Customizable instruction cache hierarchy for embedded systems

Kugan Vivekanandarajah

2005

Kugan, V. (2005). Customizable instruction cache hierarchy for embedded systems.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47483>

<https://doi.org/10.32657/10356/47483>

Nanyang Technological University

Downloaded on 26 Apr 2023 23:06:40 SGT

a402356

Customizable Instruction Cache Hierarchy For Embedded Systems

Kugan Vivekanandarajah

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
In fulfillment of the requirement of the degree of
Doctor of Philosophy

2005

TK
7895
.E42
K95
2005

Acknowledgments

First of all, I would like to take this opportunity to express my sincere gratitude to my supervisor Dr. Thambipillai Srikanthan, for his guidance, help and patience. He has provided me with numerous valuable ideas, insights, and comments throughout the course of this research. He has also been very understanding and supportive. I am very fortunate to have him as my thesis advisor.

I also would like to take this opportunity to thank Dr. Christopher T. Clarke for suggesting several corrections and improvements during many discussions I had with him. And, thanks to Mr. Saurav Bhattacharyya for his helpful suggestions and continuous encouragements. I benefited a lot from them.

Special thanks to Mr. Jagadeesh R. George and Mr. Ashish Panda for agreeing to proof read this thesis, in spite of their busy schedule. Their comments and suggestions made a big difference to this thesis.

I would also like to take this opportunity to thank all the administrative and support-staff at the CHiPES. I am especially grateful to Ms. Nah Kiat Joo, Ms. Merilyn Yap and Mr. Chua Chiew Song for all the help, when it mattered most.

As always, I am grateful to my family and friends for their tremendous support and encouragement. Especially, I thank my mother and father for all the advice that they have offered me over these years and for motivating me to pursue higher-studies.

Many other people contributed to the success of this work and while I would like to acknowledge individually each by name, I would inevitably leave out deserving friends and relatives. I apologize for such omission and convey my respect and admiration to all who contributed to the extraordinary experiences I have been fortunate to enjoy.

Contents

Acknowledgments	i
List of Figures	vi
List of Tables	ix
Abstract	x
1 Introduction	1
1.1 Motivation	1
1.2 Major Contributions of the Thesis	3
1.3 Organization of the Dissertation	3
1.4 List of Publications Resulted from this Thesis	4
2 Literature Review	7
2.1 Introduction	7
2.2 Principles of Memory Hierarchy	8
2.3 Architecture Level Techniques for High Performance	8
2.3.1 Reducing Cache Access Time	10
2.3.2 Reducing Miss-Rate	11
2.3.3 Reducing Miss-Penalty	12
2.4 Architecture Level Techniques for Power Reduction	13
2.4.1 Dynamic Energy Reduction Techniques	14
2.4.1.1 Reducing The Cache Access Energy	15
2.4.1.2 Reducing Bus/Interconnect Energy	21
2.4.2 Static Energy Reduction Techniques	22
2.5 Design Space Exploration for Near-Optimal Cache Hierarchy Design	24
2.6 Summary	25

CONTENTS

3	Filter Cache Based Instruction Hierarchy	27
3.1	Introduction	27
3.2	Simulation Environment for Filter Cache Hierarchy	27
3.2.1	Performance Simulator - SimpleScalar	28
3.2.2	Cache Access Energy Model - CACTI	28
3.2.3	Evaluation Metrics	29
3.2.4	Workload - MediaBench	31
3.3	Performance Evaluations	31
3.3.1	Instruction Cache Hierarchy and Embedded Systems	31
3.3.2	Filter Cache (FC)	33
3.3.3	Performance Evaluations	34
3.4	Predictor Based Filter Cache Hierarchy	36
3.4.1	NFPT Prediction Algorithm	37
3.4.2	Performance Evaluation	42
3.4.3	Limitations of NFPT Prediction Algorithm	42
3.5	Summary	43
4	Pattern Prediction Algorithm for Filter Cache Based Instruction Hierarchy	44
4.1	Introduction	44
4.2	Pattern Prediction Algorithm	44
4.2.1	Prediction Accuracy of Pattern Predictor	49
4.2.1.1	Prediction Accuracy of NFPT vs. Pattern Predictor	50
4.2.1.2	PHT Size on Prediction Accuracy of Pattern Predictor	50
4.2.2	Energy-Delay Product Comparison	51
4.2.2.1	Filter cache Hit Rate	51
4.2.2.2	Filter cache Performance Index	52
4.2.2.3	Average Memory Access Cycles	53
4.2.2.4	Energy-Delay Product	53
4.2.2.5	Optimal Filter Cache Size for Energy-Delay Product Reduction	54
4.3	Static Pattern Prediction Scheme	55
4.3.1	Motivation for Static Pattern Prediction Scheme	57

CONTENTS

4.3.2	Static Pattern Table (SPT)	58
4.3.3	Prediction Accuracy	61
4.3.4	Average Memory Access Cycles	61
4.3.5	Normalized Access Energy	62
4.3.6	Energy-Delay Product	63
4.4	Hardware Complexity Comparison	63
4.4.1	Hardware Area Cost Comparison	64
4.4.2	Pattern Predictor	67
4.5	Summary	70
5	Profile Directed Tuning of Instruction Cache Hierarchy	72
5.1	Introduction	72
5.2	Loop Profiling	73
5.3	Optimizing Filter Cache Size	80
5.3.1	Performance Comparison	86
5.4	Optimizing L1 Cache Size	87
5.4.1	L1 Cache Size Optimization Methodology	87
5.4.2	Performance Comparison	90
5.5	Summary	94
6	Dynamic Filter Cache Management Scheme	96
6.1	Introduction	96
6.2	The Rationale for Dynamic Management of Filter Cache	96
6.2.1	Line Buffer	98
6.2.2	Filter Cache Enable/Disable Mechanism	99
6.2.3	Filter Cache Hit/Miss Predictor	102
6.3	Energy-Delay Tradeoff	102
6.3.1	Average Memory Access Cycles	105
6.3.2	Normalized Energy	105
6.3.3	Normalized Energy-delay Product	108
6.4	Summary	110

CONTENTS

7	A Framework for Customizing Predictor Based Instruction Hierarchy	111
7.1	Introduction	111
7.2	Components of the Framework	111
7.2.1	Selecting The Filter Cache Size	113
7.2.2	Predictor Selection and Configuration	113
7.2.3	Selecting The L1 Cache Parameters	114
7.2.4	Criteria for Invoking Dynamic Cache Management Scheme .	115
7.3	Summary	115
8	Conclusion and Recommendations	118
8.1	Conclusion	118
8.2	Recommendations for Further Research	120
A	MediaBench Benchmark Suite	122
	Bibliography	124

List of Figures

2.1	Performance Growth of CPU and DRAM Main Memory	7
2.2	Direct Mapped Cache Structure	9
2.3	Set Associative Cache Structure	10
2.4	Typical Organization of a Memory Hierarchy	13
2.5	Organisation of Filter Cache and Loop Cache	19
2.6	Predictive Filter Cache	21
3.1	Filter Cache	33
3.2	Average Memory Access Cycles with Filter Cache for MediaBench .	34
3.3	Normalized Energy Consumed with Filter Cache for MediaBench . .	35
3.4	Energy-Delay product with Filter Cache for MediaBench	35
3.5	Predictive Filter Cache	36
3.6	Filter Cache Prediction Process During Instruction Access From Filter Cache	38
3.7	Filter Cache Prediction Process during Instruction Access From L1 Cache	38
3.8	Average Memory Access Cycles with Predictive Filter Cache for Me- diaBench	40
3.9	Normalized Energy Consumed with Predictive Filter Cache for Medi- aBench	40
3.10	Energy-Delay product with Predictive Filter Cache for MediaBench .	41
3.11	An Example Where NFPT Fails	41
4.1	An Example for Hit/Miss Patterns in the Filter cache	46
4.2	FSM of the 2-Bit Saturating Counter	47

LIST OF FIGURES

4.3	Prediction Algorithm	48
4.4	Prediction Accuracy for 256 and 512 Byte Filter Cache	49
4.5	Prediction Accuracy with Different PHT Size	50
4.6	FCHR for 256 and 512 Byte Filter Cache	52
4.7	FCPI for 256 and 512 Byte Filter Cache	53
4.8	Average Memory Access Cycles	54
4.9	Normalized Energy-Delay Product for 256 and 512 Byte Filter Cache	55
4.10	Normalized Energy-Delay Product for 128Byte-2048Byte Filter Cache Sizes	57
4.11	Example for Memory Layout of Tiny Loops	58
4.12	Static Pattern Prediction Algorithm	60
4.13	Prediction Accuracy for 256 and 512 Byte filter cache	61
4.14	Average Memory Access Cycles for 256 and 512 Byte Filter Cache .	62
4.15	Normalized Access Energy for 256 and 512 Byte Filter Cache	63
4.16	Normalized Energy-Delay Product	64
4.17	Simplified Architecture of The Prediction Algorithm	65
4.18	Single Cycle Pattern Prediction Architecture for Filter cache Access Prediction	67
4.19	Experimental Setup for Architecture Validation and Energy Measure- ment of the Prediction Logic	68
5.1	Basic Operations of Loop Profiler	75
5.2	Basic Blocks in a Loop	76
5.3	Computing the Required Cache Size with Basic Blocks (Each Basic Block is Shown with Start and End Addresses)	78
5.4	An Example of a CFG with Predefined Minimum Edge Count (Mini- mum Edge Count of 1000 for the Application “grep”)	79
5.5	Access Energy vs. Cache Size for a Single Loop	82
5.6	Computing the Effective Execution Time of an Outer Loop in Nested Loops	83
5.7	Comparing Relative Access Energy with Calculated and Simulated Values for Filter Cache	84

LIST OF FIGURES

5.8	Energy Reduction with Optimal Filter Cache vs. Calculated Filter Cache for MediaBench Applications	85
5.9	CFG of a Typical Application	88
5.10	Energy Reduction with Optimal vs. Calculated L1 Cache Sizes for MediaBench Applications	91
5.11	Comparing Relative Access Energy with Calculated and Simulated Values for L1 Cache	92
5.12	Leakage Power (W) for Various L1 Cache Sizes	94
6.1	Simple Architecture for Dynamic Filter Cache Management	97
6.2	Line Buffer Enable Signal Generation with Previous Address Bits A_3 And A_2	99
6.3	Simple Loop Detection Based Filter Cache Enable/Disable FSM for Dynamic Filter Cache Management	100
6.4	AMAC Comparison for Dynamic Filter Cache. (AMAC of One Depicts No Performance Penalty)	104
6.5	Normalized Energy Consumption for Dynamic Filter Cache (Includes the Overhead for the Pattern Predictor Logic)	106
6.6	Normalized Energy-Delay Product Comparison for Dynamic Filter Cache	107
7.1	Proposed Techniques for Custom Low-Energy Instruction Hierarchy .	112
7.2	Proposed Framework for Customizable Filter Cache Hierarchy Generation	116

List of Tables

3.1	Statistics for the Most Frequent Tiny Loop for Different Applications	32
4.1	Patterns Used in a Benchmark with 512 Byte Filter Cache	56
4.2	Proposed Static Prediction Table	59
4.3	Hardware Area Cost for Predictor Configurations without Filter Cache	65
4.4	Hardware Complexity Comparison of Predictors	66
4.5	Power Cost Comparisons for Pattern Predictor and Static Pattern Predictor Configurations	69
4.6	Overall Average Percentage Access Cycles Degradation (in %) and Energy-Delay Product Reduction (PEDPR) for Conventional Filter Cache, Pattern Predictor, Static Pattern Predictor and NFPT Predictor	70
5.1	Optimal Filter Cache Size for Energy Reduction	86
5.2	Comparison of Optimal vs. Calculated L1 Cache Sizes for Media-Bench Applications (in KB)	93
6.1	Loop of Application That Cannot be Fully Represented in Filter Cache	109
6.2	Average Percentage Access Cycles Degradation (APACD) and Average Percentage Energy Reduction (APER)	109

Abstract

Cache hierarchy bridges the ever increasing performance gap between the fast processing core and slow main memory. In an effort to improve the total system performance, the high performance microprocessor designers commit more and more transistor budget for cache memory systems. As a result, power consumption of cache memory subsystems has been increasing notably over each processor generation. Although a number of cache optimization techniques to reduce power consumption have been reported in the literature, power consumption in cache memories is a major concern and accounts for as much as 50% of the total processor power. Since the instruction cache consumes the majority of this power, predictor based schemes have been proposed for the instruction cache hierarchy in an attempt to provide for an overall reduction in the energy delay product.

In this research work, several techniques have been proposed to reduce the energy delay product of the instruction cache memory hierarchy. Existing techniques, including the Next Fetch Prediction Table (NFPT) based cache memory hierarchy, have been evaluated thoroughly. A new prediction scheme called ‘pattern predictor’, which is superior to NFPT in terms of VLSI efficiency and prediction accuracy, has been proposed. Techniques to optimize the VLSI architecture of the pattern predictor have been established by examining the relationship between the prediction accuracy and the Pattern History Table (PHT) size for a given filter cache size. Simulations on a Simple Scalar platform show that, for the MediaBench benchmark suite, energy-delay savings of up to 23% over the existing prediction algorithm are possible with the pattern predictor. We have also proposed a static prediction technique to further reduce the VLSI complexity and access energy of the pattern predictor.

One of the main motivations in this research work is to devise efficient techniques to tailor the instruction cache hierarchy for application specific implementa-

LIST OF TABLES

tions. Novel loop profiling based methodologies have been devised to compute near-optimal filter cache and L1 cache sizes. These methodologies incorporate mathematical models to eliminate exhaustive simulations. Extensive simulations were performed using the Simple Scalar platform to fully validate the proposed models. Simulations with the MediaBench benchmark suite show that the proposed methodologies for tuning the cache sizes can provide for up to 47% energy reduction by tuning the filter cache and up to 18% in L1 cache when compared with typical configurations.

Criterion to turn off the filter cache during run-time has been developed after examining the dynamic characteristics of loops. Simulations confirm that, for the MediaBench benchmark suite, the proposed dynamic management scheme can be incorporated into the pattern predictor based instruction cache hierarchy to further reduce the energy delay product by 5.8%. For the MediaBench benchmark suite, the proposed techniques for both tuning and dynamic management have been shown to reduce the overall access energy by 54.45%, while limiting the performance degradation to a mere 0.95%. This represents an overall reduction in the energy-delay product by 54%.

