALU, 1 FP mlt/div, 1 mem prt
Memory Hierarchy	
L1 D-cache Size	32 KB, 32-way, 32B blocks, wr bk
L1 I-cache Size	32 KB, 32-way, 32B blocks, wr bk
L1 latency	1 cycle
L2	N/A
Memory latency	36 cycles
D-TLB/I-TLB Size	32/16-entry, fully assoc., 170-cycle miss
Branch Prediction	
BTB	128-entry, 1-way
Direction Predictor	bimodal predictor, 128 entries
Return-address-stack	N/A

To investigate how cache complexity impacts overall processor energy efficiency we study a number of alternative cache configurations. As we reported earlier in table 4.2 we double the number of cache entries, block size and cache associativity to model such alternatives. We also examine a combination of such configuration changes.

Results

We evaluate the energy and latency budget for both instruction cache and data cache. Unlike previous study on high performance processors, we present data and Instruction cache evaluation in the same subsection. In Section 4.4.1, we study energy efficiency for alternative cache configurations. We report the energy budget for each alternative and compare cache energy consumptions with energy budgets. Next, we estimate the upper bound for cache energy consumption. We estimate the maximum cache energy budget available for different benchmarks for an ideal data cache with zero miss rate and no hit penalty. In Section 4.4.2, we investigate cache latency break-even point for both data and instruction caches. We compare a realistic cache with a non-realistic cache with zero miss rate and non-zero hit penalties.

4.4.1 L1 Data and Instruction Cache: Energy Analysis

Figure 4.14 reports energy budget per benchmark for different instruction and data cache configurations. Different curves represent different cache configurations introduced in table 4.2. Comparing figures 4.14(a) and 4.14(b) shows that the budget available to the instruction cache is generally lower than that available to the data cache. The calculated

energy budget varies from instruction cache to data cache and from one application to another. For the data cache, often the processor with a cache having twice the number of cache entries, twice of a block size and twice associativity, *Csab*, has the highest cache energy budget. This could be due to the fact that *Csab* improves performance by reducing capacity, compulsory and conflict misses simultaneously resulting in higher delay reduction compared to other configurations.

Increasing the cache size does not always result in a higher energy budget. For example, as presented in figure 4.14 (a) a processor with twice the number of cache entries, twice of a block size and assoicativity, *Csab,* has a lower budget for all benchmarks when compared to *Ca.* Accordingly, for this processor, replacing *Ca* with *Csab*, could increase energy excessively without improving performance considerably.

In figure 4.14 (a), for all applications with the exception of *lame*, changing the cache configuration has little impact on data cache energy budget.

We witness larger energy budgets for *lame* and *crc* in data cache energy budget. Our study shows that for both of these benchmarks, the overall performance is more sensitive to cache organization compared to others. Many applications (*e.g.*, *adpcm_c* and *basicmath*) do not benefit from larger caches. This might be explained by the small data footprint for these applications.

As presented in figure 4.14 (b), and similar to 4.14 (a), *lame* has a higher energy budget compared to other applications. In general, however, variations in cache configuration make a bigger impact on instruction energy budget compared to data cache energy budget.

Csa (where the number of entries and cache associativity are doubled), *Cs* and *Ca* have the highest energy budget for almost all applications. Accordingly doubling the number of entries and associativity has a greater impact on the instruction cache. This is different from the results obtained studying the data cache. In the case of the instruction cache, using larger instruction cache structures (*i.e.*, Csab) often does not improve performance considerably but results in a substantial energy increase.

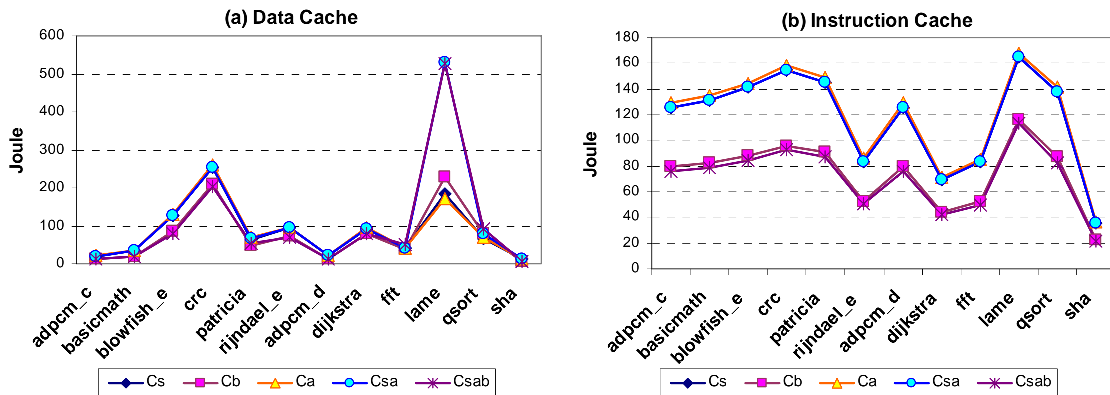

Figure 4.14: Energy budgets for alternative data cache and instruction cache configurations.

While figure 4.14 reports the energy budget available to each data/instruction cache configuration, it does not report energy consumption. In figure 4.15 we report the percentage of energy budget consumed by each of the cache organizations. The 100% line represents the calculated break-even energy budget for each cache structure. Energy consumptions exceeding this line show that the energy spent is not worth the performance improvement gained. Under such circumstances, the reference cache structure is a better choice. As presented, unacceptable cache enhancements include *Cb* and *Csab* (for *crc, rijndael_e* and *blowfish_e*) for the instruction cache. In the case of the data cache, *Csab* and *Cb* exceed the energy budget limit for all applications.

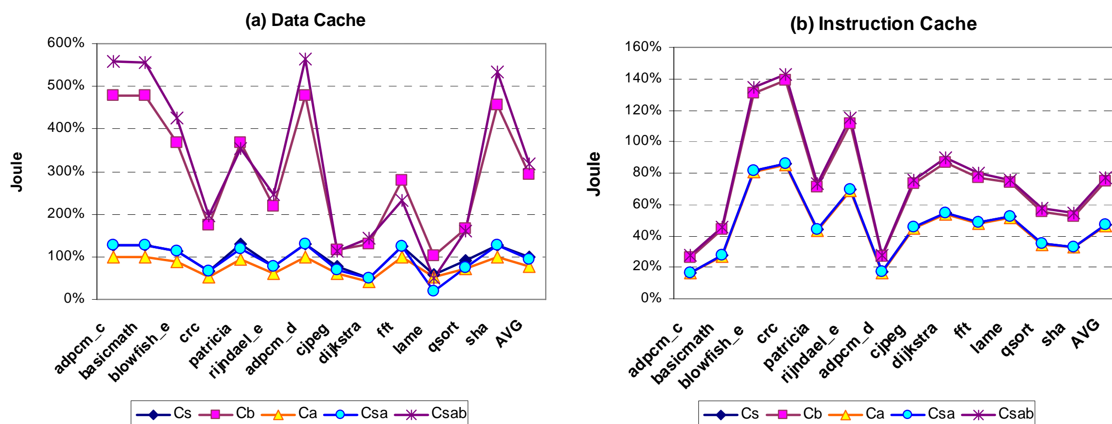

Figure 4.15: Relative data and instruction cache energy consumption compared to the energy budget for each application and cache organization. (The 100% line shows the energy budget limit.)

In figure 4.16, we report the average share of energy budget consumed by each cache configuration. As reported exploiting complex instruction caches pays off as the average energy consumption is within energy budget limit. However, for the data cache *Cb* and *Csab* exceed the limit indicating that applications would not always benefit from more complex data caches.

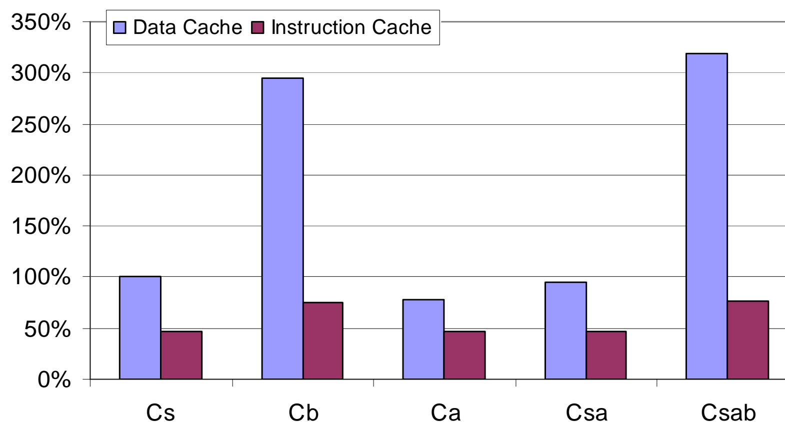

Figure 4.16: Average percentage of L1 data and instruction cache energy consumption compared to the energy-budget for selected processor. (The 100% line shows the energy budget limit.)

L1 Cache Energy Budget Upper Bound

Using the equations (4.8-a) and (4.8-b) we repeat our calculation to estimate the energy budget upper bound.

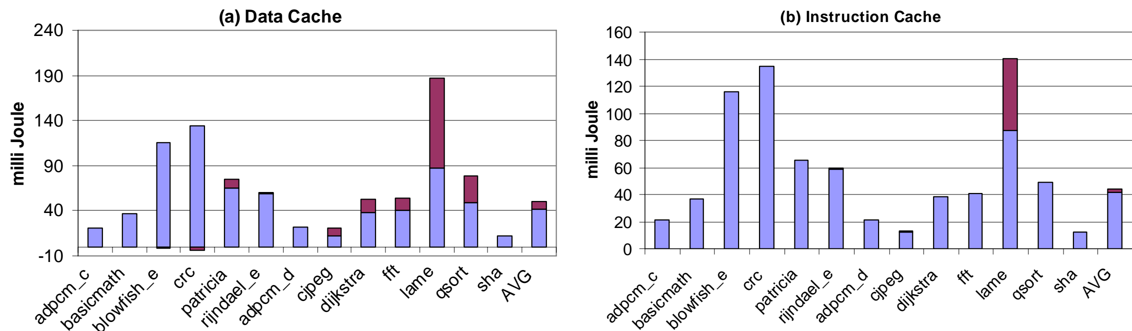

Figure 4.17: The entire bar reports energy budget available to an ideal cache. The lower part of each bar shows the cache energy consumption for the reference cache. (Values more than 70 mjoules are not reported.)

Figure 4.17, reports the maximum energy budget calculated for ideal data and instruction caches when used in an XScale like processor. The lower part of each bar represents the original processor's cache energy consumption where as the upper part of each bar indicates the amount of energy which could be invested in the cache to achieve an ideal cache. On average, data and instruction cache spend 85% and 96% of their maximum energy budget respectively. The reference data cache consumes more energy than the upper bound energy for *blowfish_e* and *crc*. For these applications, the energy already spent on the reference cache (even if it was work as perfect as an ideal cache) seems to be too much. The difference is presented by a negative bar in figure 4.17 (a).

Among all applications, *lame* can use more energy to achieve an ideal cache. Note that in the ideal cache modeled here we assume that all cache misses (including compulsory misses) are eliminated. Therefore applications such as *lame* with large data footprints can potentially benefit more than others by using an ideal cache.

Latency Break-even Point

We repeat our technique used in previous section 4.2, to find the Latency break-even points in an embedded processor architecture. In figure 4.18 we report the performance gap between a non-realistic cache with zero miss rate and a processor using the original non-ideal cache for the applications studied in this work. As depicted in figure, the data cache has a break-even point of about 10 cycles while the instruction cache has a break-even point of 1 cycle.

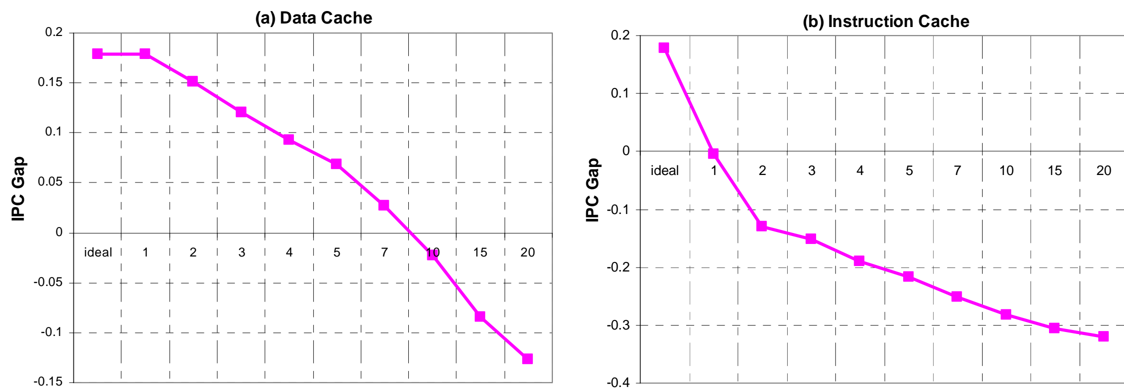

Figure 4.18: Hit latency impact on performance gap between non-realistic and realistic data and instruction caches.