A framework for the systematic customization of the filter cache based instruction cache hierarchy has also been proposed to tailor the pattern predictor based filter cache hierarchy. This provides for an integral environment to realize notable reductions in the energy delay product without incurring exhaustive simulations. The proposed high-speed techniques to customize the filter cache based instruction cache hierarchy provide for an effective alternative to implement a power-sensitive instruction cache hierarchy in embedded systems.

Chapter 1

Introduction

1.1 Motivation

VLSI technologies have been increasing the processor speed and DRAM capacity dramatically. However, they also have introduced a large, growing performance gap between processor and main memory. The limitation of the memory system causes poor total system performance in spite of higher processor performance. Cache hierarchy bridges the performance gap between the processor and DRAM main memory. The disparate growth of processor speed and the DRAM speed makes the performance of cache hierarchy central to the total system performance.

Traditionally performance has been the only focus while designing the cache memory hierarchy. Thus, techniques for aggressive optimizations have been proposed without considering the power consumption. However, recent advancements in process technologies have made power and energy consumption central for the system design; not only for battery operated portable system but also when powered by a direct energy source. Higher energy consumption leads to increased heat dissipation. This demands more expensive packaging and cooling technology, which, in turn increases the cost, and decreases reliability of products in all segments. As a result, techniques for reducing energy consumption have been proposed, including architecture level techniques. However, energy consumption is still a major design issue and has become one of the limiting factors in modern VLSI design.

Power analysis of embedded processors indicate that up to 30% of processor power is consumed in the instruction caches [69]. Earlier experimental analysis indicates that,

1.1 Motivation

if the appropriate data memory hierarchy mapping techniques are applied first [23, 53], and if all methods to reduce power in the datapath are applied [14], this number goes up to 50% [5]. Hence, reducing this part of the budget is crucial in reducing the overall power consumption of the system.

The auxiliary cache based instruction hierarchy is one of the effective means to reduce the energy-delay product. Filter cache, an auxiliary cache, based memory hierarchy is effective in reducing the energy-delay product in the instruction cache hierarchy. Filter cache relies on frequent tiny loops for energy reduction in the instruction caches, at the cost of some performance. This is particularly advantageous for embedded applications, as most of the execution time is spent in tiny loops known as computational kernels. Minimizing the performance penalty associated with the filter cache is essential for it to be used in power sensitive embedded systems, which also demand high performance. Therefore, predictive filter cache is proposed to limit the performance penalty, in the filter cache hierarchy.

The performance of the hit/miss prediction algorithm dictates the effectiveness of the predictive filter cache. The performance improvement obtained with the predictive filter cache might not be sufficient for certain applications. Thus, it is necessary to further reduce the energy-delay product in the instruction cache hierarchy, while reducing the performance penalty to negligible levels. In addition, the optimal cache hierarchy which results in maximal energy savings depends on the characteristics of the application. Therefore, to realize maximum energy savings, the filter cache hierarchy has to be designed in an application specific manner. The widely used exhaustive simulation based design space exploration methodology increases the design time and design cost, which is not acceptable for embedded system.

Therefore, in this thesis, we have researched ways to improve the performance penalty associated with the filter cache hierarchy, while maintaining the energy-delay product reduction with the filter cache hierarchy. In addition, we also have studied loop profiler directed methodologies for tuning filter cache hierarchy, without the need for the time consuming exhaustive simulations.

1.2 Major Contributions of the Thesis

1.2 Major Contributions of the Thesis

This section identifies the novel algorithms, techniques and methodologies that have been made during the course of this research and documented within this thesis.

The contributions in this thesis can be summarized as:

1. Performance evaluation of filter cache hierarchy with an NFPT prediction algorithm for a single issue processor (Chapter 3).
2. A novel prediction algorithm called pattern prediction algorithm for the filter cache hierarchy that is superior to NFPT in terms of prediction accuracy and VLSI cost (Chapter 4).
3. The proposal of a static prediction algorithm to further minimize the hardware complexity (Chapter 4).
4. A tailored loop profiling platform for profiling of applications (Chapter 5).
5. A loop profiling directed methodology for the identification of the optimal filter cache size (Chapter 5).
6. A loop profiling directed methodology for the identification of the optimal L1 cache size (Chapter 5).
7. A filter cache management scheme for dynamic management of the filter cache (Chapter 6)
8. A framework for the systematic configuration of the filter cache based hierarchy (Chapter 7)

1.3 Organization of the Dissertation

This thesis is organized into eight chapters with each chapter dedicated to a specific topic.

In Chapter 1 the motivations and major contributions of this thesis are outlined.

1.4 List of Publications Resulted from this Thesis

In Chapter 2 a detailed review of the existing state of the art architecture level techniques for low-power high performance instruction cache hierarchy design is presented.

In Chapter 3 filter cache based instruction hierarchy is investigated as a means to reduce the energy consumed in the instruction cache hierarchy in embedded processors. The limitations of a filter cache based instruction hierarchy are explained.

In Chapter 4 an alternate filter cache prediction algorithm, known as pattern predictor, capable of improved prediction accuracy, with very minimal hardware complexity is proposed. In addition, a simplified version of the pattern predictor known as static pattern predictor, to further reduce the hardware complexity, is also proposed.

In Chapter 5 a simple loop profiling directed methodology to tune the filter cache and L1 cache for maximizing energy savings is proposed. The proposed methodology, utilizing the loop profile, arrives at the near optimal filter cache and L1 cache size, without the need for exhaustive simulation.

In Chapter 6 a simple loop detection based dynamic filter cache management scheme to further reduce the energy-delay product in the filter cache hierarchy is proposed. The proposed filter cache management scheme dynamically detects the usefulness of the filter cache and is able to enable/disable it, with minimal hardware complexity.

In Chapter 7 a unified framework to generate custom low-power instruction hierarchy based on the proposed techniques is presented. Thus, the custom filter cache based instruction cache hierarchy can be realized without increasing the design time and cost.

Chapter 8 concludes this work and suggests future research directions.

1.4 List of Publications Resulted from this Thesis

International Journal Papers

- K. Vivekanandarajah, T. Srikanthan and S. Bhattacharyya, "Energy-delay efficient filter cache hierarchy using pattern prediction scheme", IEE Proceedings - Computers and Digital Techniques, Vol. 151, Issue 2, March 2004.

1.4 List of Publications Resulted from this Thesis

- K. Vivekanandarajah, T. Srikanthan, C.T. Clarke and S. Bhattacharyya, "A Novel Static Prediction Scheme for Filter Cache Structures", IEICE Transactions on Electronics, Special issue on "Low-Power System LSI, IP and Related Technologies", Vol.E87-C, No.4, pp.543-548, April 2004.

International Conference Papers

- K. Vivekanandarajah, and T. Srikanthan, "Custom Instruction Filter Cache Synthesis for Low-Power Embedded Systems" 16th IEEE International Workshop on Rapid System Prototyping, pp. 607-610, Alberta, Canada, June 2005.
- S. Bhattacharyya, T. Srikanthan and K. Vivekanandarajah, "Area and Power Efficient Pattern Prediction Architecture for Filter Cache Access Prediction in the Instruction Memory Hierarchy", IEEE International Symposium on VLSI Design, Automation and Test, pp. 345-348, Taiwan, April 2005
- K. Vivekanandarajah, T. Srikanthan and S. Bhattacharyya, "Dynamic Filter Cache for Low Power Instruction Memory Hierarchy", EUROMICRO Symposium on Digital System Design, IEEE CS Press, Rennes, France, pp. 607-610, August/September, 2004.
- K. Vivekanandarajah, T. Srikanthan and S. Bhattacharyya, "Decode Filter Cache for Energy Efficient Instruction Cache Hierarchy in Super Scalar Architectures", in proceedings of the IEEE/ACM Asia South Pacific Design Automation Conference, (ASP-DAC'04), Page(s): 373-379, Yokohama, Japan, January 2004.
- K. Vivekanandarajah, T. Srikanthan, C. Clarke and S. Bhattacharyya, "Static Pattern Predictor (SPP) Based Low Power Instruction Cache Design", in Proc Embedded Systems and Applications Conference 2003 (ESA'03), Las Vegas, Nevada, USA, pp.210-215, June 2003.
- K. Vivekanandarajah, T. Srikanthan, S. Bhattacharyya and P.V. Kannan, "Incorporating Pattern Prediction Technique for Energy Efficient Filter Cache Design", in Proc. 3rd IEEE International Workshop on System-on-Chip for

1.4 List of Publications Resulted from this Thesis

Real-Time Applications (IWSOC 2003), IEEE CS Press, Alberta, Canada, pp.44-47, June/July, 2003.

Chapter 2

Literature Review

2.1 Introduction

VLSI technologies have been increasing processor speed and DRAM capacity dramatically. However, they also have introduced a large, growing performance gap between processors and main memory (DRAM).

By improving not only the clock speed but also instruction level parallelism (ILP), the processor performance has been improving at a rate of 60% per year. On the other hand, the access time to DRAM has been improving at a rate of less than 10% per year[55]. Moreover, current memory systems suffer from a lack of memory bandwidth caused by I/O-pin bottleneck. This problem is well known as “Memory Wall” [78]. The limitation of memory systems causes poor total system performance in spite of

Figure 2.1: Performance Growth of CPU and DRAM Main Memory

2.2 Principles of Memory Hierarchy

higher processor performance.

Cache hierarchy bridges the performance gap between the processor and DRAM main memory. The disparate growth of processor speed and the DRAM speed makes the performance of cache hierarchy central to the total system performance.

2.2 Principles of Memory Hierarchy

Cache hierarchy works based on the principles of locality, namely temporal and spatial locality[55]. That is, programs tend to reuse data and instructions they have recently referenced. In addition, empirical observation shows that a program spends 90% of its time in 10% of its code, which is often known as the 90-10 rule[55]. Capitalizing on these observations, cache hierarchies try to achieve the average performance of the high-speed memory with the average cost of the cheapest memory.

Cache structures buffer the recently accessed contents, thus they will have a miss rate associated for a given application. The misses in cache memories can be divided into capacity, compulsory and conflict misses (excluding the cache coherence misses which will be present in multi processor systems[55]). Capacity misses result if the cache can not contain all the blocks needed during the execution of the program. Compulsory misses are due to the very first access that can not be present in the cache. And, conflict misses result if a block is discarded and later retrieved because too many blocks have mapped to the same cache set. The design goal of cache hierarchy is to improve the memory access performance by reducing the time taken to fetch the data or instruction.

2.3 Architecture Level Techniques for High Performance

From the miss rate point of view, a bigger cache with higher associativity will improve the performance of the cache system. However, increasing the cache size also leads to increase in the cache access time, thus the reduction in the average access time involves a trade off between miss rate and access time. Equation 2.1 shows the average access

2.3 Architecture Level Techniques for High Performance

Figure 2.2: Direct Mapped Cache Structure

time for the cache hierarchy.

$$T_{avg} = T_{hit} + M \times T_{penalty} \quad (2.1)$$

Where T_{hit} is the cache access (or hit) time, M is the miss rate and $T_{penalty}$ is the cost incurred on a miss (the miss penalty).

$$T_{penaltyL(i)} = T_{hitL(i+1)} + M_{L(i+1)} \times T_{penaltyL(i+1)} \quad (2.2)$$

Where $T_{penaltyL(i)}$, $T_{hitL(i)}$ and $M_{L(i)}$ represent the miss penalty, the access time and miss rate respectively for the i^{th} -level cache.

As shown in the Equation 2.1, it can be understood that the average memory access time in the memory system can be improved by reducing the cache access time, reducing the cache miss rate and reducing the miss penalty. The following text summarizes the state of the art techniques proposed for realizing this.

2.3 Architecture Level Techniques for High Performance

Figure 2.3: Set Associative Cache Structure

2.3.1 Reducing Cache Access Time

Hit time and the cycle time in a cache memory can be reduced by using a smaller, less associative cache structure. However, smaller, less associative cache memories will have relatively higher miss rate.

On the other hand, set associative caches have fewer conflict misses, as the cache line can reside in multiple ways within a set. Thus, set associative caches will have lower miss rate with relatively smaller cache size. However, set associative caches suffer from longer access time, as the way selection has to be performed after the tag comparison (See Figure-2.2 and Figure-2.3). This is a concern, especially in the first level cache memories. Therefore, to decrease the critical access time, speculative way selection is proposed, where, a way is speculatively selected. If the first examination finds the way, the cache access time will be as fast as direct mapped cache. In the worst case, the speculative cache access will take more than one cycle. Speculative way selection can be done either using locality information[18, 36] or partial tag comparison[46]. As expected, when the way prediction is wrong, it can result in

2.3 Architecture Level Techniques for High Performance

performance penalty.

Another technique to improve the hit time is to pipeline the cache writes. That is, write hits usually take longer than read hits because the tags must be checked before data can be written. Thus, pipelining the writes will make the write access time faster.

2.3.2 Reducing Miss-Rate

As shown in Equation 2.1, reducing the cache miss rate also reduces the average memory access time. The miss rate of the cache depends on cache size, associativity, line size and cache replacement policy. Increasing the cache size decreases the capacity misses, thus, decreases the miss rate till capacity misses becomes zero. Increasing the cache associativity also decreases the miss rate, as increasing the associativity decreases the conflict misses. However, in general, increasing the cache size and associativity also increases the cache access time, thus involve tradeoff with the cache access time.

Cache line size also influences the cache miss rate. That is, increasing the cache line size results in exploiting spatial locality, thus, reduces the compulsory misses. However, as shown in Equation 2.3, increasing the line size also increases the miss penalty, thus involves tradeoff. In merged DRAM/logic system, dynamically variable line size cache[30] also was proposed to optimize its line-size according to the characteristic of the program, and attempts to improve the performance by exploiting the high on-chip memory bandwidth. In this case, the line size is adopted for varying amount of spatial locality within and among programs at run time.

$$T_{Main_Memory} = T_{DRAM_array} + \frac{Line_Size}{Band_Width} \quad (2.3)$$

Where T_{Main_Memory} , T_{DRAM_array} , $Band_Width$ and $Line_Size$ stands for the time to load a cache line, DRAM latency, memory band width and cache line size.

Direct mapped cache structures have smaller access time with higher miss rate. Thus, to improve the miss rate, pseudo associative cache is proposed so that hit time of the direct mapped cache is achieved with the miss rate which is equal to that of set associative cache. That is, in cache memories, cache lines are mapped to the set using a mapping function. If large amount of cache lines are mapped to a set, it will lead

2.3 Architecture Level Techniques for High Performance

to conflict misses. Thus, different cache mapping techniques were used to improve the distribution of cache lines to various sets as in set associative cache. For instance, hash-rehash cache attempt to avoid conflict misses by two different hashing functions, where, the first mapping function that is same as direct mapped cache is used. If the first search does not succeed the second mapping function is used to search the line. Variations of these schemes are proposed including column associative cache[6], sequential and parallel multicolumn cache[83], group-associative cache[56] and reactive associative cache[10].

Victim cache is also an effective means to reduce the overall miss rate without affecting the cycle time[33]. In this set up, a small fully associative cache is placed in-between the cache and its refill path. Victim cache contains the blocks that are evicted from the cache and is checked on a miss. Thus, it reduces conflict misses with a less associative cache.

Another way to decrease the misses is to employ prefetching, where instructions and data are prefetched, before the CPU requests for it, into the cache or an external buffer that can be accessed more quickly than main memory. Prefetching reduces the compulsory misses as it fetches data before the processor requests for it. Prefetching can be done by using either software[17, 40] or hardware approaches[87]. Software prefetching uses the compiler to transform the code; usually by adding extra explicit fetch instructions to instruct the hardware which information is to be prefetched. Prefetching relies on utilizing the bandwidth that otherwise would be unused.

2.3.3 Reducing Miss-Penalty

As shown in the Equation 2.1, reducing the cache miss penalty is also crucial for reducing the average memory access time. Thus, multiple levels of cache memories are used to reduce the average access time in a typical high performance memory hierarchy. The first level will match the clock speed or the cycle time of the processors. Other levels will exist so that the miss penalty is minimized.

Miss penalty also can be minimized by giving read misses priority over write misses[55], by using sub block placement policy[55], by using early restart[55], critical word first and non-blocking caches[20].

2.4 Architecture Level Techniques for Power Reduction

Figure 2.4: Typical Organization of a Memory Hierarchy

We have seen the established architecture level techniques for improving the performance in a cache hierarchy, to this point. There has been substantial amount of research focused on improving the performance of the memory hierarchy and myriad of techniques have been proposed to improve the performance. In general, the memory hierarchy design techniques depend on many factors and require tradeoff with cycle time of the processor and VLSI area. Nevertheless, the recent advancements in process technologies, though provides transistors in abundance, has made power and energy consumption the limiting factor in VLSI design. Thus, energy consumption also must be considered from the beginning while designing the memory hierarchy.

2.4 Architecture Level Techniques for Power Reduction

Increasing the cache capacity or increasing the hit rate reduces the frequency of off-chip accesses due to improved cache hit-rates. Though, this approach seems to be useful in that the energy dissipated for driving external I/O pins is reduced, this will increase the on-chip energy consumption due to frequent accesses to the bigger cache memory. For instance, the on-chip cache for the Strong ARM SA110 consumes approximately 43% of the total chip power[48].

Figure-2.4 shows the typical organization of a two-level cache hierarchy. As can be expected from the organization, memory hierarchy dissipates energy in CPU on-chip

2.4 Architecture Level Techniques for Power Reduction

bus interconnect, on-chip and off-chip cache memories, off-chip bus interconnects and main memory.

Energy dissipation in integrated circuits can be divided into static and dynamic components. The dynamic component, also known as switching power, arises because of charging and discharging of capacitive gates and wires, thus, depends on activity of the component. Static component, which arises due to sub threshold leakage of reverse biased junctions and gate oxide leakage, depends on the technology (including various other factors) and will be dissipating energy as long as the supply voltage for the transistors are on. In general, static energy consumed is proportional to the number of transistors used. Cache memories, in typical high end embedded processors, account for major portion of the transistor budget[39]. Thus, they consume substantial amount of both static and dynamic energy. Higher energy consumption leads to increased heat dissipation. This demands more expensive packaging and cooling technology, which, in turn increases cost, and decreases reliability of products in all segments of the computing space; from portable systems to high-end servers. Therefore, there exists a strong need for power and energy aware design of cache hierarchies.

With energy consumption becoming the major design issue, several architecture level techniques have been proposed to achieve overall energy reduction. The following section summarizes the state of the art dynamic and static power reduction techniques.

2.4.1 Dynamic Energy Reduction Techniques

Equation 2.4 presents the average access energy for cache hierarchy with single level cache memory.

$$E_{Average_Memory_Access} = E_{Hit} + M \times E_{Main_Memory_Access} \quad (2.4)$$

Where $E_{Average_Memory_Access}$, E_{Hit} , M and $E_{Main_Memory_Access}$ stand for the average memory access energy, access energy when instruction/data is hit in the cache, Miss rate of the cache and the access energy for main memory respectively.

2.4 Architecture Level Techniques for Power Reduction

As can be seen from the equation, the average dynamic memory access energy in the cache hierarchy can be reduced by reducing the cache access energy and maintaining the miss rate or by reducing the energy consumed by the bus interconnect in the event of cache miss. In this thesis, we are interested only in cache memory hierarchy, thus techniques for reducing the energy consumption in the DRAM memory are not considered. The following subsection presents the state of the art low energy techniques for dynamic energy reduction in the cache memories and the bus interconnects.

2.4.1.1 Reducing The Cache Access Energy

Equation 2.5 presents the dynamic energy dissipation of CMOS technology circuits.

$$E_{Dynamic} = 0.5 \times C \times V_{dd}^2 \quad (2.5)$$

Where, V_{dd} is the supply voltage and C is the total effective capacitive load. In case of cache access, C is the total capacitance of all the cache components such as bit-lines, word lines, memory cells, and so on. Thus, to reduce the cache access energy and to maintain the miss rate, one has to reduce the total effective capacitance C . In cache memory, with architecture level techniques, it can be achieved using compressed caches, partitioning the cache structure, behavioral optimization of the cache memories and/or by using auxiliary cache based memory hierarchy. The following section summarizes these techniques.

Using Compressed Caches : Instructions and data in a typical system exhibit large amount of redundancy. Thus, the contents of the cache can be compressed to achieve the same miss rate with relatively smaller cache sizes. Typical requirement of the compression algorithm is such that, it should be possible to decompress part of the instruction/data on the fly and it should not introduce too much overhead in terms of power and performance. Instructions and data in cache memory exhibit different kinds of redundancy. Thus, different compressions schemes are formulated to exploit these redundancies.

The compiler emits instructions from high level language using scheme called syntax directed translation, hence a typical instruction stream will have same instruction blocks repeated in many places. Therefore, some compression schemes break the program into blocks of size n , where n corresponds to the cache line size. A suitable

2.4 Architecture Level Techniques for Power Reduction

algorithm compresses each block individually. A Line Address Table (LAT) is used to record the address of each compressed cache line within the compressed program[76]. Cache Lookaside Buffer (CLB) is added to alleviate LAT lookup cost[76]. During program execution, upon each cache miss, LAT entry cached in CLB is used to calculate the compressed cache line address. The most commonly used algorithms for block compression are variations of Huffman and arithmetic that compresses serially within the block, byte wise. A few of such algorithms can be found in[45, 76, 82]. Alternatively, application is profiled offline for the most frequent instructions. Once the most frequent instructions are known, N most frequent instructions could be compressed into $\log_2 N$ bits. The program is stored in the compressed format ($\log_2 N$ bits binary), which has one to one correspondence with the original instruction. Every time an instruction is fetched it is first decompressed by means of the Instruction Decompression Table (IDT) and then passed to the processor decoding logic. This architecture is motivated by the fact that software programs normally use only a subset of the all possible instructions offered by the processor's instruction set.

On the other hand, in the data caches, over 70% of the bits that are read from or written to are zeros[71]. Thus, dynamic zero compression for caches was proposed in[71], where, to reduce data cache energy dissipation by compressing bytes, the all zeros values are represented by a single bit and accessed. On the other hand, another technique called sign compression and cache line bisection takes advantage of the high occurrence of all-ones and all-zeros. These all-ones and all-zeros primarily occur because of the large number of small values typically stored in an L1 data cache. In order to take advantage of it, additional hardware is embedded in the cache RAM array to manage the reading and writing of compressed values. This scheme also provides overflow bytes so that we can utilize the unused portion of the cache from the first scheme[37].

Though the cache compression results in better memory bandwidth utilization and lower cache energy, it also adds sizeable overhead in terms of decompression and cache management logic for the compressed data. In addition, compressed cache also results in performance penalty as it has overhead associated with managing and decompression of the contents.

Partitioning the Cache Array : One of the major sources of power dissipation in a conventionally organized cache can be attributed to transitions in the bit lines of

2.4 Architecture Level Techniques for Power Reduction

the data and tag arrays. Bit line dissipations occur when the bit lines are precharged or discharged. Thus, this technique partitions the cache array into smaller modules so that, while accessing the cache, only part of the smaller modules need to be accessed, resulting in lesser effective cache capacitance for cache accesses. This can be done using horizontal partitioning and/or vertical partitioning[32].

In typical set associative cache access, all the cache ways corresponding to the index part of the address are activated, thus, when the tag comparison is done, the way corresponding to the tag is sent out immediately by the multiplexer. And also, the complete cache line is enabled, though in most cases ultimately a single word is read out[55].

In cache sub banking[34, 61], a horizontal partitioning scheme, the cache data array is sub banked horizontally. Thus, with the use of offset part of the address, only the sub bank with the referenced data is activated. Since this is done in parallel to the tag comparison, no cache access time overhead is incurred. In addition, bit line partitioning[25, 61], a vertical partitioning technique, achieves energy savings by reducing the active bit-line.

This organization, an effective scheme, does not compromise the cache cycle time (and other aspects of performance, such as the cache hit ratio and cache access rate). However, in modern Deep Sub-Micron Process (DSM) process techniques, the energy savings possible with this scheme is often not adequate and needs to be applied with other techniques.

Behavioral Optimization of the Cache Memories : In typical cache memory, in an effort to match the cycle time of the processor core, all the ways in a set associative caches are searched in parallel. Each way consists of one tag sub array and one data sub array. Thus, while accessing for the data, energy is consumed in all the ways due to the parallel access and only one way has the data referenced by the processor. Thus, phased caches, way prediction caches and way memorization caches have been proposed to optimize the cache accesses of the set associative cache.

In the phased caches, tag comparison and the data access are done in series so that the unnecessary data array accesses are avoided at the cost of increased cache access time[28]. In way predicting cache[31], only one way is speculatively selected using Most Recently Used(MRU) information[18]. If the way prediction is wrong, other remaining ways are searched in parallel. The MRU based way predicting has

2.4 Architecture Level Techniques for Power Reduction

prediction accuracy of 85%[18], thus will require 2 cycles to access the remaining 15% of the time, resulting in noticeable performance penalty. Way memorization cache avoids unnecessary tag comparisons by recording previous tag reference. That is, in typical set associative caches, tag comparison is performed for every access. Thus, to reduce the unnecessary tag comparisons, a conditional tag comparison scheme is proposed[54], where if two successive instructions i and j resides in the same cache line, the tag comparison for the j can be avoided. The conditional tag comparison caches reduce the energy consumed in tag comparison when the spatial locality is high.

Direct addressed cache[75], a hardware-software design for an energy-efficient microprocessor data cache, allows software to access cache data without a hardware cache tag check. These tag-unchecked loads and stores save the energy of a tag check when the compiler can guarantee an access will be to the same line as an earlier access

Auxiliary Cache Based Memory Hierarchy : Another class of approach for reducing the average effective energy per access is to add an unusually small (64 to 512 byte) cache, in the cache memory hierarchy between the L1 cache and the processor core. Various auxiliary cache hierarchies have been proposed to alleviate the problem of increased energy dissipation in the instruction cache hierarchy. These auxiliary cache structures exploit the spatial and temporal locality within the small loops of applications.

Line buffer[61], loop cache [12, 44] and filter cache [39] are examples of such cache memories. Scratch pad memory[9], another compiler managed memory, has been proposed for reducing the energy consumed in the data memory hierarchy. Energy savings in this auxiliary cache comes from accessing the energy efficient auxiliary cache, which is physically placed close to the execution core. They perform in slightly different ways to extract the energy savings possible by introducing a small cache between the execution core and main cache.

In instruction caches, due to the high spatial and temporal locality exhibited by loops in the instruction stream, these auxiliary structures are accessed more often. This is particularly advantageous for embedded applications as most of the execution time in embedded applications is spent in tiny loops known as computational kernels[21, 72].

Power analysis of embedded processors indicates that up to 30% of processor power is consumed in the instruction caches[69]. Earlier experimental analysis indicate[5]

2.4 Architecture Level Techniques for Power Reduction

Figure 2.5: Organisation of Filter Cache and Loop Cache

that, if the appropriate data memory hierarchy mapping techniques are applied first [23, 53], and if all methods to reduce power in the datapath are applied [14], this number goes up to 50%. Hence, reducing this part of the budget is crucial in reducing the overall power consumption of the system. The auxiliary cache based hierarchy is one of the effective means to achieve this, as it results in substantial energy-delay product reduction. The auxiliary caches such as loop cache and filter cache are an effective means to reduce the energy-delay product.

In the loop cache [12, 44], the controller detects small loops by detecting a control flow change caused by any branch instruction having a short negative offset, that is, a short backwards branch, or *sbb*, instruction. A *sbb* could be any existing branch instruction and not a special or new instruction. A detected *sbb* would trigger the filling of the tiny cache, which is called a loop cache, during the next iteration of the small loop. The fill occurs non-intrusively as the processor continues to fetch and execute from regular instruction memory. Upon detecting the triggering *sbb* again, the loop cache controller would switch instruction fetching over to the loop cache. Fetches would continue from the loop cache until the triggering *sbb* was encountered but not taken, meaning the loop was being exited. If the entire loop did not fit in the loop cache, then the cache would be filled completely with the first part of the loop, and fetching would switch back and forth between the loop cache and regular instruction memory.

2.4 Architecture Level Techniques for Power Reduction

A control flow or cof instruction (other than the triggering *sbb*) is encountered within a small loop would immediately terminate the loop cache filling or fetching. In other words, only loops with straight-line execution were supported. Filling the loop cache in the presence of branches is difficult, because we usually will not see all the loop instructions during an iteration of the loop, as the branches cause some loop code to be skipped. Thus, a preloaded loop cache[27], which is a variation of the dynamic loop cache was introduced. Also, one-level hybrid loop cache and a two-level hybrid loop cache[27], each of which combines features of dynamic and preloaded loop cache are proposed to tackle these problems. Still, in this case also, the entire loop should fit in the loop cache. Thus, this scheme is not capable of complete energy savings with the temporal locality in tiny loops.