Chapter 5

Reducing Non-optimal LRU Decisions in Chip Multiprocessors

This chapter explores cache memory management issues in chip multiprocessors and employs a speculative technique to bridge the performance gap between the commonly used Least Recently Used (LRU) replacement algorithm and the optimal replacement policy. We evaluate the non-optimal decisions (NODs) made by the LRU algorithm and provide a taxonomy of mistakes to identify and avoid similar decisions in future incidents.

We employ a history based algorithm which records the past cache behavior, and speculate on future replacements. The technique's effectiveness (accuracy and coverage) depends on the hardware budget; however, with a moderate overhead our suggested replacement algorithm eliminates a significant amount of NODs resulting up to 9% improvement on average in cache hit rate.

The preliminary results of this work were presented as a poster presentation at the thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08). The final results have been submitted for publication to the PACT 2008 Workshop on MEmory performance:DEaling with Applications, systems and architecture, and to be send for publication to Elsevier's Journal of Microprocessors and Microsystems.

5.1 Introduction

Chip multiprocessors continue to demonstrate performance improvements, largely as a result of innovative techniques introduced at architectural and VLSI levels. The speed and scalability of such architectures greatly depend on the performance of the memory hierarchy. Despite advances, memory speed has improved at an insufficient pace, continuing to serve as a bottleneck. The growing gap between processor and memory performance has made cache misses increasingly expensive. Moreover, despite ongoing research, existing non-optimal replacement policies result in too many cache misses.

Least Recently Used (LRU) is a widely used replacement policy as it offers simplicity and relatively acceptable performance. However, previous studies have shown that in highly associative caches there is a considerable performance gap between LRU and Belady's theoretical optimal replacement policy [45]. This motivates investigating new replacement solutions that address the shortcomings associated with the conventional LRU.

One of LRU's well-known inefficiencies is its inability to cope with some access patterns, which manifests in two common scenarios. The first scenario occurs as a result of late cache block evictions. In a 4-way associative cache for example, it takes up to three evictions (depending on the access pattern) before conventional LRU moves the most recently used (MRU) block to the LRU position. Assuming that the block is not accessed during this period, this late eviction could result in performance loss, as the block continues to occupy cache space without contributing to performance. Consistent with previous studies [39] [2], we refer to these blocks as *Dead Blocks* (or *D-Blocks*). In an optimal scenario, D-Blocks should be replaced as soon as they leave the MRU

position. One particular group of D-Blocks consists of those which are only accessed once before leaving the cache. This group is referred to as *Bypass Blocks* (or *B-Blocks*). The second scenario occurs as a result of early cache block evictions where LRU, while keeping less important blocks, replaces a cache block that will be needed shortly. We refer to these blocks as *Live Blocks* (or *L-Blocks*). L-Blocks should not be evicted as long as better candidates exist.

Previous work has investigated non-optimal LRU decisions from different perspectives. Gonzales *et al.* lowered the impact of non-optimal decisions made by LRU by splitting the cache to separate spatial and temporal caches [7], and used speculative bypassing to avoid access patterns which are likely to pollute the cache. Lai *et al.* [2] proposed dead block prediction to identify dead blocks and downgrade them faster, while Tyson *et al*. [18] identified dead blocks that will never be inserted in the cache again. Johnson and Hwu [60] used a counter based approach to bypass low reuse cache blocks at runtime. Kampe *et al.,* studied live blocks, using a shadow directory [39] and a mistake history table to identify live, dead and bypass blocks, and changed the victim selection policy accordingly.

The goal of this work is to extend previous work and investigate LRU's non-optimal decisions (here referred to as *NOD*s) further. We are motivated by our observation that while dead and live blocks account for a significant portion of NODs, it is possible to extend their definition and introduce more inclusive concepts. In particular we show that by extending the definition of a live block and taking into account a series of consecutive replacements, we can identify a larger group of NODs referred to as *Hasty Blocks* (or H-Blocks). By postponing the eviction of these blocks we reduce cache miss

frequency. We also take advantage of the predictable access behaviors exhibited by some cache blocks and extend the concept of dead blocks to introduce *Predictable Blocks* (or P-Blocks). By identifying P-Blocks we reduce the cache miss rate by taking a just-in-time approach. Our optimized LRU replaces these blocks immediately after their final cache access.

In section 5.2, we study H-Blocks in more detail and describe how we identify them. We look at P-Blocks and their detection algorithm in section 5.3. In section 5.4 we introduce our Speculative history based cache Replacement Algorithm (SRA) to avoid NODs. In section 5.5, we describe our methodology, while section 5.6 presents our experimental results for CMPs. In section 5.7, we consider implementation issues and introduce the Modified LRU-style History Table (MLHT) as a minimal hardware solution to implement our replacement technique. In section 5.8 we exploit MLHT in SRA technique and report for a more comprehensive NOD reduction.

5.2 H-Blocks

Early eviction of cache blocks, while they are likely to be referenced again, could harm performance. This is particularly true when better eviction candidates exist, e.g., cache blocks not being referenced till their eviction. Such early replacements are more frequent in the presence of high temporal locality, which is more visible at the first level of cache. Therefore, in this work we focus on the L1 data cache. Before we explain about H-Blocks, we describe an illustration which is often used in this work to show the order of replacements and the cache set status after replacement decisions.

Cache Status Representation

In figure 5.1 we present a simple LRU replacement example. Each row in figure 5.1 (a) shows the status of a cache set after replacement. The cache associativity is four and letters represent the value of different cache blocks in the set. The cache blocks enter the cache set from the Most Recently Used (MRU) position and will be evicted from LRU position. Note that this representation does not necessarily mean LRU replacement's implementation are working in the same fashion.

Requested	MRU Set Blocks LRU	Evicted Blocks
-	D C B A	
E	E D C B	A
C	C E D B	A
D	D C E B	A
F	F D C E	B A
D	D F C E	B A
G	G D F C	E B A

(a)

Requested	MRU Set Blocks LRU	Evicted Blocks
-	D C B A	
E	E D C B	A
F	F D* C* E	B A
G	G D* F C	E B A

(b)

Figure 5.1: (a) Simple example of LRU replacement policy. Each row shows an access to the imaginary cache set presented in this illustration. (b) Same example illustrated with asterisk notation to save space.

If a cache block is accessed between two replacements it will be moved to the MRU position. The evicted blocks are kept in the same order they are evicted. To save illustration space, we only present the replacements in the rows. Yet, in order to keep the information about the accessed blocks between two replacements, we mark accessed blocks with an asterisk notation. Figure 5.1 (b), shows the same replacement example of figure 5.1 (a) using asterisk notation to save space.

H-Blocks Presentation

H-Blocks are a subset of LRU victim blocks that are referenced often shortly after their eviction. If block 'A' is replaced in the presence of other blocks that are not accessed between block A's replacement and recall, we refer to block 'A' as an H-Block.

To explain H-Blocks better in figure 5.2 we present a simple example. Each row in figure 5.2 (a) shows the status of a cache set after replacement. The dashed line in figure 5.2 (a) separates the two different alternatives in the cache access pattern when block 'A' is recalled. We assume that block 'A' is evicted early in the first row. The third row in figure 5.2 (a) illustrates an example of an H-Block where the evicted block is referenced immediately after replacement. We refer to the evicted block as an H-Block unless all other blocks in the set are referenced between the eviction and the recall of the block (first alternative in figure 5.2). Previous studies [39] have referred to the evicted block as a live block (L-Block) since the block would have still been alive and referenced shortly if it had not been picked as a victim.

Requested	Set Blocks	Evicted Blocks
-	D C B A	
E	E D C B	A
A	A E* D* C*	B A
A	A E* D C	B A

(a)

Requested	Set Blocks	Evicted Blocks
-	D C B A	
E	E D* C B	A
F	F E* D C	B A
A	A F* E D	C B A

(b)

Figure 5.2: (a) Block 'A' is recalled in third row. Due to other blocks access pattern two scenarios can happen in third row; Scenario 1 (above the dashed line): block 'A' is not an H-block since other blocks are referenced after 'A' evicted; Scenario 2 (below the dashed line): underlined block 'A' is a simple example of an H-block since 'D' and 'C' have not been accessed. (b) H-blocks (unlike live blocks) are not restricted to references immediately after eviction.

In Figure 5.2 (b), we show an example to better clarify how an L-Block is different from an H-Block. H-Blocks are not restricted to blocks referenced directly after eviction. If block 'A' is requested again in any of the consequent rows (i.e. row 2, 3, ...)

it still is viewed as an H-Block. However, previous studies consider 'A' as an L-Block only if referenced immediately after eviction (i.e., third row in figure 5.2 (b)). Thus, H-Blocks extend the concept of L-Blocks and offer a more inclusive approach by tracking the evicted blocks for a longer history. The necessary condition to define an H-Block is the presence of at least one inactive block between the H-Block eviction and recall.

5.2.1 H-Block detection implementation

We use a counter based approach to detect hasty replaced blocks in the LRU algorithm. In our method, we associate every cache set with a *replacement counter (RC)*, incremented when a cache block from that set is replaced. In other words, RC indicates the number of replacements made to the associated set in the cache. We also associate each set with a second counter referred to as the *activity counter (AC)*. AC counts the active cache blocks accessed between two replacements. Active blocks are cache blocks which are referenced at least once between the two replacements in the same set. In figure 5.3, active blocks are marked with an asterisk.

AC #	RC #	Req.	Set Blocks	Evicted Blocks
0	0	-	D C B A	
1	1	E	E D* C B	A
1	2	F	F E* D C	B A
1	3	A	A F* E D	C B A

Figure 5.3: AC is reset but RC increments on every replacement

When a replacement occurs, AC resets but RC will keep incrementing. In figure 5.3, we extend figure 5.2 (b) to include AC and RC counter values after each replacement. Assuming that the initial value of RC is zero in the first row, after 'A' is replaced in the second row RC is incremented to 1. The asterisk notation on 'D' shows

that 'D' has been active (accessed) before 'A' is replaced. However, since 'C' and 'B' were not accessed AC is only incremented once. AC is reset in the third row to accurately count the active blocks after the eviction of 'A.'

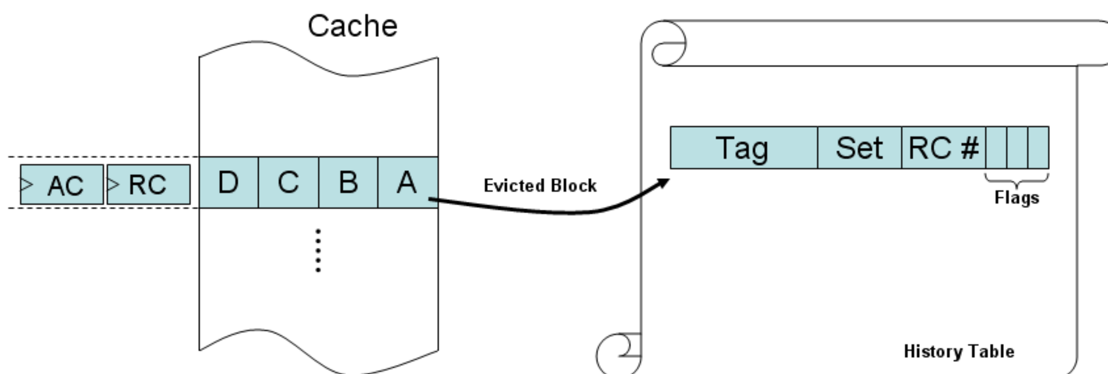

Figure 5.4: History table and the fields recorded for each evicted block

Upon a cache replacement, every evicted block tag is stored in a history table. Each entry in the history table contains the address tag and the set index of the evicted block. It also stores the replacement counter's value (RC) associated with the cache set of the vacated position. The history table entries also reserve two bits to mark a cache block as hasty, dead or bypass (Figure 5.4).

On every cache replacement, the table is accessed to search for both the evicted block and the requested block. If the evicted block exists in the table its RC value will be updated; otherwise the evicted block tag is stored in a new entry. Also, the requested block is looked up in the table. If the requested block exists, its stored RC value in the table is compared to the current set's RC value. As shown in figure 5.5, if the difference is less than the associativity size the block is marked as hasty. The exception is when the set's AC value equals associativity. In this case all blocks inside that particular set were fully active and accessed at least once after the block was evicted. Thus, the block eviction was not hasty and it is not flagged as an H-Block.