Filter cache[39], the most effective in energy reduction, has been shown to achieve 58% power saving at the cost of 21% performance loss. Filter cache adds a tiny cache, in the order of 256 byte, in the first level near the processing core, as shown in Figure-2.5. Though the energy reduction possible with the filter cache is significant, the performance degradation associated with filter cache might not be acceptable in most cases. Adding predictive[67, 68] capability, as shown in Figure-2.6, to the filter cache hierarchy reduces the performance penalty associated with the hierarchical access of the filter cache. The prediction accuracy is central to the working of the predictive filter cache. Though the Next Fetch Prediction Table (NFPT) based predictive filter cache reduces the performance penalty, for certain cases it predicts wrongly. And also, since the information needed for prediction is maintained in a table and is proportional to the filter cache size, the hardware complexity of the predictor can become high. If the energy consumption is high, it can marginalize the energy-delay product reduction possible with the filter cache. Thus, there is a need for a predictor that is capable of higher prediction accuracy with minimal hardware complexity.

Another approach proposed to improve the performance in filter cache based hierarchy is dynamic cache management scheme[11]. In the dynamic cache management scheme, L0 cache or filter cache contain only the most frequently executed basic blocks. For achieving it, this scheme uses branch prediction confidence estimation. That is, a branch that is predicted taken with high confidence is expected to be taken during the program, in the particular phase of the program. If it is not taken, it will be considered to be behaving unusually. If the program behaves unusually, it will probably

2.4 Architecture Level Techniques for Power Reduction

Figure 2.6: Predictive Filter Cache

drive the thread of control to a portion of the code that is not very frequently executed. And, only the basic blocks that are executed frequently will be accommodated in the filter cache. Thus, only the most frequent basic blocks such as basic blocks from loops will be cached in the filter cache. This scheme disables and enables the filter cache based on the unusual behaviors of the branches. However, since this scheme relies on sophisticated branch prediction schemes for enabling and disabling the filter cache, it cannot be used in embedded processors, where these sophisticated branch prediction schemes are often not used. In addition, it has been shown that the performance of this scheme is inferior to the predictive filter cache[68]. Nevertheless, filter cache being small relies on small loops for energy savings. When these loops are not executed, filter cache may increase the performance penalty. Thus, it is expected that a loop detection based dynamic cache management scheme with a predictive filter cache can be effectively used to reduce the performance penalty. Thus, there is a need for a loop detection based filter cache management scheme to improve the performance.

2.4.1.2 Reducing Bus/Interconnect Energy

Apart from the cache memory, bus/interconnect also consume considerable amount of energy. The intra-memory interconnect energy consumption and the energy spent amongst the on-chip inter-block wires like memory buses can be very high if not optimized properly. This is due to their relatively large switching capacitance and high

2.4 Architecture Level Techniques for Power Reduction

activity. Due to the intrinsic capacitance of the bus lines, a considerable amount of power is required at the I/O pins of the microprocessor[13]. More specifically, it has been estimated that the capacitance driven by I/O pins is much larger than the one seen by internal nodes [13]. As a consequence, optimization of average power can be achieved by minimizing the number of transitions. Bus encoding techniques are proposed in recent literature such that the bus switching is minimized when data is transferred. Numerous encoding techniques have been proposed to minimize the switching for various data set[7, 13, 19]. A redundant encoding scheme called bus-invert code is proposed[60], which performs well when the data to be transferred is of random nature. Hence, it works well for data hierarchy. For transferring address busses, gray coding was proposed to encode the patterns as it is guaranteed to produce only single bit transition[62], as the addresses will have higher spatial locality. A number of techniques rely on introducing redundancy to save power. That is, these techniques add one or more bits[15] so that effective bus switching can be minimized. The encoding logic, decoding logic and the redundancy introduces overhead in terms of area and performance. However, bus-encoding schemes are effective in reducing the energy consumption, especially for off-chip accesses, and therefore, need to be used to with other cache energy reduction techniques.

2.4.2 Static Energy Reduction Techniques

Until very recently, only dynamic power has been the significant source of power consumption. However, with DSM technology, as smaller geometries exacerbate leakage, static power is also starting to dominate the power consumption equation in microprocessor design, necessitating power and energy aware design techniques in all levels of the design flow. While dynamic energy will still remain a concern, leakage energy also has to be thought of in the DSM process technology.

Various circuit level techniques have been proposed to tackle the leakage energy when the component is not in use. One such technique known as gated-Vdd[57] is integrated into the cache structures to dynamically shutdown portions of the cache. This technique is applied at the cache block granularity[79]. These techniques assume that the state of the cache memory can be lost. And also, gated-Vdd adds an additional

2.4 Architecture Level Techniques for Power Reduction

transistor to the memory cells, thus, will make the memory cell slower, and big in terms of area.

Another technique, called cache decay[35], turns off the cache lines if a pre set number of cycles have elapsed since its last access. This is based on the observation that cache lines see a flurry of use when brought in, and then a period of dead time between the last access and the point when a new data item is brought in. A key aspect of this technique is the desire to balance potential for savings in leakage energy against the potential for incurring the extra level two or main memory access energy.

Turning off the cache is not the only way that leakage can be reduced. Significant leakage reduction also can be achieved by putting a cache line into low power drowsy mode[24]. In circuit level, adaptive body-biasing with multi threshold CMOS, where threshold voltage is changed dynamically, is used to put the memory cell in drowsy mode. For data caches, cache lines are periodically put into drowsy mode without regard to their access history[24] and woken up when they are used. For instruction caches, the cache is divided into sub banks and at any given time only one sub bank is in active mode. Although, it is known that the transition from drowsy mode to normal operation mode introduces energy and performance penalty, a table based next sub bank prediction is used to hide part of this[38] in instruction cache.

Customization of architecture to the needs of the application is one of the most effective ways to tackle dynamic as well as static energy reduction without impacting the performance. This is especially advantageous in embedded systems which are often application specific. And also, core-based processors allow a designer to choose a particular cache configuration [1, 2] and there are instances of processor designs that allow their caches to be configured during system reset or even during runtime[8, 85]. Techniques such as selective cache ways[8], way halting cache[85], way concatenation cache[85] enable tuning of cache parameters to the needs of an application at runtime just by changing the configuration register.

2.5 Design Space Exploration for Near-Optimal Cache Hierarchy Design

2.5 Design Space Exploration for Near-Optimal Cache Hierarchy Design

Tuning cache parameters to the needs of a particular application can save power and energy (both static and dynamic). Architecture tuning is the process of customization of architecture to most efficiently execute a particular application (or set of applications) under given constraints on size, performance, power and energy. Every application has different cache requirements that cannot be efficiently satisfied with one predetermined cache configuration. Miss rates or energy consumption may be unnecessarily high if a cache configuration does not fit the application. And the cache size should reflect the working set of the application. If the cache is too large, the energy required to fetch from the larger cache may be unnecessarily high. If the cache is too small, excess energy may be wasted due to thrashing. The diversity among applications leads to very different cache requirements for different applications[85].

However, the major problem with the architecture tuning is that it is not feasible to simulate all possible combinations of processor and cache from the specified space. Because of the multi-dimensional design space, the total number of possible designs can be exceedingly large. The design space will be exceedingly large when design space exploration needs to be done for a cache hierarchy involving multiple cache parameters, thus, requiring higher design time and design cost.

Traditionally simulation based exhaustive search methods are used to explore the optimal cache designs, where a design is simulated with all the possible parameters and analyzed to find the optimal cache hierarchy. But the time required for an exhaustive search is often prohibitive. When the design space becomes too large, iterative heuristics[84] are used to reduce the design space needed to be explored, so that the near-optimal cache configuration is reached without actually simulating the entire configuration. The most crucial aspect to this design methodology is the simulation speed. Thus, some techniques focused on reducing the time required by using trace compression and /or simulating multiple configurations[29, 63] in a single pass. In trace compression, a reduced trace is obtained which approximates the behaviors of the original trace. Techniques are formulated to find approximate[42] or lossless trace reduction[58, 73, 77] to improve the simulation speed.

2.6 Summary

Another class of approach uses analytical design space exploration, where certain characteristics are extracted from the application and used to either find the miss rate for given cache parameters or, to find the optimal cache parameters for given set of requirements. In data caches, source code analyzing method is proposed to determine the optimal data cache size by analyzing array access patterns of the application[51]. This was later extended to decide upon the distribution of the on-chip data memory into scratchpad memory and data caches[52]. In addition, an analytical algorithm also was proposed to analyze the data memory references and it is guaranteed to satisfy designer provided performance constraints[26]. Reuse distance[86] was also used to estimate the capacity miss rates of a fully associative cache[86]. These techniques are mostly targeted towards data caches.

2.6 Summary

Traditionally performance has been the only focus while designing the cache memory hierarchy. Thus, techniques for aggressive optimizations have been proposed without considering the power consumption. However, recent advancements in process technologies have made power and energy consumption central for the system design, not only for battery operated portable system but also when powered by direct energy source. Higher energy consumption leads to increased heat dissipation. This demands more expensive packaging and cooling technology, which, in turn increases the cost, and decreases reliability of products in all segments. As a result, techniques for reducing energy consumption have been proposed, including architecture level techniques. However, energy consumption is still a major design issue and has become the limiting factor in modern VLSI design.

Instruction caches in embedded systems consume considerable amount of the total energy. The auxiliary cache based hierarchy contains the tiny loops close to the processing core, thus, reduces the access energy in the instruction caches. This is particularly advantageous for embedded applications, as most of the execution time in embedded applications is spent in tiny loops known as computational kernels. Loop cache and filter cache exploit the temporal locality for reducing the energy consumed in the instruction cache hierarchy. Loop cache, though does not have any performance limitation, does not exploit the full access energy savings, as loops which can not be

2.6 Summary

fully contained is not cached. Filter cache, another auxiliary cache, shows potential for high energy-delay product reduction at the cost of some performance. Minimizing the performance penalty associated with the filter cache hierarchy is crucial for power sensitive embedded systems, which also demand high performance. Thus, reduction in the performance penalty in the filter cache hierarchy, while maintaining the energy-delay product reduction is desirable. Techniques to customize the cache hierarchy of the embedded systems through application profiling will provide for reduction in the energy-delay product. In addition, very little work has been reported to reflect the benefits of dynamic management in cache hierarchy.

Chapter 3

Filter Cache Based Instruction Hierarchy

3.1 Introduction

It is evident from chapter 2 that the energy consumption in the instruction cache hierarchy must be reduced, especially in the case of embedded systems. The filter cache based hierarchy has been shown to be effective in reducing the energy-delay product in the instruction cache hierarchy. Filter cache was initially proposed for instruction and data caches in embedded processors. Later, predictive filter cache was introduced to reduce the performance penalty in instruction filter cache, for a wide-issue superscalar processor. Thus, to understand the effectiveness of the filter cache and predictive filter cache for the instruction cache, we implemented the published techniques with the widely used simulation platforms. The next section presents the simulation environment. The subsequent section evaluates the performance of the filter cache and predictive filter cache configurations.

3.2 Simulation Environment for Filter Cache Hierarchy

This section presents the performance simulator, the model for cache access energy calculation, the workload used and the performance metrics used for evaluation.

3.2 Simulation Environment for Filter Cache Hierarchy

3.2.1 Performance Simulator - SimpleScalar

We use SimpleScalar [16], an instruction set simulator, in all our simulations to extract the cache access statistics. The SimpleScalar tools are widely used in computer architecture research [4]. For example, more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs [4].

The SimpleScalar tool set could be used to model different processor architectures with different Instruction Set Architecture (ISA). SimpleScalar also supports multi-level cache architecture with each cache level using its own cache policies. It support replacement policies of LRU, Random and FIFO, with various other cache parameters such as line size, associativity, write back/ write through configurations.

The processor model we simulated is a single issue processor, typically found in a general purpose embedded processor, with ARM ISA, where instructions are fetched from the cache hierarchy every clock cycle. If an instruction cache access results in a miss then it leads to a pipeline bubble. Also, the first level instruction access is in the critical path. L1 cache used in the experiment is 32 KB with a 32 Byte line size and a 4-way set-associativity. This setup is used in all the experiments as the base case so that the experimental results are compatible to the instruction cache used in high-end embedded microprocessors [48]. In addition, all our simulations are done with the LRU cache replacement policy.

3.2.2 Cache Access Energy Model - CACTI

For the access energy computation, we use CACTI ver-3.0 [74]. CACTI is an integrated cache access time, cycle time, area, aspect ratio, and power model. Integration of all these models together allows the users to study the tradeoffs between time, power, and area based on the same assumptions and hence the results are mutually consistent. It takes as input the cache capacity, associativity, cache line size, the number of read/write ports, and the feature size. It uses analytical models to compute the access time and the energy consumed by the cache for different configurations. The model divides the data and tag array into number of sub arrays such that the configuration has

3.2 Simulation Environment for Filter Cache Hierarchy

the best access time and energy consumption, as determined by its optimization function. The results obtained with CACTI are found to be close to the values obtained from post-layout simulations [85].

3.2.3 Evaluation Metrics

For comparing the combined energy and performance metrics we use energy-delay. This is because many approaches can be used to reduce power by sacrificing arbitrary amounts of performance. By introducing an arbitrary reduction in the clock rate, we can achieve an arbitrary reduction in the power consumed by CMOS circuits. Clearly, power alone is not a good design metric, and must be evaluated along with some concern for performance. Thus, we use the Energy-Delay product metric, which has been used for CAD tools and circuit designs, to evaluate both energy and performance at the same time. In this thesis, as in earlier research [39], we present the energy-delay product as the main metric for comparing cache configurations.

In addition, reducing the performance penalty also plays a crucial role in reducing the energy consumed in the rest of the processor. Although the filter cache hierarchy reduces the instruction access energy in the instruction fetch, it introduces performance penalty, as the filter cache misses take an additional cycle to fetch instructions from the L1 cache. Let's call this performance penalty introduced by the filter cache as $T_{penalty}$. If the processor core consumes E_{core} power without the filter cache, it will now consume $(1 + T_{penalty}) \times E_{core}$. The filter cache reduces the dynamic power by a factor S , thus the new energy consumption will be $S \times E_{Dynamic}$. In addition, adding filter cache marginally increases the leakage power in the instruction cache hierarchy. Thus, the new leakage power will be equal to $(1 + T_{penalty}) \times \frac{L1_Size + FC_Size}{L1_Size} \times E_{Static_IC}$.

The effect of the filter cache based instruction hierarchy, in the overall energy consumed, can be evaluated with the following equations, where Equation 3.1 presents the energy consumed for a processor architecture without the filter cache and Equation 3.2 presents the new energy consumption with the filter cache. As it is evident from the equations, reducing the performance penalty is vital to achieve overall energy reduction in the processor.

$$E_{Conv.Processor} = E_{Dynamic_IC} + E_{Static_IC} + E_{Core} + E_{Rest} \quad (3.1)$$

3.2 Simulation Environment for Filter Cache Hierarchy

$$E_{New_Processor} = S \times E_{Dynamic_IC} + (1 + T_{penalty}) \times \left(\frac{L1_Size + FC_Size}{L1_Size} \times E_{Static_IC} + E_{Core} + E_{Rest} \right) \quad (3.2)$$

Thus, we also report the average memory access cycle (or performance) and the access energy separately so that we can also compare the performance and energy consumption separately. The Instructions Per Cycle (IPC) can also be compared to study the performance tradeoffs. However, since IPC is a combined metric which accounts for processor stalls due to data control hazards, and also depends on the processor micro architecture, it is not possible to study the effect of just the filter cache. Thus, as in previous research [39, 67], we resort to using the average memory access cycles as our performance metric. In our discussions, we will present average memory access cycles, normalized access energy, and normalized energy-delay product. For the base case, we use a single level cache hierarchy with 32 KB with 32 bytes line size and 4-way set-associative instruction cache.

For example, if an application access the filter cache N times and, it resulted in N_{Hit} and N_{Miss} then average memory access cycles, normalized access energy, and normalized energy-delay product will be as shown in Equation 3.3, Equation 3.4 and Equation 3.5 respectively. It should be noted here that for the filter cache studies that we don't consider the misses in the L1 cache, as we are comparing the {filter cache + L1 Cache} hierarchy with the conventional L1 cache hierarchy.

$$Average_Memory_Access_Cycle(AMAC) = \frac{N_{Hit} + 2 \times N_{Miss}}{N} \quad (3.3)$$

$$Normalized_Access_Energy(NAE) = \frac{N_{Hit} \times E_{FC} + N_{Miss} \times E_{FC} + N_{Miss} \times E_{L1(128Bit)}}{N \times E_{L1(32Bit)}} \quad (3.4)$$

$$Normalized_Energy_Delay_Product(NEDP) = NAT \times NAE \quad (3.5)$$

Where E_{FC} and E_{L1} stands for filter cache and L1 cache access energy respectively. The filter cache based hierarchy imposes an overhead in the L1 caches. That is,

3.3 Performance Evaluations

instead of reading a one instruction, 16 bytes of instructions must be read. Hence, in our models we use the L1 cache access with 32-Bit output for the base case and the L1 cache access with 128-Bit output for the L1 cache in the filter cache based hierarchy.

3.2.4 Workload - MediaBench

For evaluating the effectiveness of the low power techniques, we need to validate it with a workload that represents that of typical embedded applications. In our experiments we use applications from the MediaBench [43] suite as our workload. Applications from the MediaBench suite can be viewed as small-embedded programs or computation kernels that might be found as a part of a larger embedded program. Each benchmark comes with its own example input and expected output. The MediaBench suite represents the workload of emerging multimedia and communications systems. It also represents the concept of portable embedded applications written in high-level languages, as processor architects and software developers are moving in this direction. MediaBench also has been widely used in published literature as the workload of choice to represent the embedded applications. Appendix A shows the benchmarks we used in the experiments from MediaBench. All the applications were compiled with gcc 2.93 and were run to completion.

The next section presents the results for filter cache and predictive instruction filter cache based instruction hierarchy with the MediaBench suite.

3.3 Performance Evaluations

Filter cache exploits temporal locality within tiny, frequent loops for achieving energy reduction. Thus, we present the loop characteristics of the embedded application to confirm the insight behind the filter cache hierarchy before we delve into filter cache and predictive filter cache in detail.

3.3.1 Instruction Cache Hierarchy and Embedded Systems

Cache memories work on the principles of locality, which is partly due to the loops found in the application. In embedded applications, most of the execution time is spent

3.3 Performance Evaluations

Benchmark	Loop Size (inst)	No of Time Executed	% of total time
ADPCM-DEC	111	147520	99
ADPCM-ENC	153	147520	99
EPIC	44	154448	72
UNEPIC	63	21748	31
G721.DEC	122	590080	45
G721.ENC	124	590080	36
GSM.TOAST	159	1180160	35
GSM.UNTOAST	148	1180160	77
MIPMAP	12	62499	34
OSDEMO	79	287915	64
TEXGEN	257	26880	58
MPEG2.DEC	14	3784703	32
MPEG2.ENC	158	1832160	59
PEGWIT	39	168928	38
PEGWIT.DEC	39	168960	37
PEGWIT.ENC	32	328661	29
JPEG.DEC	17	33300	14
JPEG.ENC	7	173848	10

Table 3.1: Statistics for the Most Frequent Tiny Loop for Different Applications

in a small amount of code. This code segment is called the critical code segment. These critical segments consist of tiny loops called computational kernels that are executed repeatedly. It has been shown that in embedded applications, these critical code segments, which account for 1% of the total instructions, execute on an average for 50 % of the time [72]. In applications such as GSM speech coding, ADPCM coding, image compression and decompression, MPEG and G.721 voice coding one simple kernel loop typically accounts for a substantial amount of the execution time [21]. Table 3.1 lists the frequent tiny loop statistics for the MediaBanch [43] benchmark suite, which also confirms the published results. Thus, capitalizing on these observations, auxiliary cache based hierarchies are proposed for dynamic energy savings in the instruction memory hierarchy.

3.3 Performance Evaluations

Figure 3.1: Filter Cache

3.3.2 Filter Cache (FC)

Filter cache [39], which is an auxiliary cache, achieves better energy-delay product by trading some performance. Filter cache has been shown to achieve a 58% energy saving at the cost of a 21% performance loss for the MediaBench [43] benchmark suite. Filter cache achieves energy savings by accessing a small energy efficient cache more often than the main cache. In doing so, it takes advantage of the immediate spatial and temporal locality exhibited by the frequent tiny loops in the instruction stream.

As we have seen from the Table 3.1, the embedded applications spent a substantial amount of time in small loops. Thus, keeping these loops in the small filter cache close to the processing core reduces the dynamic energy consumption. However, since filter cache is now in the first level of the memory hierarchy, when the accessed instruction is not found in the filter cache, it will result in a one-cycle access penalty to fetch the instruction from the L1 cache. Since, instruction accesses show a high amount of temporal and spatial locality, instruction filter cache exhibits a higher hit rate and a substantial energy reduction. However, the performance penalty associated with the filter cache hierarchy is high, even though it results in a substantial energy-delay product reduction. The next subsection presents the results for instruction filter cache.

3.3 Performance Evaluations

3.3.3 Performance Evaluations

Figure 3.2: Average Memory Access Cycles with Filter Cache for MediaBench

Figure 3.2 shows the average memory access cycles for instruction accesses (average memory access cycles of one depicts the case where filter cache is not present) with MediaBench applications in the filter cache hierarchy. Figure 3.3 shows the normalized access energy (normalized access energy of one depicts the case where filter cache is not present) for the same benchmarks. In addition, Figure 3.4 shows the normalized energy-delay product (normalized energy-delay product of one depicts the case where filter cache is not present). As can be seen from the figures, filter cache results in a substantial energy-delay product reduction while suffering a performance penalty.

With the 256-byte filter cache, filter cache hierarchy results in a 43.42% energy reduction with 14.14% performance penalty resulting in a 33.87% energy-delay product reduction for instruction cache accesses. For the 512-byte filter cache, the results are 55.97% energy reduction with 7.61% performance penalty resulting in a 52.11% energy-delay product reduction. It should be noted here that we are evaluating just the instruction filter cache along with bigger instruction caches to reflect the recent trends in commercial instruction cache usage [48], thus our results are slightly different from the published results [39]. However, they demonstrate the effectiveness of the filter cache hierarchy for energy reduction in the instruction hierarchy.

3.3 Performance Evaluations

Figure 3.3: Normalized Energy Consumed with Filter Cache for MediaBench

Figure 3.4: Energy-Delay product with Filter Cache for MediaBench

3.4 Predictor Based Filter Cache Hierarchy

Though the energy-delay product reduction achieved due to the filter cache is significant, the performance degradation introduced by having to access the filter cache and the main cache in the sequential manner might not always be acceptable. Therefore, it was noted that having a predictor based parallel access path between processor core and the filter cache or main cache could reduce the performance penalty to acceptable levels.

3.4 Predictor Based Filter Cache Hierarchy

Figure 3.5: Predictive Filter Cache

In predictive filter cache [68] as shown in Figure 3.5, predictive parallel access path exists from the processor core to the filter cache and L1 cache. Incorporation of the predictor leads to four operating scenarios. The following section considers the latency and energy tradeoffs for a prediction-based hierarchy.

1. Predicted location: FC; Actual location: FC: In this case, we will get the power savings expected with filter cache and there will not be any performance penalty. The instruction will be fetched in one cycle with the least energy requirement.
2. Predicted location: FC; Actual location: L1: The prediction was to access the filter cache, but turned out to be a miss, so a second cycle is used to access the L1 cache. Here, there will be a penalty of an extra one-clock cycle to access

3.4 Predictor Based Filter Cache Hierarchy

the instruction compared to the case where filter cache is not present. This case is similar to filter cache without predictor, as anyway we would have accessed the instruction in the same hierarchical order. However, if the predictor were more accurate then, it would have predicted an access to the L1 cache. Thus limiting false accesses to the filter cache is one of the operational priorities of the predictor involved. If a predictor is statistically found to be predicting the filter cache wrongly, then it is filter cache biased.

3. Predicted location: L1; Actual location: FC: The prediction was to access the L1 cache, but the instruction wanted is (also) already present in the filter cache. In this case, we are not taking the advantage of having the filter cache, so this will result in higher instruction access energy compared to the case where filter cache is used without predictor. Meanwhile, the possible gain of this arrangement of filter cache with a predictor is lost due to the inaccuracy of the predictor. In this case, we term the predictor to be L1 cache biased.
4. Predicted location: L1; Actual location: L1: The prediction was to access the L1 cache, and the instruction required is not in the filter cache. In this case, access energy will be slightly less than the conventional filter cache based configuration and the access time will be one cycle better than the filter cache without predictor. This will be equivalent to the case where filter cache is not present.

Ideally the predictor should not have any of cases (ii) and (iii). The next section explains the Next Fetch Prediction Table (NFPT) [67, 68] algorithm presented for predictive filter cache.

3.4.1 NFPT Prediction Algorithm

The Next Fetch Prediction Table (NFPT) [67, 68] based predictor is introduced for predictive access of the filter cache, limiting the performance degradation to 1.3%, at energy savings of 31.5% for the instruction cache with spec95 benchmark suite[68], in a wide-issue superscalar processor whose fetch width is equal to the filter cache line size (which is 4 instructions or 16 Byte). NFPT table based predictor predicts whether the instruction line to be fetched will be present in the filter cache or not, in

3.4 Predictor Based Filter Cache Hierarchy

the previous cycle. Hence, based on this prediction the fetch unit will attempt to fetch the instruction either from the filter cache or L1 cache (as predicted).

The NFPT based predictor captures temporal locality within small loops to predict whether the next instruction is expected to be in the filter cache. To predict, it utilizes the fact that while executing tiny loops the same control path might be taken repeatedly. The algorithm for prediction is that, if the tag of the current fetch address is equal to the tag of the predicted next fetch address based on the previous control path (i.e. based on the next fetch entry in the NFPT for the corresponding cache line), then a hit in the filter cache is predicted. Otherwise the main cache is accessed. The assumption here is that if the control path is in a small loop, the tags of the current fetch address and the predicted next fetch would be equal. Only if it is part of a small loop would it remain in the cache until the next iteration, otherwise it would be flushed out.

Figure 3.6 and 3.7 shows the architecture, where the NFPT table maintains the tag portion of the next fetch address for each instruction line stored in the filter cache. Thus, when an instruction line is fetched from the filter cache, the corresponding NFPT entry is compared to check if the next fetch is going to be a hit or miss in the filter cache. To keep the NFPT table small, the four least significant bits of the tags are stored in the NFPT table. Still, the NFPT size needed for the filter cache prediction is dictated by the filter cache size and increases with increasing filter cache size.

The NFPT was extended for single-issue processors, where the NFPT is used to predict hit/miss in the instruction cache line accesses. That is, for the current fetch address $\{\text{fetch_addr}\}$, the most likely next fetch address is $\{\text{fetch_addr} + 4\}$. If $\{\text{fetch_addr}\}$ and $\{\text{fetch_addr} + 4\}$ map to the same cache line, the filter cache will be the next fetch source. If $\{\text{fetch_addr}\}$ and $\{\text{fetch_addr} + 4\}$ map to different cache lines, the partial tag in the NFPT entry indexed by $\{\text{fetch_addr}\}$ is compared with the lowest 4 bits of the tag part of $\{\text{fetch_addr}\}$. If the values are equal, the next fetch source is filter cache and vice versa. We implemented the NFPT based predictive filter cache and evaluated the energy-delay results with MediaBench suite. The following subsections present the results for the predictive filter cache with the MediaBench suite.