The pseudo code in figure 5.5 explains our algorithm for detecting hasty blocks.

```
Search history table linearly
  IF requested block in history table THEN
     IF (set AC != associativity) and ((set RC – stored RC) <
associativity) THEN
        flag request block as hasty block
     ENDIF
  ENDIF

  IF replaced block in history table THEN
     update replaced block RC value
  ELSE
     enter replaced block in history table
  ENDIF
```

Figure 5.5: Our H-Block detection algorithm in simple pseudo code

5.2.2 H-Block and L-Block Comparison

As stated earlier, H-Blocks are more frequent than live blocks. Figure 5.6 shows that on average H-Blocks are about twice as frequent as L-Blocks. We conclude that identifying and eliminating H-Blocks could have a greater impact on cache performance when compared to live blocks.

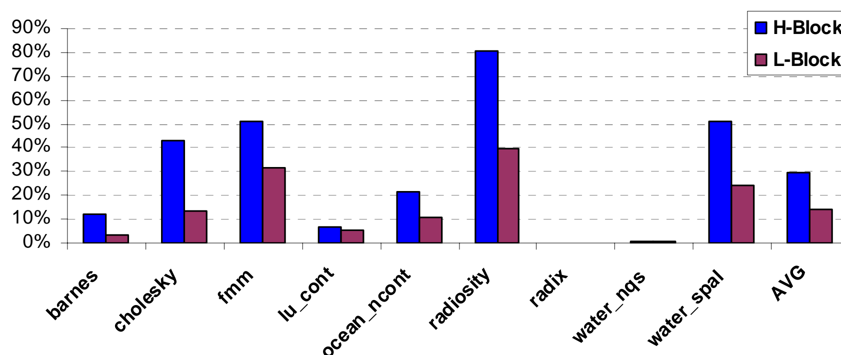

Figure 5.6: H-Blocks and L-Blocks comparison

Note that as reported in figure 5.6, *Radix* has no L- or H-Block. This is because radix has a very low temporal locality compared to other benchmarks. *Water (spatial)* has the highest rate of H- to L-Block frequency and *Barnes* has the highest H-Block frequency.

5.3 P-Blocks

Some cache blocks have regular and predictable access patterns. These patterns make the number of accesses made to the block, while it resides in the cache, highly predictable. We refer to these blocks as *Predictable Blocks* or P-Blocks. When a P-Block is referenced for the last time, it will not be referenced again till it reaches the LRU position. To improve cache efficiency it is important to identify and evict these blocks as soon as the last reference is made. A P-Block's life cycle depends on the number of references made to the block. Therefore, maintaining the P-Block after the last reference is unnecessary.

P-Blocks differ from D-Blocks as D-Blocks are only accessed while in the MRU position. As stated earlier, D-Blocks will never be referenced after leaving the MRU position, while P-Blocks may traverse the set and be referenced again. Moreover, P-Blocks are referenced for a known number of times before they are replaced. While the number of references made to a D-Block in MRU position might vary each time it appears in cache, a P-Block is always referenced for a fixed number of times. In Figure 5.7, we illustrate a simple example to better explain P-Blocks.

```
WHILE (T < limit)
{
   T += [(A/B)+C]*D;  //Load A,B,C,D in cache

   FOR (i=0;i<7;i++)
   {
     T += E*i;        //Load E
   }
}
```

Figure 5.7: A simple example to explain P-Blocks. We assume 'A', 'B', 'C', 'D' and 'E' are all mapped to the same set in a 4-way associative data cache. So long the program runs in the WHILE loop, 'A', 'B', 'C' and 'D' are loaded calculating T's initial value. 'E' will be loaded in the inner FOR loop and will be referenced exactly 7 times before eventually being replaced by 'D'.

As shown in figure 5.8 (a), due to the order of block accesses in the given code (figure 5.7), the LRU policy would inefficiently replace and recall three blocks (‘B,’ ‘C’ and ‘D’) from the cache set. This could be prevented with the early eviction of ‘E’ as a P-Block.

P-Blocks could be viewed as a more inclusive form of bypass and dead blocks. Ideally, early eviction of P-Blocks will free cache resources for other blocks without harming cache performance. In figure 5.8 (b), we depict an alternative approach resulting in a lower miss rate for the same code in figure 5.7.

AC #	RC #	Req.	Set Blocks	Evicted Blocks
0	0	-	D C B A	
3	1	E	E D* C* B*	A
1	2	A	A E* D C	B A
0	3	B	B A E D	C B A
0	4	C	C B A E	D C B A
0	5	D	D C B A	E D C B A

(a)

AC #	RC #	Req.	Set Blocks	Evicted Blocks
0	0	-	D C B A	
3	1	E	E D* C* B*	A
1	1	A	A D C B	E* A

(b)

Figure 5.8: (a) LRU policy evicts the blocks on the order of access which results in five misses per while loop iteration (b) Early eviction of block ‘E’ as P-Block results in two misses per each while loop iteration

5.3.1 P-Block detection implementation

To identify P-Blocks we record the number of hits for each evicted block in the history table. We compare this number to the counted number of hits for the same block before being replaced. If both numbers match, the block is marked as predictable.

In order to count the number of hits for each block we add a hit counter to each block in the cache. This counter is automatically reset after each replacement. The value of the hit counter is passed to the history table for P-Block detection. The hit counter’s value is recorded in the history table for future references.

5.3.2 Dual tagged P-Blocks

D-Blocks and B-blocks can also have predictable number of accesses. If a D- or B-block is replaced after the same number of accesses before eviction it can be called a P-Block. These blocks are referred to as *Dual Tagged Blocks*. However, our speculative method only considers P-Blocks which do not overlap with D- or B-blocks. Therefore, a D- or B-block with a predictable number of accesses will not be tagged as P-Block in this study (unless stated otherwise). This is due to the fact that in our study we distinguish between P-Blocks and other blocks and report for the improvements achieved by the elimination of P-Blocks.

5.4 Avoiding LRU Non-optimal Decisions

The conventional LRU replacement policy treats all blocks in the same set equally, which results in non-optimal decisions. In order to avoid LRU's NODs we suggest keeping H-Blocks longer than LRU policy permits. In addition, we propose early eviction of P-Blocks when the block is unlikely to be reused to open space for other reusable blocks.

Using the same scheme described earlier, we identify and record H- and P-Blocks in a history table (figure 5.4). We slightly change the LRU management policy and cache structure in order to use the H- and P-Blocks' information in history table. In the following we describe our Speculative Replacement Algorithm (SRA).

5.4.1 Flagging NOD Blocks

The history table flags NOD blocks by adding two extra bits to each block stored in the cache. In addition, we add a 3-bit counter to record the number of hit accesses a block receives before eviction. This counter is called the *Hit Counter* and its value will be used later to identify any P-Blocks in the cache (following the algorithm described in section 5.3). The counter size will affect the accuracy of the P-Block detection algorithm. In case a small counter is chosen, the recorded access value for any block would only be valid up to the counter's significance (a larger counter holds a more accurate value for number of block accesses). For those blocks which are referenced frequently, the small hit counter does not present a correct hit count and will roll-over; however, for P-Blocks the counter value should still match the recorded value in history table.

In table 5.1, we show how identified blocks are flagged in the cache. If the cache block is new to the history table or not identified as any of the known NODs it will be flagged as an unknown block and the flag bits will be zero. If the block is identified as an H-Block the flag bits will all be set to one. The flag bits are set to "01" for P-Blocks and to "10" for D- or B-Blocks.

Table 5.1: Tagged blocks in the cache

Name	Flags	Keeping Priority
H-Block	11	highest
Unknown Block	00	normal
P-Block	01	lowest
D-, B-Block	10	lowest

5.4.2 Speculative Replacement Algorithm (SRA)

In our proposed replacement policy, in case of an eviction, H-Blocks have the highest priority to remain in the cache. After H-Blocks, unknown Blocks will remain longer depending on their position in the set. The unknown blocks residing in the LRU position are evicted earlier than those residing in the MRU position. D- and B-Blocks in this study are viewed as Unknown blocks. The lowest priority is given to P-Blocks. These blocks are victimized earlier than other blocks as they are found in the cache and ready to be replaced.

P-Blocks are ready for replacement as soon as their anticipated number of accesses before eviction is reached. P-Blocks are placed in the cache with their hit counter initialized to the “two’s complement” of the number of times they are anticipated to be accessed. Thus, when the P-Block’s hit counter increments to zero the P-Block is ready to be evicted.

Our suggested cache replacement policy will slow down the pace for H-Block replacement in order to promote their stay in the cache. Upon each replacement the victim is first selected among the blocks which are invalidated due to the concurrency policy between the processors’ cache. If all existing blocks in the set remain valid, the first P-Block found from LRU to MRU positions and ready to be replaced will be selected as the victim. If no P-Block exists, the untagged blocks will be victimized according to the conventional LRU policy. Finally, if all blocks are flagged as hasty, the H-Blocks will evict the cache from the LRU position.

False Sharing and Coherence Misses

Applying SRA for cache management will change the way blocks will be evicted from the cache memory; however, since the MESI protocol is still used to keep the coherency of the data shared between different cores, coherence misses will neither decrease nor increase. Upon a replacement the invalidated blocks by MESI protocol will evict the cache before any other blocks and this is consistent with LRU replacement algorithm.

However, since the eviction policy is changed in SRA algorithm, false sharing might change due to new policy in keeping the cache blocks. False sharing might increase or decrease in some benchmarks depending on there data access pattern.

5.4.3 History Table Access Time

Our speculative replacement technique (SRA) will not affect cache access time. To explore this further, we consider both hit and miss penalty. First, the cache hit latency will not change because SRA does not increase the cache size significantly (less than 4% increase). Second, the history table lookup is only made when there is a miss on the level one cache. In such case the history table will be accessed in parallel with second level of cache hierarchy while it retrieves the missed data. Since the history table is much smaller than second level of cache, the history table lookup time will be covered by the second level of cache access time. Hence, the history table is not on cache access critical path and the cache access time remains unaffected.

5.5 Methodology

For our simulations, we use the execution driven mode of the SuperESCalar (SESC) simulator [23]. We studied L1 data caches in a quad core CMP processor using the memory subsystem presented in table 5.2. Each processor has a local 4-way set associative L1 data cache but shares the L2 cache with the other three cores in the CMP processor.

We use the MESI protocol to keep the coherency between the L1 cache memories. In case the data is not found in the first level of the cache memory, a memory request will be sent to the shared L2 cache memory as well as all other cores that snoops the bus. The configuration is shown in figure 5.9. Unless stated otherwise, we present our results for the default configuration specified in table 5.2. In section 5.6.4 we report how variations in the L1 data cache configuration impact our results. To avoid cold start misses we skip the first 500M instructions and start simulations with next-500M instructions.

Table 5.2: Configuration of each core in our CMP model

Processor	**Interconnect**	**Memory System**
branch predictor: 16K entry bimodal and gshare branch penalty: 17 Fetch/issue/commit: 6/4/4 RAS: 32 entries BTB: 2K entries, 2-way	bus clk cycle: 7 processor cycles switch latency: 1 cycle link latency: 1 cycle Interconnect width: 64B	cache blk size: 32B split I-L1/ D-L1: 32/8KB, 4-way L1 latency: 3 cycl L2: 1 MB/8-way L2 latency: 11 memory latency: 469 processor cycl

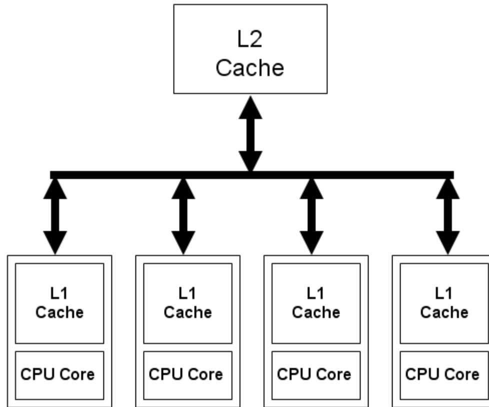

Figure 5.9: Our modeled CMP system

We use a suite of SPLASH-2 benchmarks [52] and pick the benchmarks with the highest number of memory references. Default data sets are used for programs in the benchmark suite. In table 5.3, we report the details for the benchmarks used in this study.

Table 5.3: Suite of SPLASH-2 Benchmark used

Benchmarks	Input Parameters	No. Access	Miss Rate
Barnes	16,384 particles	9.28 M	8.36 %
Cholesky	tk29.O	10.37 M	14.97 %
Fmm	16,384 Particles	10.29 M	1.78 %
Lu(contiguous)	512×512 matrix, B=16	7.68 M	3.25 %
Ocean (not cont)	258 points	28.4 M	2.77 %
Radiosity	-batch	9.89 M	0.22 %
Radix	8M	2.99 M	1.99 %
Water(nsquared)	4k molecules	6.26 M	6.60 %
Water(spatial)	4k molecules	6.78 M	1.00 %

5.6 Results

In order to distinguish our results from previous work, we only report the cache performance improvement achieved by eliminating P- and H- blocks. Section 5.6.1 compares the NOD distribution between LRU and the application of our proposed replacement algorithm. In section 5.6.2 we vary the size of our history table and report its

accuracy and coverage in identifying H- and P-Blocks. Section 5.6.3 presents the overall miss rate reduction as a result of lower replacement frequency in the cache. Finally, in section 5.6.4 we vary the size of L1 data cache and study the sensitivity of the results.