3.4 Predictor Based Filter Cache Hierarchy

Figure 3.6: Filter Cache Prediction Process During Instruction Access From Filter Cache

Figure 3.7: Filter Cache Prediction Process during Instruction Access From L1 Cache

3.4 Predictor Based Filter Cache Hierarchy

Figure 3.8: Average Memory Access Cycles with Predictive Filter Cache for Media-Bench

Figure 3.9: Normalized Energy Consumed with Predictive Filter Cache for Media-Bench

3.4 Predictor Based Filter Cache Hierarchy

Figure 3.10: Energy-Delay product with Predictive Filter Cache for MediaBench

Figure 3.11: An Example Where NFPT Fails

3.4 Predictor Based Filter Cache Hierarchy

3.4.2 Performance Evaluation

Figure 3.8 compares the average memory access cycles for instruction accesses with MediaBench applications in the predictive filter cache hierarchy and conventional filter cache hierarchy. Figure 3.9 compares the access energy for the same benchmarks and the Figure 3.10 compares the normalized energy-delay product. With the 256-byte filter cache, predictive filter cache hierarchy results in a 42.99% energy reduction with 3.29% performance penalty resulting in a 40.86% energy-delay product reduction. For the 512-byte filter cache, the predictive filter cache results are 54.98% energy reduction with 1.76% performance penalty resulting in a 54.14% energy-delay product reduction. As can be seen from the figure, predictive filter cache reduces the performance penalty associated with the sequential access in the conventional filter cache. Overall, predictive filter cache hierarchy reduces the performance penalty associated with the filter cache to 3.29% and 1.76% for 256 and 512 byte filter cache respectively, for the benchmarks simulated.

This demonstrates that the predictive filter cache is effective in reducing the performance penalty associated with the filter cache. As a result, predictive filter cache achieves higher energy-delay product reduction compared to the conventional filter cache. It is also clear that, in general, the 512 byte filter cache results in a higher energy-delay product compared to the 256 byte filter cache as it can contain more frequent loops compared to 256 byte filter cache. However, increasing the filter cache indefinitely will increase the filter cache access energy and reduce the overall energy-delay product reduction. In the next chapter we will study the effect of changing the filter cache size on energy-delay product reduction, in detail.

3.4.3 Limitations of NFPT Prediction Algorithm

Although the NFPT scheme generally works well, there are instances where it does not predict correctly. For instance, a loop consisting of two instruction lines with a procedure call in the second line that spans into two instruction lines is executed with a filter cache having four instruction lines. When this loop executes, though the filter cache can contain all the four instruction lines, the NFPT based predictor will predict them to be not there as the tags are different. Here we assume that the procedure call is not inlined by the compiler and the procedure call is located in such a way that the

3.5 Summary

tag of the instruction line corresponding to the procedure call are different from the tag of the main loop instruction line. Figure 3.11 shows a part of an application as pseudo code, which illustrates the scenario we discussed. This has a loop with three functions that are placed in the memory addresses as shown in the figure. If the size of the loop is smaller than the filter cache (of 256 Byte), then except for the first iteration, the instruction can be supplied from the filter cache. However, for the instruction lines with the “function call” and “function return”, the NFPT will predict incorrectly as the tag part will be different.

As we have seen earlier, an improvement in the prediction accuracy not only improves the performance of the memory hierarchy; it also reduces the energy consumed in the rest of the processor. Thus, in the subsequent chapters, we will study various ways to improve the filter cache performance.

3.5 Summary

Filter cache based instruction hierarchy has been shown to reduce the instruction fetch energy at the cost of performance. Predictive filter cache, based on the next fetch prediction table, has been proposed to limit the performance penalty associated with the filter cache. Though, the next fetch prediction table scheme reduces the overall performance penalty, it does not work well for some cases, due to the inherent limitation of the NFPT prediction scheme. In addition, the hardware complexity of the predictor increases with the filter cache size leading to an increase in the VLSI cost and power consumption. In the next chapter, we will present alternate prediction algorithms that are both hardware efficient and superior to NFPT.

Chapter 4

Pattern Prediction Algorithm for Filter Cache Based Instruction Hierarchy

4.1 Introduction

As discussed in chapter 3, NFPT is not an ideal candidate for energy-delay product reduction. Hence, an alternate prediction scheme known as pattern predictor is proposed in this chapter to overcome the limitations. In addition, we have also proposed a simplified prediction algorithm known as the static pattern prediction algorithm to further minimize the hardware complexity. We evaluated the hardware complexity of the proposed techniques to further exploit its design flexibilities. The next section presents the pattern prediction algorithm for the filter cache predictor. In the subsequent section we present a simplified version of the pattern predictor to further reduce the hardware complexity.

4.2 Pattern Prediction Algorithm

As the issue rate and depth of the pipelining in a high-performance superscalar processor increases, various branch prediction algorithms have been proposed to keep the pipeline busy with useful instructions. The two-level adaptive branch prediction algo-

4.2 Pattern Prediction Algorithm

rithm is effective and widely used for predicting the branch direction. The two level adaptive predictors have been researched quite extensively for branch prediction and numerous variants of this have been proposed [80]. These studies have identified the predictor and the configuration that best predicts the branches in a given set of benchmarks. These mechanism uses two levels of branch history information to make the prediction, the history of the last k branches encountered, and the branch behavior of the last s occurrences of the specific patterns of these k branches. These two levels of information are stored in the branch history register and pattern history table. These binary predictors have also found widespread applications [81]. The two-level adaptive predictor can be meaningfully implemented in three different ways. They are:

1. “GAg” with Global branch history register and global pattern history table: all the branch predictions are based on the same global history register and global pattern history table. Thus, this scheme requires very minimal hardware to implement. However, its prediction accuracy could be worse compared to the other schemes due to the interferences from other branches in the global pattern history table and shift register. This global technique works particularly well when the direction taken by sequentially executed branches is highly correlated [47].
2. “PAg” using Per-address branch history and global pattern history table: in order to reduce the interference in the first level branch history information, one history register is associated with the each distinct static conditional branch. Since, this scheme uses a global pattern history register, interference still exists in the pattern history table.
3. “Pap” using Per-address branch history and per-address pattern history table: in order to completely remove the interference, each static branch has its own pattern history table. However, the hardware complexity of this scheme is substantially higher compared to that of the “GAg” scheme. The local technique works well for branches with simple repetitive patterns [47].

Some branch’s outcome depends on previous branch direction; as a result, the global scheme works well for this. And, in some branches, the branches exhibit simple repetitive patterns. Therefore, some techniques combined the global local branch

4.2 Pattern Prediction Algorithm

Figure 4.1: An Example for Hit/Miss Patterns in the Filter cache

prediction [47]. In this thesis, we are interested in using two-level adaptive predictor for predicting the hit/miss for the instruction line access in the filter cache.

The hit/miss patterns of the instruction line accesses exhibit similar patterns when the instructions from loops are executed. If the loop is fully containable in the filter cache, the patterns are going to be continuous hits. Similarly, if the loop cannot be containable in the filter cache till the next iteration, the patterns are going to be continues misses. Additionally, when a loop which is only partially contained in the filter cache is executed, it will exhibit repeated regular hit/miss patterns. Thus, the hit/miss patterns of instruction line accesses, when executing a loop, exhibits regular patterns as in correlated branch directions. Therefore, these hit/miss patterns can be predicted with two-level adaptive predictors.

For example, if we look at Figure 4.1, a loop with 6 instruction lines and a 4 line filter cache, the instruction lines {1,5} have conflicting interferences. Similarly, {2,6} have conflicting interferences. Thus, assuming instructions 1 to 6 are executed for each iteration, it is going to exhibit a “..0011001100110011..” pattern repeatedly, where 1 stands for the filter cache line hit and 0 stands for the filter cache line miss.

Since, the two-level predictor is used for predicting hit/miss patterns of the instruction lines while executing loops, interference of patterns is minimal. This is because

4.2 Pattern Prediction Algorithm

the small filter cache contains only the loop that is currently being executed. Even when the patterns interfere, these loops will not be active together. Thus, the patterns will be adapted to the new loop after few wrong predictions. Therefore, in our setup, we use the “GAg” scheme due to its ability to predict these patterns with higher prediction accuracy and its simplicity. The “GAg” predictor consists of a global shift register (SR) to record the hit/miss patterns of instruction lines and a global pattern history table (PHT) consisting of 2 bit saturating counters. When the previous N outcomes of hit/miss are known, by indexing the pattern history table (PHT) with the value we can predict whether the next one is going to be hit or miss. The size of the PHT (2^N entries) is determined by the size of the SR (N bits) used. The finite state machine (FSM) shown in Figure 4.2 depicts the PHT updating and prediction process. We call this variant of the “GAg” predictor for predicting the hit/miss of the instruction access as pattern predictor in our discussions.

Figure 4.2: FSM of the 2-Bit Saturating Counter

Figure 4.3 illustrates the algorithm employed in the prediction process. Drawing from the insight provided by branch prediction research, we assume that the next instruction fetch address will be the current fetch address plus “4” (here each instruction is 32-bits wide and we assume that most branches are not taken, so that next instruction address is current instruction address plus four Bytes). We access the predictor only

4.2 Pattern Prediction Algorithm

Figure 4.3: Prediction Algorithm

when cache line changes are encountered. That is, since instructions have a higher spatial locality, we predict a hit in the filter cache if the cache line being accessed is unchanged. In the event of a change, we access the PHT corresponding to the SR value, and the prediction of a hit or a miss in the filter cache is made depending on whether the PHT value is above or below the pre-set threshold. For a two-bit saturating counter, a threshold value of 2 is used. When the outcome of the prediction is known, the SR and PHT values are updated accordingly, as in the normal branch predictor scheme.

4.2 Pattern Prediction Algorithm

In the next section we present the results for the pattern prediction algorithm. Our experiment results are grouped into two sections of importance. In the first section, we discuss the prediction accuracy of the predictors. Prediction accuracy has the highest impact in the energy savings and the average memory access latency. In this section, the prediction accuracy of the pattern prediction algorithm is compared with NFPT for 256 and 512 Byte filter cache sizes. Then, investigations on the ability of the pattern predictor to cater to the varying requirements of the application, in an attempt to improve the prediction accuracy, are performed. In the next section, we discuss the energy-delay product. Energy-delay product provides for unified quantification of the effectiveness of the architecture. In this section, the factors affecting the energy-delay product measure are first discussed. In the discussions that follow, we show that the energy-delay product of a pattern prediction based filter cache is lower than that based on NFPT. Then, the effect of filter cache size on the energy-delay product is presented.

4.2.1 Prediction Accuracy of Pattern Predictor

Figure 4.4: Prediction Accuracy for 256 and 512 Byte Filter Cache

4.2 Pattern Prediction Algorithm

4.2.1.1 Prediction Accuracy of NFPT vs. Pattern Predictor

The pattern predictor with 32 entry PHT was employed at first for 256 Bytes and 512 Bytes of filter cache memory as suggested in [39, 68]. Figure 4.4 compares the prediction accuracy of NFPT and Pattern predictor for 256 Byte and 512 Byte filter cache sizes, where the prediction accuracy is defined as:

$$\text{Prediction_Accuracy} = \frac{(\text{TotalPredictedFC} : \text{ActualFC} + \text{TotalPredictedL1} : \text{ActualL1}) \times 100}{\text{TotalPredictions}} \quad (4.1)$$

This metric compares the prediction performance of both the prediction algorithms. As it can be seen from the figure, for 256 Byte filter cache, the pattern predictor always results in a higher prediction accuracy than the NFPT predictor. For the 512 Byte filter cache, the pattern predictor results in higher prediction accuracy except for ADPCM.ENC. For the benchmarks ADPCM.ENC, the performance of pattern predictor is comparable to NFPT. The relationship between the filter cache size, PHT size and the prediction accuracy shall be further investigated in the following section.

4.2.1.2 PHT Size on Prediction Accuracy of Pattern Predictor

Figure 4.5: Prediction Accuracy with Different PHT Size

4.2 Pattern Prediction Algorithm

Having established the advantage of the pattern predictor over NFPT with the prediction accuracy, we investigate the effect of varying PHT size on prediction accuracy. Figure 4.5 shows that, in general, the prediction accuracy increases marginally as we increase the PHT size. It is also evident that, some benchmarks benefit more from having bigger PHT than the others. When the applications have many active loops, increasing the PHT size may result in improved performance. This is not so for benchmarks (e.g. ADPCM) with only few active loops. For such cases, higher prediction accuracy is observed with smaller SR values. For a 256 Byte filter cache, EPIC, UN-EPIC, OSDEMO etc. show that the prediction accuracy marginally improves along with the increase in the PHT size. Based on our findings, we can safely assume that the prediction accuracy with 8-entry and 32-entry PHT is generally comparable and in most cases better than that obtained using the NFPT prediction scheme. This is because, in embedded applications, few tiny loops account for most of the execution time of the application. Thus, to represent these loops, we need only few entries in the PHT. In addition, since we are caching only tiny loops in the filter cache, the hit/miss patterns they exhibit are similar for most of the loops. Therefore, we choose the PHT size to be 32 to generate all the results in the subsequent sections. In the next section we study the factors which influence the energy-delay product.

4.2.2 Energy-Delay Product Comparison

4.2.2.1 Filter cache Hit Rate

Figure 4.6 compares the Filter Cache Hit Rate (FCHR) which is calculated as the fraction of the instruction accesses which are hit in the filter cache. This metric evaluates the effectiveness of the filter cache for saving energy. A conventional (hierarchical) filter cache architecture will always have the maximal FCHR as all the instructions are first checked in filter cache. As for the predictor based implementation, with 256 Byte and 512 Byte filter cache configurations, pattern predictor results in very marginally inferior hit rate with benchmark OSDEMO compared to NFPT. This is despite the fact that pattern predictor results in higher overall prediction accuracy (i.e. predictions for both hits and misses) for this benchmark. This is because most of the incorrect predictions by the pattern predictor tend to be filter cache miss-predictions, whereas most of the NFPT's incorrect predictions are filter cache hit-predictions. As discussed earlier,

4.2 Pattern Prediction Algorithm

Figure 4.6: FCHR for 256 and 512 Byte Filter Cache

the incorrect filter cache hit-predictions will degrade the performance and incorrect filter cache miss-predictions will result in reduced access energy savings.

4.2.2.2 Filter cache Performance Index

The main motivation for incorporating a predictor is to improve overall performance of conventional hierarchical organization with the help of a predictor based implementation. Thus, we introduce another metric called Filter Cache Performance Index (FCPI), which is the ratio of correct filter cache predictions verses the total number of correct and incorrect predictions towards the filter cache. FCPI can be used to evaluate the performance improvements in the presence of filter cache, as incorrect predictions towards filter cache will lead to pipeline bubbles. An ideal predictor will have an FCPI of one.

Figure 4.7 compares the FCPI for pattern prediction and NFPT. Improvement in FCPI is noted when the pattern predictor is combined with the 256 Byte filter cache compared to NFPT. The same could be said for the combination of 512 Byte filter cache and pattern predictor.