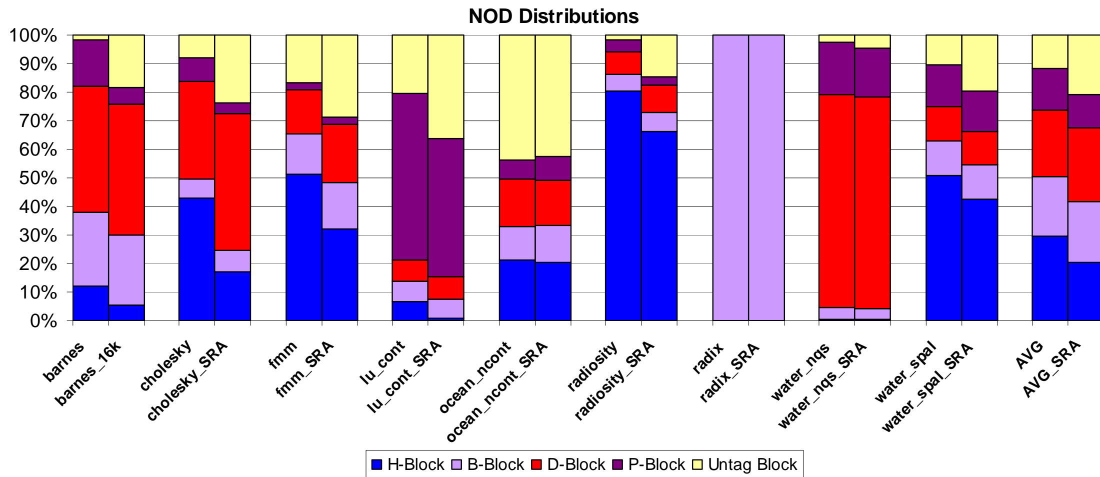

Figure 5.10: NOD distribution. For each benchmark the left bar represents the LRU replacement policy and the right bar is when the SRA is used with a 16k entries history table.

5.6.1 NOD Distribution

In table 5.3, we report the cache miss rate for the given CMP system under the LRU replacement policy. In figure 5.10, we report NOD distribution both when LRU (left bar) and SRA (right bar) are used. We used the detection algorithms explained earlier (sections 5.2 and 5.3) to identify H- and P-Blocks. In order to make comparisons with previous work possible we also measure and report D- and B-block frequencies.

As illustrated in figure 5.10, on average 88% of the replaced blocks belong to one of the NOD classes studied here. Except for *ocean (non-contiguous)*, each application is dominated by one of the NODs. H-Blocks account for a majority of NODs in *cholesky*, *radiosity*, *water (Spatial)* and *fmm*. P-Blocks are the most common NOD in *lu (contiguous)*, whereas *radix* benchmark is dominated by B-blocks. In the case of *water (nsquared)* and *barnes* benchmarks D-Blocks outnumbered other NODs.

Note that almost all NODs (99.9 %) in *radix* are B-blocks. This can be explained by the unique workload characteristics associated with *radix*. *Radix* is an iterative integer sort algorithm. In each iteration, a processor passes the parameters to other processors and uses the others' parameters to permute local variables for a new iteration. In this all-to-all communication, data is passed over through a destination-determined writes rather than reads [52]. Our studies show that most *radix* cache accesses made after warm up are cache write-back misses. This is due to write requests to locations in memory which are only accessed once and will not be referenced again in the near future. This will evict the data from the cache before being referenced again. Therefore, almost all blocks are B-Blocks. Since our speculative method will only reduce H- and P-Blocks and will not affect the Radix benchmark, we do not report for this benchmark in our next result evaluation.

Also note that on average 12% of evicted blocks in LRU replacements are untagged, which means they are either not an NOD or of an unknown NOD type. This provides opportunities for further studies.

P-Blocks account for 15% of the NODs. In the case of *lu (Contiguous)* more than 58% of the NODs are of P-Blocks. H-Blocks account for about 30% of cache replacements, which is twice as much as L-Blocks.

5.6.2 Accuracy and Coverage

Figure 5.11 depicts the accuracy and coverage of predicted H- and P-Blocks when the history table size is limited. We used an 8KB data cache in each core to study our technique under a high number of cache misses. In figure 5.11 (a) we report our

prediction accuracy for H-Blocks. We use the same algorithm explained in section 5.2 and 5.3, using limited size first-in-first-out (FIFO) history table structures. The history table size varies from 512 to 64k entries.

As reported in the figure, the average accuracy is between 83% and 93%. In case of small history tables of size 512 entries, the recorded accuracy is higher. This is because only a few H-Blocks are identified and these H-Blocks most likely have a predictable behavior. The worst prediction accuracy is reported for *ocean (non-Contiguous)*. This benchmark has the largest number of cache accesses among other SPLASH2 benchmarks and 45% of its replaced blocks are of an unknown type (figure 5.10).

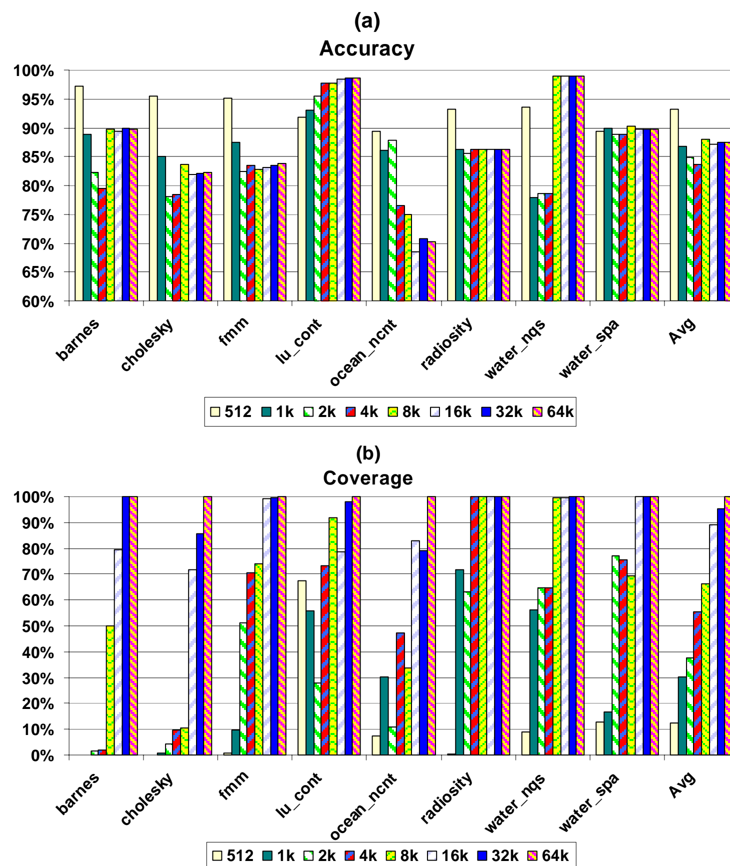

Figure 5.11: H-Blocks prediction accuracy and coverage achieved by different history table configurations and for the Splash 2 benchmarks studied here. For each benchmark bars from left to right report for tables of size 512 to 64k entries.

Note that mispredicting an H-Block does not harm the performance as it only results in the early eviction of an unidentified H-Block, which follows the traditional LRU policy.

In 5.11 (b), we report the percentage of H-Blocks identified correctly. On average different configurations of history tables identify from 12% up to 100% of H-Blocks. We conclude from figure 5.11 (b) that a history table of size 16K is large enough to predict most of the H-Blocks. Also, we find that increasing the table size above 32K does not impact coverage.

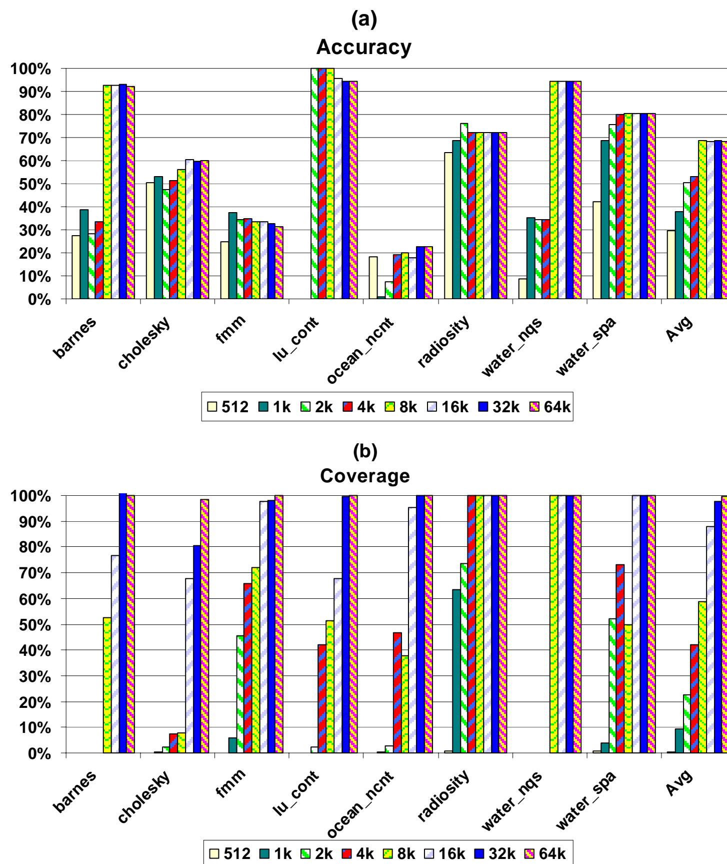

Figure 5.12: P-Block prediction accuracy and coverage achieved by different history table configurations, and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries.

In figure 5.12 we report accuracy and coverage for P-Blocks. As shown in 5.12 (a), our algorithm yields a lower accuracy for the prediction of P-Blocks. This is due to

the vulnerability of P-Blocks to be mistaken for other blocks (section 5.3.2). D- or B-blocks which behave like P-Blocks are not recorded in the history table. Thus, predictable blocks which occasionally behave like D- or B-blocks (*dual tagged blocks*) are not identified by our speculative approach (SRA). As shown in 5.12 (a), this affects the accuracy of P-Block prediction for those benchmarks with too many dual tagged blocks.

On average an 8K history table identifies 60% of P-Blocks. However, in order to predict more than 90% of P-Blocks a 16k history table is needed.

5.6.3 Miss Rate Reduction

As reported in section 5.5, our goal is to change the cache replacement policy to prevent hasty replacement of H-Blocks and facilitate early eviction of P-Blocks. As shown in figure 5.10, this reduces the number of NODs, which in turn reduces the cache miss rate.

We vary the history table configurations in the CMP processor studied here and report for the cache replacement reduction achieved by H- and P-Block elimination. Figure 5.13 shows the cache miss rate reduction due to the high data availability achieved by our approach.

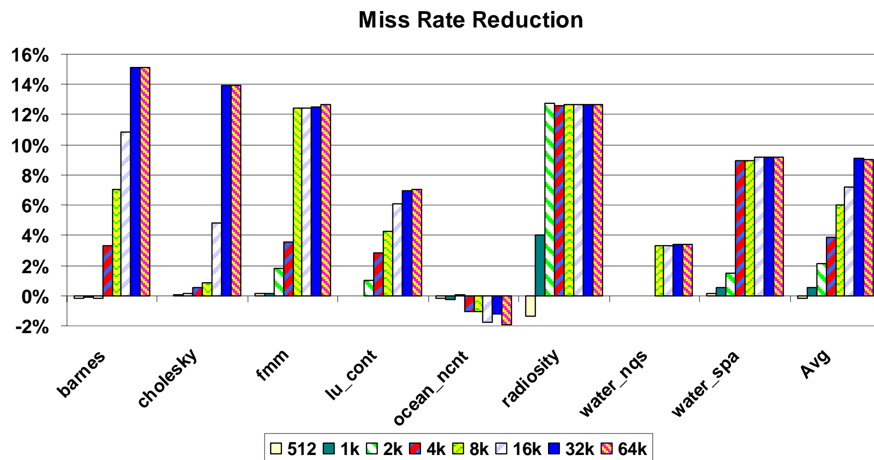

Figure 5.13: Decreasing cache block replacement by using the SRA cache management technique. For each benchmark the bars from left to right report for tables of size 512 to 64k entries.

Our cache management policy reduced cache evictions and achieved a 9% improvement in cache miss rate exploiting a 64K entries FIFO-style history table. The only benchmark which did not gain any improvement was *ocean (non-contiguous)*. This is due to the poor prediction accuracy for this benchmark when using our algorithm. In case of *barnes* high accuracy and coverage for both H- and P-Blocks resulted in a maximum 15% reduction in cache miss rate.