4.2 Pattern Prediction Algorithm

Figure 4.7: FCPI for 256 and 512 Byte Filter Cache

4.2.2.3 Average Memory Access Cycles

Figure 4.8 compares the average memory access cycles or performance [39] which is the average access cycles incurred for instruction fetch. Here, the average memory access cycles is obtained by restricting our analysis only to filter cache and L1 cache accesses. This average memory access cycles will be one if an ideal predictor is employed. It can be seen that the combination of 256 Byte filter cache and pattern predictor results in better performance for all the benchmarks. The same could be said for the combination of 512 Byte filter cache and Pattern predictor. These observations reassure that the pattern predictor exhibits superiority over NFPT.

4.2.2.4 Energy-Delay Product

The impact on the incorrect predictions towards both the filter cache and L1 cache will affect the energy-delay measures. This is because incorrect predictions towards filter cache will incur additional access cycles while incorrect accesses to L1 cache will require additional access energy due to notable disparity between filter cache and L1 cache sizes. Figure 4.9 compares the normalized energy-delay product [39] for 256 Byte and 512 Byte filter cache with both the NFPT and the pattern predictor. As it

4.2 Pattern Prediction Algorithm

Figure 4.8: Average Memory Access Cycles

can be seen from Figure 4.9, the configuration with a 256 Byte filter cache and pattern predictor results in better normalized energy-delay product for most of the benchmarks. The same could be said for the combination of a 512 Byte filter cache and Pattern predictor compared to NFPT and a 512 Byte filter cache.

4.2.2.5 Optimal Filter Cache Size for Energy-Delay Product Reduction

Figure 4.10 shows the normalized energy-delay products of pattern predictor for various filter cache sizes. It is evident that the optimal energy-delay product for a given benchmark is governed by the characteristics of the application. This implies that the size of filter cache can be tailored for a given application. Our investigations show that, in general, a filter cache size of 512 Bytes results in an optimal energy-delay product for the benchmarks experimented. However, tuning filter cache for the application can result in energy savings of up to 47% compared to using a fixed 512 Byte filter cache. Thus, in chapter 5 we will present a methodology to obtain custom filter cache size for optimal energy reduction.

We have thus far seen that the pattern predictor results in an improved energy-delay product reduction compared to NFPT, with minimal complexity. The next sec-

4.3 Static Pattern Prediction Scheme

Figure 4.9: Normalized Energy-Delay Product for 256 and 512 Byte Filter Cache

tion presents a simplified version of the pattern predictor known as the static pattern predictor to further reduce the complexity and power consumption of the predictor.

4.3 Static Pattern Prediction Scheme

We have seen that the pattern predictor based on the two-level adaptive predictor is effective in predicting hit/miss in the filter cache. This dynamic predictor results in higher energy-delay product reduction compared to the NFPT scheme, with relatively low hardware complexity. However, in this dynamic scheme, the PHT must be updated for each instruction line access, thereby contributing to the power consumption of the predictive filter cache. This has motivated towards eliminating the saturation counters that require constant updating at run-time.

In this section, a static prediction scheme is proposed for predicting hits and misses in the filter cache. This static scheme is based on the two-level prediction scheme.

4.3 Static Pattern Prediction Scheme

Pattern Used	No of time Used	% of Time Hit Resulted	% of Time Miss Resulted	% time used
00000	11494170	6.0	94.0	43.2
00001	1162260	55.74	44.26	4.37
00010	499189	27.82	72.18	1.88
00011	862695	66.74	33.26	3.24
00100	410566	8.17	91.83	1.54
00101	164698	70.71	29.29	0.62
00110	273252	25.6	74.4	1.03
00111	594962	82.84	17.16	2.24
01000	613892	13.46	86.54	2.31
01001	40997	50.52	49.48	0.15
01010	19725	31.06	68.94	0.07
01011	110229	86.65	13.35	0.41
01100	451847	51.44	48.56	1.7
01101	97187	79.25	20.75	0.37
01110	144669	54.97	45.03	0.54
01111	519184	89.3	10.7	1.95
10000	1049494	20.99	79.01	3.94
10001	175790	29.22	70.78	0.66
10010	85725	25.53	74.47	0.32
10011	121975	82.7	17.3	0.46
10100	298780	16.83	83.17	1.12
10101	8382	72.88	27.12	0.03
10110	176871	90.58	9.42	0.66
10111	53808	79.54	20.46	0.2
11000	800178	30.44	69.56	3.01
11001	107744	83.92	16.08	0.4
11010	108102	51.21	48.79	0.41
11011	35680	93.31	6.69	0.13
11100	551857	18.23	81.77	2.07
11101	121994	65.79	34.21	0.46
11110	519260	17.19	82.81	1.95
11111	4931761	90.72	9.28	18.54

Table 4.1: Patterns Used in a Benchmark with 512 Byte Filter Cache

4.3 Static Pattern Prediction Scheme

Figure 4.10: Normalized Energy-Delay Product for 128Byte-2048Byte Filter Cache Sizes

4.3.1 Motivation for Static Pattern Prediction Scheme

The motivation for the static pattern prediction algorithm comes from the fact that a small filter cache takes advantage of temporal locality for tiny loops. If the loop is smaller than the filter cache, it will have continuous instruction line hits as complete temporal locality is possible (assuming there is no conflicts). On the other hand, it will have continuous misses when there is no temporal locality possible. When the loop is partially containable in the filter cache, patterns have to be maintained to predict. In a partially containable loop, the loop will be placed in one or more memory chunks. Since the loop is partially containable, there will be conflict within these memory chunks when placed in the filter cache. Thus, the hit/miss pattern will show a few hits followed by a few misses and so on. For example, if we look at the Figure 4.11, for the first loop the pattern shows a few hits followed by a few misses (or vice versa). For the second loop the pattern shows a few hits follows by a few misses and then a few hits (or vice versa) depending which blocks of the function and the main loop conflicts. That is, the hit/miss patterns will be ones followed by zeros and vice versa. Profiling application also validates this as shown in Table 4.1. Thus, in general, most of the loops

4.3 Static Pattern Prediction Scheme

Figure 4.11: Example for Memory Layout of Tiny Loops

will have similar hit/miss pattern while executing these loops. That is, if the previous instruction line accesses are hit in the filter cache, the probability of current one also being hit is high. In similar way, if the previous filter cache lines accesses are misses, the probability of current one being miss is also very high. We profiled the hit/miss patterns to verify this. Table 4.1 shows our profiling results for one benchmark. That is, for most frequently used SR values, the outcome of the prediction is heavily biased in one direction. Thus, the heuristic intuition here is that a Static Pattern Table (SPT) can be used as effectively as a dynamic PHT based predictor. Other benchmarks also exhibit similar behavior. Profiling for patterns validated this. Thus, based on the patterns exhibited by the benchmarks, a constant static prediction table given in Table 4.2 was created (for all the benchmarks).

4.3.2 Static Pattern Table (SPT)

Though, the static prediction table may not be optimal for all benchmarks, tailored solutions lead to only marginal gains. Hence, the Static Pattern Table (SPT) was fixed

4.3 Static Pattern Prediction Scheme

Pattern	Next Prediction	Pattern	Next Prediction
00000	0	01000	0
00001	1	01001	1
00010	0	01010	0
00011	1	01011	1
00100	0	01100	0
00101	1	01101	1
00110	0	01110	0
00111	1	01111	1
10000	0	11000	0
10001	0	11001	1
10010	0	11010	1
10011	1	11011	1
10100	0	11100	0
10101	1	11101	1
10110	1	11110	0
10111	1	11111	1

Table 4.2: Proposed Static Prediction Table

for all the benchmarks in order to maximize the benefits of this approach for realizing efficient hardware implementations. Figure 4.12 shows the algorithm used in the static filter cache hit/miss prediction. It is assumed that the address for the next instruction will be the current fetch address plus “4” due to an instruction being 32-bit wide. This would hold for most of the times based on the insights provided by branch prediction research. The predictor is accessed only when cache line changes are encountered. Thus, we predict a hit in the filter cache if the cache line being accessed is unchanged. In the event of a change, we access the SPT corresponding to the shift register value, and the prediction of a hit or a miss in the filter cache is made depending on the SPT value. In order to ensure that the prediction process does not interfere with the average cache access cycle, the prediction process must be hidden between accesses. The prediction for the next access is carried out in parallel with the current access. The simplified hardware implementation of the algorithm is shown in Figure 4.17. Our investigations also show that the SPT entries can be computed off line in order to capture any peculiar characteristics of an application, thereby assigning a unique SPT for each application. This will pave the way to further minimize energy-delay product

4.3 Static Pattern Prediction Scheme

in an application centric manner.

Figure 4.12: Static Pattern Prediction Algorithm

The following sections discuss the effectiveness of static pattern predictor with the filter cache size fixed at 256 and 512 Bytes. The prediction accuracy and the energy-delay product are compared to demonstrate the effectiveness of the predictor.

4.3 Static Pattern Prediction Scheme

Figure 4.13: Prediction Accuracy for 256 and 512 Byte filter cache

4.3.3 Prediction Accuracy

Figure 4.13 compares the prediction accuracy of pattern predictor, static pattern predictor and NFPT. It is clearly evident that the static pattern predictor performs better than the NFPT for most of the cases shown. However, this does not guarantee that the energy-delay product reduction for Static Pattern Predictor is less than that of NFPT. This is because the energy-delay product measures heavily depend on whether the wrong predictions are directed to main cache or filter cache, as discussed earlier. If the wrong predictions are directed towards filter cache, it will result in performance degradation. If, on the other hand, the wrong predictions are directed towards main cache it will result in increased access energy. The average memory access cycles and average access energy are discussed next to provide further clarifications.

4.3.4 Average Memory Access Cycles

Figure 4.14 compares the average memory access cycles of static pattern predictor with pattern predictor and NFPT. Here the normalization is done with respect to the case where filter cache is not present. As can be seen from the Figure 4.14, the static pattern predictor results in better prediction accuracy than the NFPT for most of the benchmarks. In addition, the prediction accuracy of the static pattern predictor is also comparable to pattern predictor, which uses dynamic PHT. The average performance

4.3 Static Pattern Prediction Scheme

Figure 4.14: Average Memory Access Cycles for 256 and 512 Byte Filter Cache

penalty for 256 Byte filter cache compared to the case where filter cache is not present is 3.29% for NFPT, 2.78% for Static Pattern Predictor and 2.47% for the pattern predictor. Similarly, for the corresponding figures for the 512 Byte filter cache are 1.76% for NFPT, 1.71% for static pattern predictor and 1.68% for the pattern predictor. Next, we will examine the access energy measures for both approaches.

4.3.5 Normalized Access Energy

Figure 4.15 shows the normalized access energy with respect to the base case where filter cache is not present. As before, the static pattern predictor outperforms NFPT for most of the benchmarks. The access energy reduction for 256 Byte filter cache compared to the case where filter cache is not present is 42.99% for NFPT, 45.07% for static pattern predictor and 45.40% for pattern predictor. Similarly, for the corresponding figures for the 512 Byte filter cache are 54.98% for NFPT, 56.32% for static pattern predictor and 56.54% for the pattern predictor.

4.4 Hardware Complexity Comparison

Figure 4.15: Normalized Access Energy for 256 and 512 Byte Filter Cache

4.3.6 Energy-Delay Product

Figure 4.16 shows the energy-delay product reduction for the static pattern predictor, pattern predictor and NFPT. It is clearly evident that, for the benchmarks, static pattern predictor provides for better energy-delay product reduction compared to NFPT. The energy-delay product reduction for 256 Byte filter cache compared to the case where filter cache is not present is 40.86% for NFPT, 43.30% for static pattern predictor and 43.85% for the pattern predictor. Similarly, for the corresponding figures for the 512 Byte filter cache are 54.14% for the NFPT, 55.51% for the static pattern predictor and 55.76% for the pattern predictor.

So far we have seen the pattern prediction and static pattern prediction algorithms as a hit/miss predictor for filter cache and its effect on the energy-delay product. In the next section, we compare the hardware complexity of these predictors with that of NFPT.

4.4 Hardware Complexity Comparison

This section compares the hardware costs of the NFPT, pattern predictor and static pattern predictor. We start off by comparing the hardware area cost for NFPT, pattern

4.4 Hardware Complexity Comparison

Figure 4.16: Normalized Energy-Delay Product

predictor and static pattern predictor, to study the complexities of the predictor. And, in the next subsection, we present the detailed energy measures for the pattern prediction and static pattern prediction logic, to show that the energy overhead of the pattern predictor is marginal.

4.4.1 Hardware Area Cost Comparison

Figure 4.17 shows the simplified implementation of the prediction algorithm shown in Figure 4.3. It employs an AND gate to detect whether the current fetch address and the predicted next fetch address (PC+4) maps to the same cache line. Based on this, the next fetch prediction source is selected. The register “last line” stores the last instruction line address accessed. While fetching a new instruction, the PC is compared with the “last line” to detect line change in the instruction cache. In the event an instruction line change is detected, the SR and PHT are updated. The architecture of the static pattern predictor is similar to that of the pattern predictor, except for the fact that the static pattern predictor uses 1-bit pattern table entry, and the dynamic updating of the pattern table entries is not needed here. NFPT implementation is explained in the pervious chapter. In the NFPT implementation, an NFPT table size is equal to $NCL \times 4$ bits, where NCL is the number of instruction cache lines. In addition, it

4.4 Hardware Complexity Comparison

Figure 4.17: Simplified Architecture of The Prediction Algorithm

Predictor	Area(In Gates)
NFPT-256 ¹ Byte	1560.6
NFPT-512 ¹ Byte	2798.3
NFPT-1024 ¹ Byte	5218.6
Pattern-8 Entry PHT	702.1
Pattern-32 Entry PHT	1802.1
Pattern-128 Entry PHT	6174.4

Table 4.3: Hardware Area Cost for Predictor Configurations without Filter Cache

consists of an additional register, to hold the last line, and a 4-bit comparator. Table 4.4 compares the main components for the NFPT, pattern predictor and static pattern prediction algorithm.

¹NFPT predictor is filter cache size dependent; hence its hardware complexity is being shown with respect to the filter cache size. The complexity (in gates) however, does not include the hardware cost of the filter cache itself.