5.6.4 Sensitivity Analysis

In order to find the impact of variations in the memory size on SRA technique, we increase the cache size from 8KB to 16KB and 32KB and illustrate the results in figure 5.14. We report the predicted H-Block coverage and accuracy when larger caches are used. As shown in figure 5.14, the data cache memory size has a minor influence on the history table prediction results.

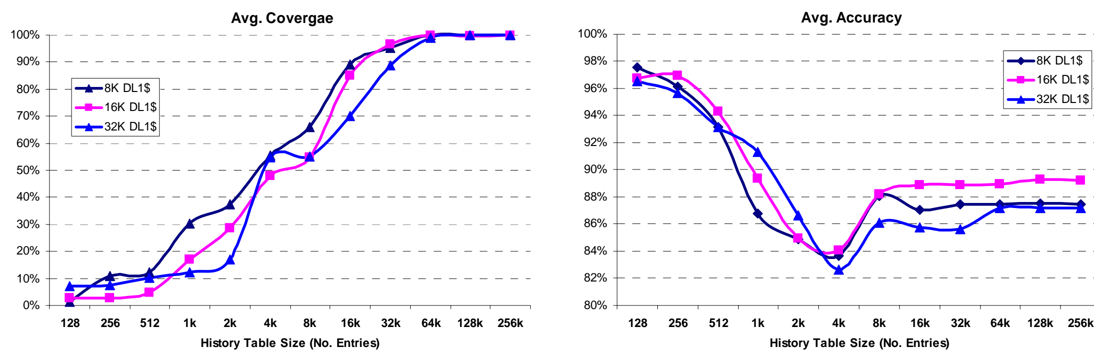

Figure 5.14: Average H-Blocks prediction coverage and accuracy for splash-2 benchmarks achieved by different history table configurations. Each line represents a different size of data cache memory per core.

The direct advantage of using a larger cache organization is the reduction in cache conflict misses that by itself improves the cache miss rate. With fewer conflicts in the

cache, NOD frequency is reduced. Therefore our replacement policy will have less impact on cache miss rate in larger caches. This is shown in figure 5.15.

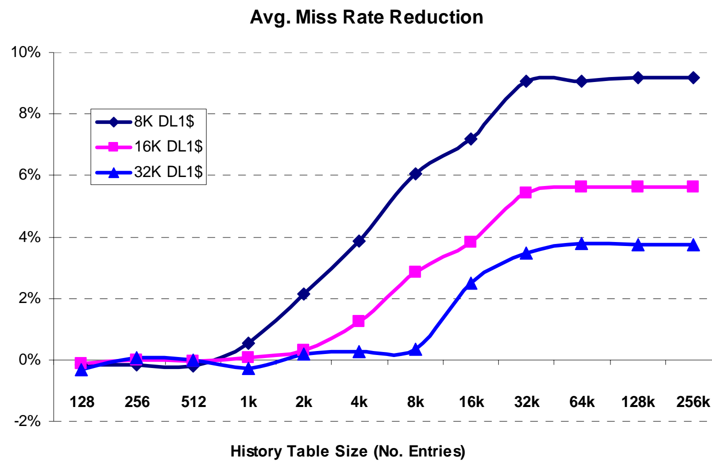

Figure 5.15: Average H-Blocks miss rate reduction for splash-2 benchmarks achieved by different history table configurations. Each line represents different size of data cache memory per core.

5.7 History Table Implementation

In this section we investigate the hardware overhead of our speculative replacement algorithm (SRA). Our general cache replacement mechanism requires hardware overhead for two reasons: first, every cache block (as explain in 5.4.1) requires two flag bits and a 3-bit hit counter. Moreover, each set of blocks needs two small RC and AC counters. Considering a 4-way set associative cache with a 3-bit hit counter for each cache block, the first hardware overhead is 4 bytes per cache entry.

The second source of overhead is in the history table. History table will maintain some information from each discarded block from the cache. Each entry in the history table contains a 13-bit compressed version [5], [2] of the evicted block's tag key, and also keeps the replacement counter value (RC) recorded at the block eviction. The block hit counter is also stored in order to identify P-Blocks, and a 2-bit field is needed to store the detected blocks (H-, P-Block, or unknown). Figure 5.16 shows the size of each portion in history table entries. As shown, each entry in the history table takes three bytes to store the required data.

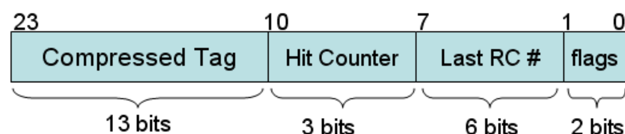

Figure 5.16: The information stored in history table entries.

In previous sections, we studied the impact of alternative history table sizes on miss rate reduction. In the following sections, we suggest alternative organizations of history table to decrease the history table size while keeping the required information.

5.7.1 LRU Style History Table (LHT)

The main idea is to implement the history table so that it only maintains valuable information (*i.e.* H- and P-Blocks' information) using a limited number of table entries. In order to achieve this we modified the FIFO-style history table organization and proposed a fully associative history table with the LRU eviction policy. The LRU eviction policy assures that the entries which are not accessed as often would be evicted to free space for more frequently used cache blocks' information.

In such history table structure, the required information gathered from evicted blocks will be placed from the head of the history table (entry location 1). When the table is full the LRU eviction policy will choose the least recently used table entry to be freed. In addition, H- or P-Blocks in the table that are referenced again are moved to the head of the history table. This implementation assures that the most recently used blocks are kept longer in the history table. We refer to this history table implementation as the LRU style history table.

5.7.2 Modified LRU-Style History Table (MLHT)

In this section we describe a modified version of LRU-style history table which reduces the hardware overhead even further. The modified structure will increase design complexity to maintain the useful information longer in the table.

In this modified version of the LRU-style history table (MLHT) we divide the history table into two substructures (fully associative Table A and FIFO-Table B). As shown in figure 5.17, the first entry in table B is the insertion point for newly arriving blocks discarded from the cache. The cache blocks' information is inserted from the

insertion point (the first entry in table B) and discarded from the tail of table B (the last entry in table B). If an H- or P-Block is referenced in table B, the block information is promoted and placed in table A. Table A discards the blocks using the LRU eviction policy; however, the discarded blocks' information from table A will not immediately leave the history table but is first demoted to table B. Finally, if such blocks are not referenced again they eventually leave the history table and are discarded from the tail of table B. Searching the history table starts from table A and continues to table B.

Table B gives the opportunity for the evicted cache blocks to be placed in table A. An ideal MLHT will have enough space in table B to identify potential H- and P-Blocks. We dedicate two thirds of the total history table size to table B so that H- and P-Blocks could stay long enough before they are referenced again. Table A should contain enough entries to maintain all active H- and P-Blocks in the history table.

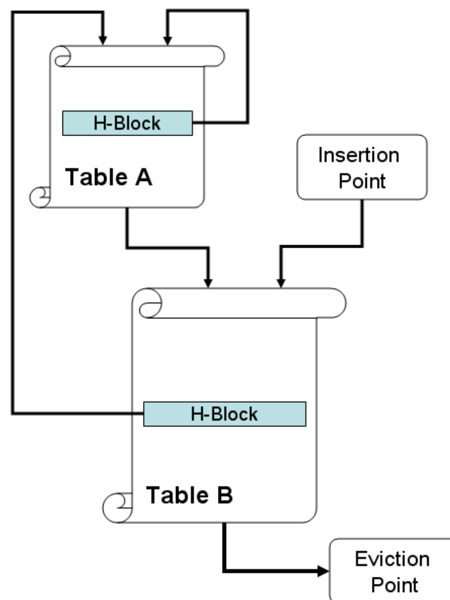

Figure 5.17: Modified version of LRU style history table (MLHT) diagram

Despite the MLHT implementation complexity, it is highly effective in reducing the size of the history table (the history table is reduced by a factor of 16). As shown in figure 5.18, on average MLHT of size 512 entries will have higher coverage for predicted H-Blocks compared to an 8k entries FIFO-style history table shown in figure 5.11. The same statement is true for P-Blocks in figure 5.19.

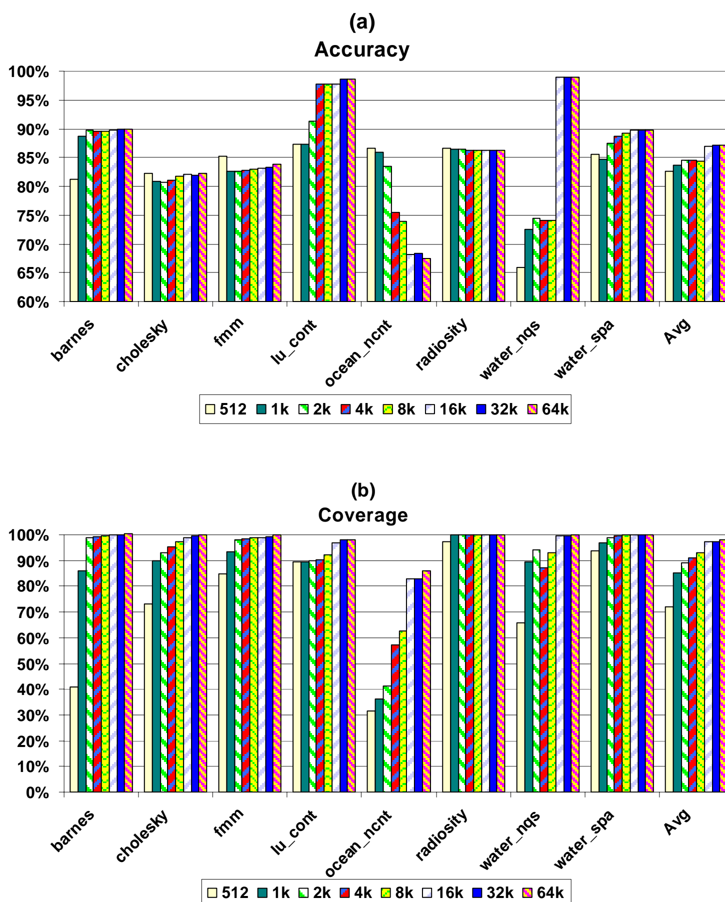

Figure 5.18: H-Block prediction accuracy and coverage achieved by different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries.

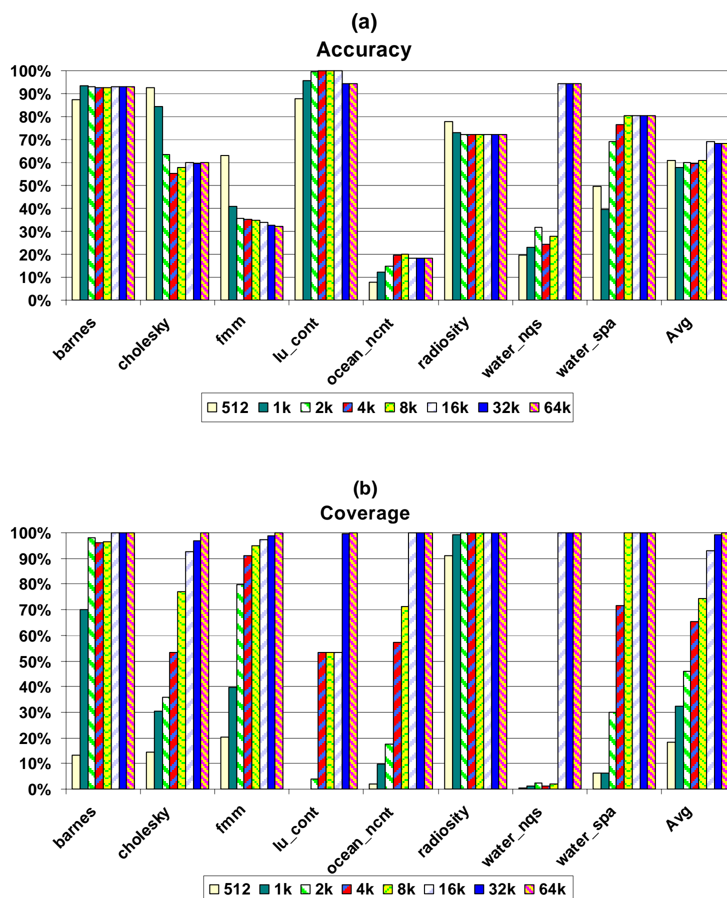

Figure 5.19: P-Block prediction accuracy and coverage achieved by different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries.

5.8 Cache miss rate reduction with MLHT

MLHT is capable of identifying H- and P-Blocks with a smaller hardware overhead. In this section we include a comprehensive result for cache miss rate reduction when both H-Blocks and P-Blocks are eliminated.

As shown in figure 5.20, depending on the MLHT size the data cache miss rate is reduced on average from 4% to 9%. The highest cache miss reduction is for *barnes* with a 15% miss rate improvement. Due to poor prediction accuracy and low coverage, our

method has a negative affect on *ocean (non-contiguous)* and will increase the cache miss rate for this benchmark.

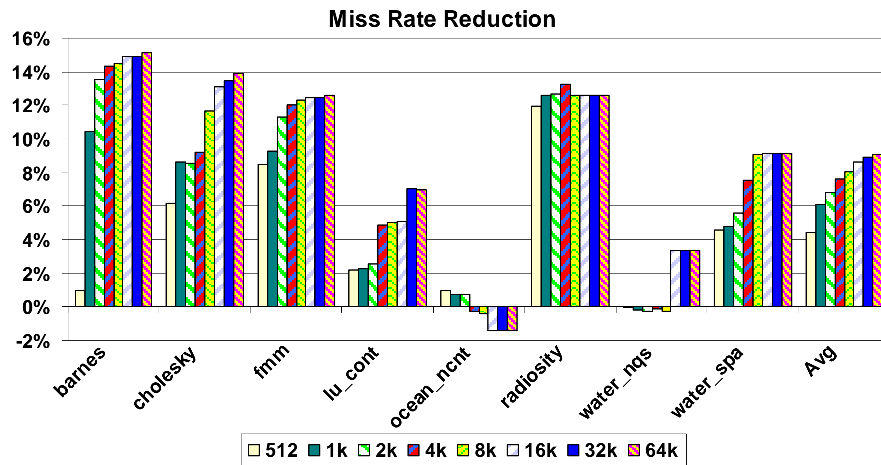

Figure 5.20: Miss rate reduction for different configurations of MLHT and for the Splash 2 benchmarks studied here. For each benchmark the bars from left to right report for tables of size 512 to 64k entries.

Chapter 6

Conclusions

In our quest for the most optimal processor architecture that could consume less energy and accelerate overall performance, we presented two optimization techniques for the cache memory system and the branch predictor: two components that exist in almost all modern processors. Using a simulation model, we quantified the relative improvements in energy consumption and performance speedup.

6.1 Summary of contributions

The rest of this chapter summarizes the key contributions of this dissertation:

Chapter 3: Design and Analysis of an Energy-aware Branch Target Buffer

Chapter 3 introduced an alternative energy-aware BTB structure. We exploited the fact that many BTB accesses are unnecessary as they consume energy without contributing to performance. Unnecessary BTB accesses occur during cycles where no branch instruction is among those fetched. We eliminate such accesses by using a filter referred to as the BLC-filter. We studied how variations in the BLC-filter configuration impacts energy use and performance in both high-performance and embedded processors. By using a small low-overhead structure we reduced BTB energy consumption considerably

with negligible 0.2% performance cost. We showed that the BLC-filter does not impact predictor delay as it stops unnecessary accesses occurring in future cycles.

Chapter 4: Cache Complexity Analysis for Modern Processors

In Chapter 4 we provided a complexity analysis for instruction and data caches in both the embedded and high performance design space. We showed that improving cache hit rate is only justifiable if the associated energy and latency overhead stays below a break-even point. We calculated and estimated this break-even point for modern processors with different execution bandwidths, considering SPEC'2k and MiBench applications. We found that the instruction cache has a lower latency break-even point compared to the data cache. We also showed that spending energy on achieving an ideal data cache is likely to result in better energy efficiency compared to that spent to achieve an ideal instruction cache. Our investigation of alternative cache designs for modeled processors revealed that not all alternatives would improve energy-efficiency.

Chapter 5: Reducing Non-optimal LRU Decisions in Chip Multiprocessors

In Chapter 5, we analyzed the non-optimal decisions made by LRU in chip multiprocessors. We introduced Hasty Blocks and Predictable Blocks as more inclusive extensions of Live and Dead Blocks, and showed that about 45% of the cache evictions are either Hasty or Predictable-blocks. H-blocks are more frequent than P-blocks and account for 30% of cache replacements. Our speculative technique offers a significant reduction in undesirable LRU policy decisions in a quad-core CMP processor without impacting cache access time. With about 3 KB memory overhead our Modified LRU-style History Table (MLHT) implementation of Speculative Replacement Algorithm

(SRA) could reduce cache miss rate by 6% on average and close the gap between the LRU and Belady's theoretical optimal replacement policy [45].

6.2 Future work

Simulation studies show that the techniques presented in this dissertation offer improvements in the energy dissipation and memory performance of modern processors. In the following, we suggest a number of directions to improve the results for future research works.

6.2.1 Further Energy Reduction through Speculative Allocation

Our BTB allocation technique reduces energy consumption during the fetch cycles where BTB access does not contribute to the overall fetching mechanism. This happens when there is no branch or jump instruction in the next sequence of fetched instructions. An extension to this work could consider fetching cycles with not-taken branch instructions.

In general, characterization of fetch cycles provides fine grain information (up to the pipeline width granularity) about fetched instructions. This information could be used for per-cycle energy or performance optimization.

6.2.2 Comprehensive Energy Budget Evaluation

Our energy budget formulation did not account for memory size impact on cache access time. A more in-depth calculation would consider memory configuration and its impacts on overall processor speedup.

Energy budget is a general concept which can be applied to other processor components. Considering energy budget along with Amdahl's law will provide a good insight for targeted design modification.

6.2.3 Implementation of History Based Cache Replacement Policy

We presented a novel cache replacement policy improving cache miss rate with no negative effects on cache parameters such as hit latency, miss penalty or throughput. However, in order to make this technique practical the most important next step is to keep the hardware overhead under a certain limitation. The suggested LRU-style history table introduces a high degree of complexity which reveals new challenges in low level design implementation. An alternative option is using a *Pseudo-LRU* algorithm to find and victimize the least recently used entries in the history table. This algorithm is easier to implement for large number of entries and requires less complexity compared to LRU policy.

Bibliography

[1] "The International Technology Roadmap for Semiconductors" [Online] http://www.itrs.net/.

[2] A. Lai, C. Fide and B. Falsafi. "Dead-Block Prediction and Dead-Block Correlating Prefetchers", *28th International Symposium on Computer Architecture*, June 2001.

[3] Amirali Baniasadi, Andreas Moshovos, "Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors", *In Proceedings of ICCD 2002*, pp. 458-461.

[4] Amirali Baniasadi, Andreas Moshovos, "SEPAS: A Highly Accurate and Energy Efficient Branch Predictor", *Proceedings of International Symposium on Low Power Electronics and Design*, 2004, pp. 38-43.

[5] An-Chow Lai and Babak Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction". *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[6] André Seznec, "Analysis of the O-GEometric History Length Branch Predictor", *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 394-405.

[7] Antonio González, Carlos Aliagas and Mateo Valero. "A Data Cache with Multiple Caching Strategies tuned to Different Types of Locality." *In Proceedings of the 9th ACM international conference on Supercomputing*, pp. 338-347, July 1995.

[8] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A frame work for architectural power analysis and optimizations", in *Proceedings of the 27th international Symposium Computer Architecture*, pp. 83–94, 2000.

[9] D. C. Burger and T. M. Austin. "The SimpleScalar tool set,version 2.0.", *Computer Architecture News*, vol. 25 (no. 3), pp. 13–25, June 1997.

[10] D. Chaver, L. Pinuel, M. Prieto, F. Tirado and M. C. Huang. "Branch Prediction on Demand: an Energy Efficient Solution", In *Proceedings of International Symposium on Low Power Electronics and Design*, 2003.

[11] D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", *Journal of Instruction-Level Parallelism*, Vol. 2, 2000.

[12] Daniel A. Jimenez and Calvin Lin. "Neural methods for dynamic branch prediction." *ACM Transactions on Computer Systems*, pp. 369–397, 2002.

[13] Daniel A. Jiménez, "Piecewise Linear Branch Prediction", in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, June 2005

[14] Daniel A. Jimenez, Stephen W. Keckler, and Calvin Lin. "The impact of delay on the design of branch predictors." *In Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, pp. 66–77, 2000.

[15] Daniel A. Jimnez. "Fast path-based neural branch prediction." *In The 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, page 243, 2003.

[16] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Mircea R. Stan, "Power-Aware Branch Prediction: Characterization and Design", *IEEE Transaction on Computers*, Vol. 53,No. 2, pp. 168-186, February 2004.

[17] E. Rotenberg, S. Bennett, and J. Smith. "A trace cache microarchitecture and evaluation". *IEEE Transaction on Computers*, 48(2):111--120, Feb 1999.

[18] Gary Tyson, Matthew Farrens, John Matthews, Andrew R Pleszkun, "A Modified Approach to Data Cache Management", in *Proceedings of 28th IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, pp. 93-103, November 1995.

[19] Gilberto Contreras, Margaret Martonosi, Jinzhan Peng, Roy Ju, Guei-Yuan Lueh, "XTREM: a power simulator for the Intel XScale eg; core." in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004),* pp. 115-125, 2004.

[20] J. Hennessy, D. Patterson, "Computer Architecture a Quantitative Approach", 3rd Edition, Morgan Kaufmann publishing, by Elsevier Science (USA), 2003.

[21] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design", *IEEE Journal of Computer,* Vol. 21(No. 7), 1984.

[22] J. Montanaro, et al. "A 160 MHz, 32b 0.5 W CMOS RISC Microprocessor", In *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC 1996)* Digest of Papers, 1996.

[23] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos. "SESC Simulator", [Online] Jan 2005, http://sesc.sourceforge.net.

[24] J.E. Smith, "A study of branch prediction strategies". in *Proceedings of the 8th International Symposium on Computer Architecture (ISCA-8)*, 1981.

[25] J.F. Edmondson, et al, "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor", *In Digital Technical Journal*, vol. 7, no. 1, 1995.

[26] Jaeheon Jeong, Michel Dubois, "Cost-Sensitive Cache Replacement Algorithms", in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, pp. 327-, 2003.

[27] John Henning, "SPEC CPU 2000: Measruring CPU performance in the new millennium", *IEEE Computer Society Press*, Pages 28-35, Vol 33, Issue 7, 2000.

[28] Juan, T., Sanjeevan, S., Navaro, J. J. "Dynamic History-Length Fitting: A third level of adaptivity for branch prediction", in *Proceedings of the 25th annual International Symposium Computer Architecture*, June 1998.

[29] K. Farkas and N. Jouppi. "Complexity/performance tradeoffs with non-blocking loads". in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 211-222, April 1994.

[30] K. Skadron, T. F. Abdelzaher, M. R. Stan: Control-Theoretic "Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management", in *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pp. 17-28, February 2002.

[31] Kaveh Jokar Deris, Amirali Baniasadi, "Investigating Cache Energy and Latency Break-even Points", *ACM SIGARCH Newsletter of Computer Architecture News* Vol. 35, No. 4, September 2007.

[32] Kaveh Jokar Deris, A. Baniasadi, "Branchless Cycle Prediction for Embedded Processors", *in Proceedings of The 21st ACM Symposium on Applied Computing (SAC2006)*, pp. 23-27, April 2006, Dijon, France.

[33] Kaveh Jokar Deris, A. Baniasadi, "SABA: a Zero Timing Overhead Power-Aware BTB for High-Performance Processors", *in Proceedings of workshop on Unique Chips and Systems in conjunction with ISPASS-2006*, 19-21 March 2006, Austin, Texas, USA.

[34] Kaveh Jokar Deris, Amirali Baniasadi, "Investigating Cache Energy Efficiency in Multimedia Processors", *in Proceedings of 20th Canadian Conference on Electrical and Computer Engineering (CCECE 2007)*, Vancouver Canada, April 2007.

[35] Kaveh Jokar Deris, Amirali Baniasadi, "Investigating Cache Energy and Latency Break-even Points in High Performance Processors", *in Proceedings of workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA) in conjunction with IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT-2006)*, Seattle USA, September 2006.

[36] Liangzhong Yin, Guohong Cao, Chita R. Das, Ajeesh Ashraf, "Power-Aware Prefetch in Mobile Environments", in *Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pp. 571-578, 2002.

[37] M. Co, D. A.B. Weikle, and K. Skadron. "A Break-Even Formulation for Evaluating Branch Predictor Energy Efficiency." *In 2005 Workshop on Complexity-Effective Design (WCED) held in conjunction with the 32nd Annual ACM/IEEE Int'l Symposium on Computer Architecture (ISCA)*, 2005.

[38] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., J. Emer, "Adaptive Insertion Policies for High Performance Caching", *International Symposium on Computer Arachitecture*, June 2007.

[39] M. Kampe, P. Stenstrom, and M. Dubois, "Self-correcting LRU replacement policies", *In Proceedings of Computing Frontiers*, 2004.

[40] M. Kharbutli, Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms", *IEEE Transactions on Computers*, Volume 57, Issue 4, April 2008 Page(s):433-447.

[41] Matteo Monchiero, Gianluca Palermo, Mariagiovanna Sami, Cristina Silvano, Vittorio Zaccaria and Roberto Zafalon. "Low-Power Branch Prediction Techniques for VLIW Architectures: A Compiler-Hints Based Approach Integration", *The VLSI Journal*, 38(3):515-524, January 2005.

[42] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[43] Moinuddin K. Qureshi, David Thompson, Yale N. Patt, "The V-Way Cache: Demand Based Associativity via Global Replacement", *In proceeding of International Symposium of Computer Architecture 2005 (ISCA-32)*, pp. 544-555

[44] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", in *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364-373, June 1990.

[45] Oliver Temam, "An Algorithm for Optimally Exploiting Spatial and Temporal Locality in Upper Memory Levels", *IEEE Transactions on Computers*, Volume 48, Issue 2 (February 1999).

[46] P. Juang, K. Skadron, M. Martonosi, Z. Hu, D. W. Clark, P. Diodato, S. Kaxiras, "Implementing branch-predictor decay using quasi-static memory cells", *ACM Transactions on Architecture and Code Optimization*, pp. 180-219, 2004.

[47] P.Y. Change, E.Hao and Y.N.Patt, "Implementation of hybrid branch predictors", in *Proceedings of the 25th International Symposium on Computer Architecture*, 1995.

[48] Palacharla S., and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", in *Proceedings of 21st International Symposium Computer Architecture*, pp. 24-33, 1994.

[49] S. McFarling, "Combining Branch Predictors", *Technical Note TN-36*, In *DEC Western Research Laboratory*, June 1993.

[50] S. pan, k.So and J. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation". in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[51] S. Wilton and N. Jouppi. "An Enhanced Access and Cycle Time Model for On-chip Caches." In *WRL Research Report 93/5, DEC Western Research Laboratory,* 1994.

[52] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. "The SPLASH-2 programs: Characterization and methodological considerations", *In Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[53] S.McFarling and J.Hennessy, "Reducing the cost of branches". *In Proceedings of the 13th International Symposium on Computer Architecture*, 1986.

[54] Scott Gifford, Chien-Wen Huang, Zimin Yang, and Cong Yu, “A Comprehensive front-end architecture for the verisimple alpha pipeline”, [Online] http://citeseer.ist.psu.edu/gifford03comprehensive.html .

[55] Song Jiang, Xiaodong Zhang, “Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance”, *IEEE Transactions Computers, Vol.* 54 (No. 8), pp. 939-952, 2005.

[56] Soong Hyun Shin, Cheol Hong Kim, Chu Shik Jhon, “An Effective Instruction Cache Prefetch Policy by Exploiting Cache History Information”, *In Proceedings of Embedded and Ubiquitous Computing 2005*, pp. 57-66.

[57] Sudarshan K. Srinivasan, Miroslav N. Velev, “Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Impress Data-Memory Exceptions”, *In Proceedings of First ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2003)*, pp. 65-74.

[58] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, Chaitali Chakrabarti, and Wayne Wolf, “Mobile Supercomputers”, *IEEE Computer*, Vol. 37, No. 5, pp. 82-84, May 2004.

[59] T. Yeh and Y.N.Patt, “Alternative implementation of two-level adaptive branch prediction”. *In Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[60] Teresa L. Johnson and Wen-mei W. Hwu., “Run-time Adaptive Cache Hierarchy Management via Reference Analysis”, *In Proceedings of the 24th International Symposium on Computer Architecture*, pp. 315-326, June 1997.

[61] V. Zyuban and P. Strenski., “Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels”, *In Proceedings of the 2002 International Symposium on Low Power Eelctronics and Design*, pp. 166-171, 2002.

[62] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, “A Prefetch Taxonomy”, *In IEEE Transactions Computers, Vol.* 53(No. 2), pp. 126-140 (2004).

[63] Z. Hu, P. Juang, K. Skadron, D. Clark and M. Martonosi. “Applying Decay Strategies to Branch Predictors for Leakage Energy Saving”, *In Proceedings of IEEE International Conferences on Computers and Processors*, 2002.

[64] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, Charles C. Weems, “Using the Compiler to Improve Cache Replacement Decisions”, *In Proceedings of IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, pp. 199-, *2002*.

Appendix A

Simulation Tools

A.1 SimpleScalar Simulator

The SimpleScalar tool set is an open source software used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. The program is developed by Todd A. Austin et al.,[9] and is written in C programming language.

The toolset includes many simulators ranging from a fast functional simulator to a detailed, cycle accurate dynamic scheduled superscalar processor. The simulator supports non-Blocking cache, out-of-order execution and branch prediction. SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. An overview of the simulator is shown in figure A.1.

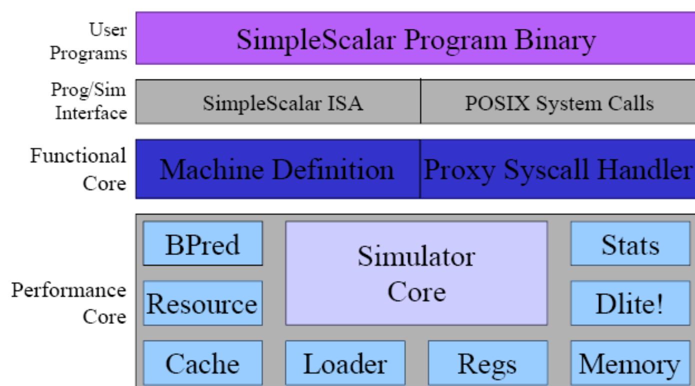

Figure A.1: Simulator Structure

After compiling the SimpleScalar on the host machine, the code accepts a configuration file and the executable code which is compiled in any of the instruction set architectures listed above. The configuration file sets the parameters for different units of the processor, which are simulated by SimpleScalar.

Simulation Parameters	Computed Values		Parameters Description
sim_num_insn	500000002	#	total number of instructions committed
sim_num_refs	271445046	#	total number of loads and stores committed
sim_num_loads	145508257	#	total number of loads committed
sim_num_stores	125936789.00	#	total number of stores committed
sim_num_branches	75536678	#	total number of branches committed
sim_elapsed_time	1577	#	total simulation time in seconds
sim_inst_rate	317057.7058	#	simulation speed (in insts/sec)
sim_total_insn	570735874	#	total number of instructions executed
sim_total_refs	300144785	#	total number of loads and stores executed
sim_total_loads	161652906	#	total number of loads executed
sim_total_stores	138491879.00	#	total number of stores executed
sim_total_branches	87089550	#	total number of branches executed
sim_cycle	253771651	#	total simulation time in cycles
sim_IPC	1.9703	#	instructions per cycle
sim_CPI	0.5075	#	cycles per instruction
sim_exec_BW	2.2490	#	total instructions (mis-spec + committed) per cycle
sim_IPB	6.6193	#	instruction per branch

Figure A.2: SimpleScalar statistical output file

The output would be both the program execution outcome and a text file representing statistical data of the simulated processor. Some of the common statistics would be how many cycles the program execution last, or how many instructions were executed. Details about prediction accuracy rate or cache miss rate can also be monitored. Figure A.2 shows some of these parameters with a typical numbers computed for an imaginary processor. One of the capabilities of SimpleScalar is that the user can measure alternative simulation parameter based on his need.

A.2 Wattch Simulator

Wattch is a framework for analyzing and optimizing modern microprocessors' energy dissipation. Wattch is much faster than existing lay-out level power tools and estimates energy dissipation within 10% of the industrial tools' accuracy.

Wattch allows the designer to evaluate energy dissipation in the early design stages. The software is a complement to the existing low-level design tools. David Brooks [8] developed the program in C. Wattch uses the power model introduced by Steve Wilton and Norm Jouppi in the Cacti tool set. We briefly discuss the Cacti toolset in the following section.

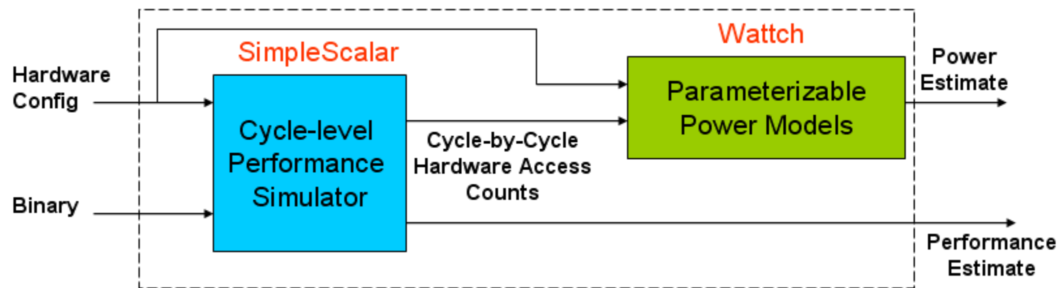

Figure A.3: The overall structure of Wattch

Wattch is integrated into other architectural performance simulators such as SimpleScalar. Common structures in superscalar processors have been parameterized by the power model used in wattch. Wattch starts with computing the power model parameters for each unit at program start up. Then, during cycle-level simulation the per-cycle usage of units is counted. The energy estimation is generated based on the counted accesses and the computed parameters. Figure A.3 shows the overall structure of the power model.

Architectural units in the processor are modeled differently based on their structure. The main four categories used for power modeling are: array structure, fully associative content addressable memory, combinational logics and wirings, and clocking. Table A.1 summarizes major hardware structures and the type of model used for each of them.

Table A.1: Common processor hardware units and the type of model used by Wattch

Type of Structure	Associated Hardware Units
Array Structure	Data and instruction cache, cache tag arrays, all register files, register alias table, branch predictors and large portions of the instruction window and load/store queue.
Fully Associative Content-Addressable Memory	Instruction window/re-order buffer wakeup logic, and load/store order checks.
Combination Logics and Wires	Functional unit, instruction window selection logic, dependency logic, and result buses.
Clocking	Clock buffers, clock wires, and capacitive loads

The power model parameters and maximum energy consumption, estimated by Wattch, was validated by comparing the numbers with industrial hardware designs and capacitance values. Table A.2 and A.3 compare the breakdown of Wattch's estimated energy consumption with the reported data for different hardware structures. The estimated values are within 10-13% of the actual data on average.

Table A.2: Comparison between modeled and reported power breakdown for the Pentium Pro®

Hardware Structure	Intel Data	Model
Instruction Fetch	22.2 %	21.0 %
Register Alias Table	6.3 %	4.9 %
Reservation Stations	7.9 %	8.9 %
Reorder Buffer	11.1 %	11.9 %
Integer Exec. Unit	14.3 %	14.6 %
Data Cache Unit	11.1 %	11.5 %
Memory Order Buffer	6.3 %	4.7 %
Floating Point Exec. Unit	7.9 %	8.0 %
Global Clock	7.9 %	10.5 %
Branch Target Buffer	4.7 %	3.8 %

Table A.3: Comparison between modeled and reported power breakdown for the Alpha 21264

Hardware Structure	Alpha 21264	Model
Caches	16.1 %	15.3 %
Out-of-Order Issue Logic	19.3 %	20.6 %
Memory Management Unit	8.6 %	11.7 %
Floating Point Exec. Unit	10.8 %	11.0 %
Integer Exec. Unit	10.8 %	11.0 %
Total Clock Power	34.4 %	30.4 %

A.3 SESC: SuperESCalar Simulator

SESC is another microprocessor architectural simulator developed by Jose Renau et al., [23] to model different processor architectures such as single processors, chip multi-processors, and processors-in-memory. SESC is an event-driven simulator and emulates MIPS instruction set architecture (ISA). SESC is written in C++ to offer simulation speed and object-oriented programming features. SESC is a very fast simulator and is capable of executing 1.5 million instructions per second.

SESC consists of two parts the instruction emulator (MINT) which executes the instructions in-order and a simulator which applies the out-of-order processor timings. Once an instruction is executed by MINT a dynamic instruction is created and would call many different events in the pipeline until it is destroyed by the reorder buffer at retirement.

SESC also reports for energy consumption, and models the processor for thermal characteristics. Finally it can simulate both in execution driven and trace driven modes.

A.4 Cacti Tool Set

Cacti [51] is a cache analyzing tool estimating access time, cycle time, area, aspect ratio and energy consumption of a modeled cache structure. Considering all these, a designer can be confident that the tradeoffs between time, power and area computations are based on the same assumption.

Cacti supports fully-associative caches, multi-ported caches, fully-independent banking, feature size scaling, power and area modeling. It takes in cache capacity,

associativity, cache-line size, the number of read/write ports, clock frequency, and the feature size. It uses an analytical model to compute the access time, consumed energy and efficiency of occupied layout and aspect ratio for different configurations. It also returns the configuration which has the best trade off for access time and energy consumption, determined by its optimization function.

Cacti calculates the wire capacitance and resistance associated with the driving wire to different parts of a cache-like structure. This equivalent capacitance and resistance helps to model the system energy consumption. Cacti estimates the dynamic power and access time using such models, it also reports the summation of static and dynamic powers. Using Cacti enables computer architects to better understand the performance tradeoffs inherent in different cache sizes and organizations.

Appendix B

Benchmarks

Simulation tools accept a suite of benchmarks as the workload input to evaluate modeled processor. In Appendix B we briefly introduce the standard testbenches used in our research.

B.1 SPEC CPU2000 Benchmarks

The Standard Performance Evaluation Corporation (SPEC) [27] is a non-profit corporation that establishes, maintains and endorses a standardized set of benchmarks. These benchmarks are developed to evaluate and compare the performance of a new high-performance computer with the performance of known reference processors. The SPEC products users are hardware vendors, software vendors, universities, customers, and consultants. SPEC's mission is to develop technically credible system-level benchmarks for multiple operating systems and environments, including high-performance numeric computing, web servers and graphical subsystems. SPEC users agree that benchmark suits are derived from real world applications. Therefore, both computer designers and computer purchasers can make decisions on the basis of the realistic workloads.

Table B.1: SPEC 2000 Integer Benchmarks

Integer Benchmarks (SPECint2000)		
Benchmark	**Language**	**Description**
164.gzip	C	Compression
175.vpr	C	FPGA circuit placement and routing
176.gcc	C	C programming language compiler
181.mcf	C	Combinatorial optimization
186.crafty	C	Game playing: chess
197.parser	C	Word processing
252.eon	C++	Computer visualization
253.perlbmk	C	Perl programming language
254.gap	C	Group theory
255.vortex	C	Object-oriented database
256.bzip2	C	Compression
300.twolf	C	Place and route simulator

Benchmarks presented in SPEC 2000, are made up of two subcomponents that focus on two different types of compute-intensive performance: integer or floating point applications. CINT2000 is used for measuring and comparing compute-intensive integer's performance whereas CFP2000 is used for floating point's performance. Table B.1 and table B.2 present the benchmarks in each group. The tables also show the language which the benchmark is written in.

Table B.2: SPEC 2000 Floating Point Benchmarks

Floating Benchmarks (SPECfp2000)		
Benchmark	**Language**	**Description**
168.wupwise	F77	Physics: quantum chromodynamics
171.swim	F77	Shallow water modeling
172.mgrid	F77	Multi-grid solver: 3D potential fields
173.applu	F77	Partial differential equations
177.mesa	C	3D graphics library
178.galgel	F90	Computation fluid dynamics
179.art	C	Image recognition / neural networks
183.equake	C	Seismic wave propagation simulation
187.facerec	F90	Image processing: image recognition
188.ammp	C	Computational chemistry
189.lucas	F90	Number theory / primality testing
191.fma3d	F90	Finite-element crash simulation
200.sixtrack	F77	Nuclear physics accelerator design
301.apsi	F77	Meteorology: pollutant distribution

B.2 MiBench Embedded Benchmark

Lack of simulation benchmarks which truly represent typical embedded applications inspired proposing a different series of benchmarks for embedded processors. MiBench is an embedded benchmark suite developed by Matthew Guthaus, et al [42]. Evaluating the design using appropriate benchmarks is critical to the design process. Several characteristics distinguish the embedded programs from the existing SPEC benchmarks including instruction distribution, memory behavior, and available parallelism.

A set of 35 applications consist the MiBench embedded benchmarks. These benchmarks are divided into six suites and each suite targets a specific area of the embedded market. The six categories are: automotive and industrial control, consumer devices, office automation, networking, security, and telecommunications. These programs are all written in C. Table B.3 shows the application benchmarks in each six categories.

Table B.3: MiBench Benchmarks

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

B.3 SPLASH-2 multi-processor Benchmark

Stanford Parallel Application for Shared Memory 2nd version (SPLASH-2) is a suite of parallel applications to facilitate the study of centralized (CMP) and distributed multiprocessors [52]. The benchmark suite is organized into two sections: Kernels and Applications. Table B.4 lists the benchmarks in each group.

Table B.4: SPLASH-2 Benchmarks

Kernels	Descriptions	Applications	Descriptions.
FFT	Complex 1D FFT	Barnes	Barnes-Hut
LU	Blocked LU Decomposition	Ocean	Ocean Simulation
Cholesky	Blocked Sparse Cholesky Factorization	Radiosity	Hierarchical Radiosity
Radix	Integer Radix Sort	Raytrace	Ray Tracer
		Volrend	Volume Renderer
		Water (spatial)	Water Simulation with Spatial Data Structure
		Water (nsquared)	Water Simulation without Spatial Data Structure
		FMM	Adaptive Fast Multipole

Appendix C

Dynamic Branch Predictors

C.1 Branch Predictors' Structure

A brief overview of branch predictors is covered in this section. In general, predictors use two type of information: local and global. The local information keeps track of behavior of branches without considering their global behavior. Figure C.1 shows the two architectures proposed for predictors using local information: *bimodal* [54], [24], [53], [21] and *two-level local predictors* [54], [50], [59].

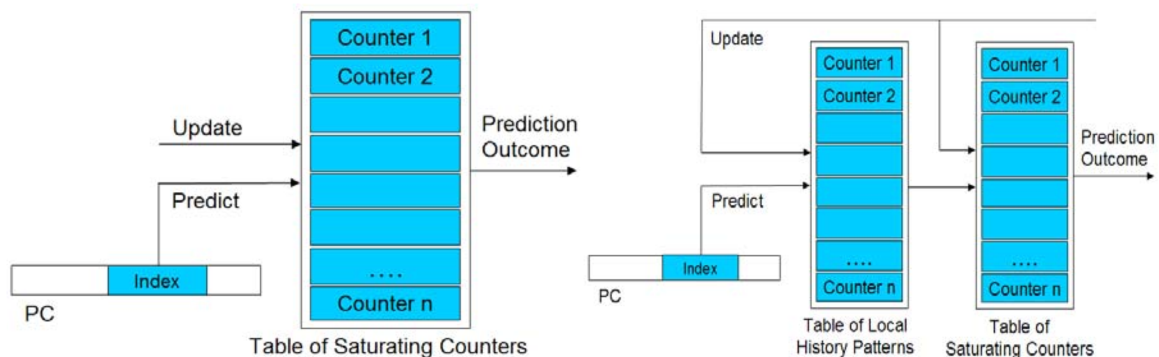

Figure C.1: Local branch predictors (left) Bimodal branch predictor (right) 2-level local history branch predictor

As illustrated, the bimodal branch predictor uses a table of saturating counters, which are accessed by a portion of the program counter (PC), labeled as index. Each counter is incremented if a branch is taken and decremented when the branch is not-taken. These types of predictors work best when branches behave monotonically either taken or not taken. In order to extend the predictor's ability so that it could well predict

all type of branches, a two-level local history branch predictor was proposed, which consists of two tables. The first table is accessed the same way as the bimodal branch predictor is used but the outcome is the history pattern of a given branch, which is used to access the second table. Every entry in the history pattern table is a shift register. The second table has exactly the same structure as the bimodal branch predictor. As a result, the combination of the two tables provides information for a branch that has different patterns in different circumstances. This mechanism works much better for the branches that are not strongly biased toward the taken and not-taken direction. However, the disadvantage associated with this predictor is that the history pattern of the branch should be saved during prediction stage so that the same entry is updated in table of saturating counters when the branch is resolved (the outcome is determined by the functional unit) [54], [49], [50], [59].

The *global predictors* take advantage of a *global history register* (shift register) that keeps track of the prediction of the last n branches. The global predictors work very well for the nested branches and complex programs with regular patterns. The disadvantage of the global predictor is that it has relatively long learning period and suffers from aliasing for the branches that have similar history patterns [54]. Aliasing occurs when two branches with the same PC accesses the same entry in the predictor's table. The simplest type of global branch prediction is depicted in the figure C.2.

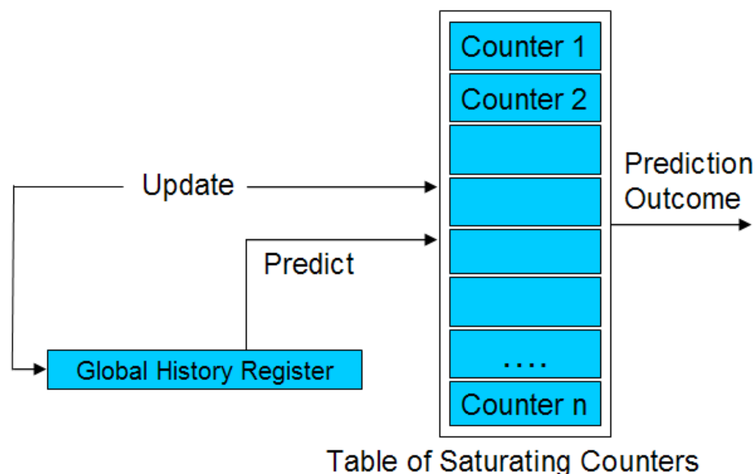

Figure C.2: Global history branch predictor

The global history register is used as an index for accessing the table. Once the result of a branch is resolved the global history register is updated by shifting the result into the shift register.

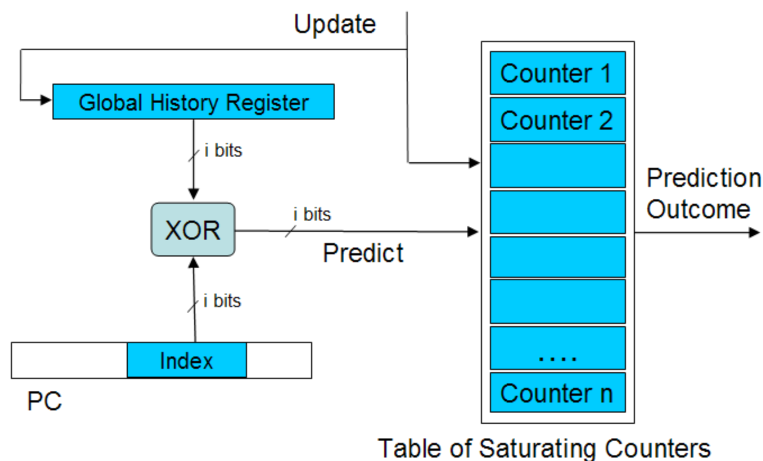

Figure C.3: Gshare Global history predictor

More advanced global branch predictors can be constructed by combining the PC and the global history register. *Gshare* and *gselect* are two of such predictors. The only difference between gshare and gselect is how the address for accessing the table is calculated. Gshare uses the logical exclusive-oring of the portion of PC with the global history register, whereas gselect concatenates them. The idea is to keep track of the

prediction of the same branches that have different history patterns. Figure C.3 shows the structure of gshare predictor.

Since global and local branch predictors are effective in different circumstances, a combined (tournament) branch predictor is proposed [54], [49], [47]. The structure of this kind of predictors is shown in figure C.4.

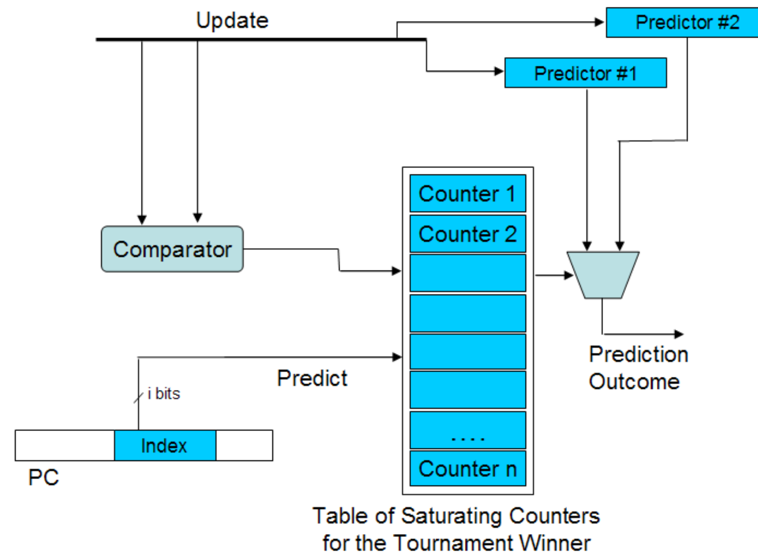

Figure C.4: Combined branch predictor

For every prediction, a tournament table with two bit up/down saturating counters is accessed. If the value of the counter is equal to zero or one, the first predictor (p1) is the winner of the tournament otherwise the second predictor (p2) will be chosen. Once the branch is resolved (the outcome of the branch is known), the following rules are applied to update the tournament table:

- If p2's prediction is correct, the tournament table counter is incremented.
- If p1's prediction is correct, the associated counter is decremented.
- If both predictions are correct or incorrect at the same time, the value of the saturating counter is unchanged.

Combined branch predictors are very effective and have been used in many commercial processors such as Alpha-21264 [54].

Recently, more complicated predictors have been proposed which are based on neural network structure. The *Perceptron* predictor or the *Path-Base neural network* predictors are two examples of such predictors. Although these predictors work under heavy computation workload, their precise prediction urged many designers to still consider them in their research works [13], [6].