4.4 Hardware Complexity Comparison

Predictor	For 256 Byte filter cache	For 512 Byte filter cache
NFPT	A register to hold last Line address, 4X16 bit Mem array to hold NFPT Table, 4-bit Comparator, and comparator for line change detection	A register to hold last Line address, 4X32 bit Mem array to hold NFPT Table, 4-bit Comparator, and comparator for line change detection
Pattern Predictor	A register to hold last Line address, 64 Bit Mem for PHT, 2-bit Comparator, and comparator for line change detection	A register to hold last Line address, 64 Bit Mem for PHT, 2-bit Comparator, and comparator for line change detection
Static Predictor	A register to hold last Line address, 32 Bit mem. for PT, and comparator for line change detection	A register to hold last Line address, 32 Bit mem. for PT, and comparator for line change detection

Table 4.4: Hardware Complexity Comparison of Predictors

As an initial estimate of relative hardware cost, the two predictors were synthesized using the Synopsys Behavioral Compiler with 0.35 micron Avanti Libra Passport library. The architectures were described using SystemC [66] and compiled to gates using the CoCentric SystemC Compiler [65]. Though the area measures produced by hardware synthesis from SystemC are not optimal, they serve to clearly illustrate the relative hardware cost of the two predictors. Table 4.3 shows the hardware resource requirements for implementing NFPT and Pattern predictor for various cache sizes and configurations. It is clear from the Table 4.3 that application-centric selection criteria for determining the predictor and PHT size may provide for an area efficient implementation of the pattern predictor. Unlike the NFPT, which is dependent on the filter cache size, the pattern predictor and static pattern predictor are application dependent. This distinct feature of the pattern predictor provides for a mechanism to propose application-centric configurations that could be tailored to satisfy area and energy-delay product trade-offs. The next sub section presents the detailed energy

4.4 Hardware Complexity Comparison

Figure 4.18: Single Cycle Pattern Prediction Architecture for Filter cache Access Prediction

measures for the pattern predictor.

4.4.2 Pattern Predictor

Until now we have seen that the pattern prediction algorithm is effective for predicting hits and misses in the instruction accesses in the filter cache. In this section, to obtain a meaningful energy measure, the design of the pattern predictor architecture was implemented in VHDL. Figure 4.18 depicts the single cycle implementation of the pattern predictor architecture. Here, as explained earlier, we access the predictor result (*sPred*) only when cache line changes are encountered. Hence, we predict a hit in the filter cache if the cache line being accessed is unchanged (*sPredEn* = 1). In the event of a change (*sPredEn* = 0), we access the PHT corresponding to the shift register value, and the prediction of a hit or a miss in the filter cache is made depending on whether

4.4 Hardware Complexity Comparison

Figure 4.19: Experimental Setup for Architecture Validation and Energy Measurement of the Prediction Logic

the pattern history table value reads above the pre-set threshold ($pCompareVal$) or below. For a two-bit saturating counter pattern history table, a threshold value of 2 (10_2) is used. When the outcome of the prediction is known the shift register and pattern history table values are accordingly updated, as in the normal branch predictor scheme. The algorithm is illustrated in Figure 4.3.

The address trace captured in the software simulation was then applied to the VHDL testbench of the architecture depicted in Figure 4.18. The overview of the pattern prediction architecture simulation and validation setup is shown in Figure 4.19. The architecture was synthesized for configurations supporting PHT's with 8, 32 and 128 entries. The synthesis was done on Synopsys Design Compiler with Avanti Passport Standard Cell Libraries for $0.35\mu m$ process technology, as in the SystemC implementation earlier.

Power characterization was done based on the switching activity collected during the simulation of the testbench on the various benchmarks of MediaBench. This provides for a reasonably accurate representation at this stage of design. The testbenches

4.4 Hardware Complexity Comparison

Predictor Configuration	VLSI Area Cost	Power (mW)
Pattern Predictor with 8 entry PHT	1360.0	34.70
SPP with 8 entry PT	463.7	2.22
Pattern Predictor with 32 entry PHT	2232.0	37.52
SPP with 32 entry PT	687.8	2.73
Pattern Predictor with 128 entry PHT	5581.45	41.35
SPP with 128 entry PT	1550.63	4.74

Table 4.5: Power Cost Comparisons for Pattern Predictor and Static Pattern Predictor Configurations

were run to completion and the switching activity was then back annotated to the gate level synthesized netlist to estimate the power requirement of the pattern predictor for the predictor configurations.

Table 4.5 compares the area and power costs for implementing the various pattern predictor and static pattern predictor configurations. As can be seen from the table, static pattern predictor consumes substantially low power compared to pattern predictor, with much reduced hardware complexity. Here, a cycle time of 2ns is used to obtain the power figures.

When the prediction architecture (32 entries PHT) cost for the pattern predictor logic is included, the average energy savings reduces by about 0.3% for the 256 and 1.6% for the 512 Byte filter cache. However, the energy-delay product reduction is still 14% and 5% higher than the hierarchical filter cache structure.

Further, for a 32 entry PHT, the synthesized design does not use up more than 2300 gates (0.14 mm²) on 0.35-micron technology and can operate in single cycle mode beyond speeds of 0.5 GHz. For simplicity, in the subsequent sections, unless explicitly stated, we present the energy figures with the pattern predictor cost included. From the Table 4.5, it can be easily appreciated that the energy cost for implementing the static pattern predictor is substantially lesser than the pattern predictor. From the table, it is also clear that the static pattern predictor and pattern predictor can be implemented with negligible hardware cost in an application centric manner. Thus, the pattern predictor and static pattern predictor let the designer tradeoff the energy-delay product and the hardware complexity.

4.5 Summary

Filter Cache Size	Conventional Filter cache		Pattern Predictor		Static Pattern Predictor		NFPT	
	APACD	PEDPR	APACD	PEDPR	APACD	PEDPR	APACD	PEDPR
256	14.14%	33.87%	2.47%	43.85%	2.78%	43.30%	3.29%	40.86%
512	7.61%	52.11%	1.68%	55.76%	1.71%	55.51%	1.76%	54.14%

Average Percentage Access Cycles Degradation (APACD), Percentage Energy-Delay Product Reduction (PEDPR)

Table 4.6: Overall Average Percentage Access Cycles Degradation (in %) and Energy-Delay Product Reduction (PEDPR) for Conventional Filler Cache, Pattern Predictor, Static Pattern Predictor and NFPT Predictor

4.5 Summary

In this chapter we presented two prediction algorithms to improve the energy-delay product in the predictive filter cache. It has been clearly demonstrated that the proposed pattern prediction based techniques are more energy efficient than the NFPT prediction schemes. The proposed pattern predictor and static pattern predictor result in better prediction accuracy and a notable reduction in the area and energy costs.

Table 4.6 summarizes the average memory access cycles penalty and energy-delay product reduction for conventional filter cache and predictive filter cache, with all the prediction algorithms. It is evident from Table 4.6 that the use of pattern predictor in filter cache hierarchy results in better or comparable energy-delay product reduction as compared to the NFPT. However, the architecture cost is directly proportional to the size of the filter cache with the NFPT based predictor. This imposes an immediate overhead with increased filter cache size. This is not the case with the pattern predictor, whose architecture complexity effectively remains constant for different filter cache sizes. The higher architecture complexity of the NFPT algorithm is evident from the comparison presented in Table 4.4. At this point it is important to note that the pattern predictor will maintain its supremacy over NFPT when architecture costs are included in the energy and delay analysis. In addition we also have presented the static pattern predictor that also provides for higher prediction accuracy than NFPT with much reduced hardware complexity.

Customization of filter cache hierarchy for an application is also efficient means

4.5 Summary

with which to tackle dynamic as well as static energy reduction, without impacting the performance. Next chapter presents a loop profiler directed methodology to rapidly customize the filter cache hierarchy.

Chapter 5

Profile Directed Tuning of Instruction Cache Hierarchy

5.1 Introduction

As we have seen earlier, low energy is one of the key design goals in current embedded systems design. With Deep Sub-Micron Process (DSM) technology, as smaller geometries exacerbate leakage, static power is also starting to dominate the power consumption equation in microprocessor designs. Customization of architecture to the needs of the application is one of the most effective ways to tackle dynamic as well as static energy reduction without impacting the performance much. This is especially advantageous in embedded systems which are often application specific. Architecture tuning aims to satisfy constraints such as size, performance, power and energy for a particular application. If the cache is larger than what is needed for the application, energy required to fetch from the larger cache may be unnecessarily high. Larger memory also dissipates higher leakage power. Conversely, if the cache is too small, excess energy may be wasted due to thrashing. The diversity among applications leads to very different cache requirements for different applications [85].

However, the major problem with the architecture tuning is that it is not feasible to perform exhaustive simulations for all possible combinations of cache parameters. Because of this multi-dimensional design space, the total number of possible designs can be exceedingly large. The design space will be even more when design space exploration needs to be done for the cache hierarchy involving multiple cache levels, requir-

5.2 Loop Profiling

ing higher design time. Typical cache parameters explored while evaluating the design space of the cache subsystems are the size, associativity, line size and cache write policy. In instruction caches, the parameter which dominates the power-performance metrics of the final system is the cache size [84]. Associativity and line size also allow the energy reduction to be tuned further but typically have a smaller impact. The goal of our work is therefore to identify the optimal filter cache and the L1 cache size for maximum energy reduction without compromising on the overall reduction in the energy-delay product.

We propose a loop profiling based methodology for selecting a near-optimal filter cache based instruction hierarchy, for a given application, without exhaustive simulations. The proposed methodology utilizes the loop profile data of the application to realize the near-optimal filter cache and L1 cache needed for a given application. The next section presents the methodology we used to extract the loop characteristics of the applications, before presenting our loop profiler directed methodology for tuning the instruction cache hierarchy.

5.2 Loop Profiling

The central challenge facing computer architects, compiler writers and application programmers is to understand the program's dynamic behavior so that the architecture and the program can be optimized for the efficient execution of the program. Events that occur when a program executes are often indefinable, but they provide a basis for understanding the behavior and improving its performance. This is particularly useful in embedded systems where the application which is going to be executed in the system is known in advance. Thus, the architecture can be tuned for the efficient execution of the application.

Previous research has focused on studying the loop characteristics of the applications, as this is the most critical region where an application spends most of its time. It was shown that in IBM System/370 applications more than half of the program execution time is spent in loops [41]. In recent literature, several techniques were proposed for dynamic loop detection for exposing more instruction level parallelism or thread

5.2 Loop Profiling

level parallelism [70]. Embedded software is generally thought to have a different behavior than desktop applications. The software tends to be written in a leaner manner and spends more time in tiny loops.

Loop profiling is often done in embedded systems for different purposes such as hardware software partitioning [64] and optimization in embedded systems. Compilers frequently use profile-directed optimizations to find the hot-paths within the applications to perform optimizations. Instruction set customization also depends on critical code segments for optimization. Thus, loop profiles for the embedded applications are readily available in most cases. When the loop statistics needed for the application is not available, the linked binary of the applications can be profiled to extract the loop characteristics of the application.

Several profiling tools have been developed to study the dynamic behavior of applications. Some tools like gprof [3] provide function level profiling and do not provide sufficient information. Tools such as ATOM [59] and SpixTools [22] are specific to a particular microprocessor family and these tools do not support ARM ISA. LOOAN [72] is another profiler developed for loop profiling in embedded applications. However, LOOAN software is not freely available. Thus, to better understand the loop behavior of the embedded applications we developed our own loop profiler software. It should be noted here that our intention is not to propose a new efficient methodology for loop profiling; in contrast we want to develop a simple loop profiling tool to facilitate in understanding the loop characteristics for a given application.

A loop profiler for the extraction of the loop profile for the executable can be implemented either by instrumenting the binary for extracting the information or by extracting the information while executing the application in an instruction set simulator. In the second method, though slower, the binary is executed unmodified, leading to more accurate results. Thus, our loop profiler is implemented using the second method. Our implementation, when coupled with SimpleScalar [16] instruction set simulator generates the loop profile of a given ARM program.

When the application is being executed in the simulator, the loop profiler constructs the Control Flow Graph (CFG) for the whole program, which is equivalent to a call graph plus the CFG for each function. Simultaneously the edge profile for the application is also constructed. Once we have the edge profile, we strip the nodes with edge counts less than a minimum value from the CFG, to separate the hot-paths. This

5.2 Loop Profiling

Figure 5.1: Basic Operations of Loop Profiler

is done so that the number of entries in the CFG is minimized. This is important for the runtime of the loop profiler as we are dealing with the whole program. Figure 5.4 shows the striped CFG for an application “grep”. This visual representation of the directed graph is generated with the tool DOT [50], which transforms a graph denoted in graphical description language into visual format.

Once we get the stripped CFG with the edge profile, we create a precedence list for each node. Algorithm 2 constructs a precedence nodes list for each node, which will be used to check whether there is a path, and Algorithm 1 then traverses the CFG from the loop head to find all the nodes which constitute the loop. Since we are dealing with the embedded applications and we are dealing just with hot paths, we can calculate it relatively fast.

We can detect the loops in the control flow graph by looking at the back edges that are not a “function call” or “function return”. Then, from the loop head we can traverse the graph and find the basic blocks that are part of the loop. All the successor nodes are

5.2 Loop Profiling

Algorithm 1 Computing The List of Basic Blocks for Each Loop

```

1: n,p: Node
2: N : list of all frequent nodes
3: for each loop (detected by loop backedge) do
4:   n:=loop_head
5:   while 1 do
6:     add n to the list of node for this loop
7:     for each  $p \in \text{Succ}(n)$  do
8:       if (p Precedes loop_tail) AND  $p \neq \text{loop\_tail}$  then
9:         push(p)
10:      Add p to the loop_list (if already not there)
11:     end if
12:   end for
13:   n=pop()
14:   if n=empty then
15:     break
16:   end if
17: end while
18: end for

```

Figure 5.2: Basic Blocks in a Loop

5.2 Loop Profiling

Algorithm 2 Algorithm for Creating Precedence List (Used to Detect if There is a Path between Two Basic Blocks)

```

1: D,T : in set of Node
2: n,p : Node
3: N : list of all frequent nodes
4: change := True : boolean
5: Precede : Node  $\rightarrow$  set of Node
6: for each  $n \in$  of  $N$  do
7:   Precede(n) := {}
8: end for
9: repeat
10:   change := false
11:   for each  $p \in$  of  $Pred(n)$  do
12:      $T \cup = Precede(p)$ 
13:   end for  $D := \{n\} \cup T$ 
14:   if  $D \neq Precede(n)$  then
15:     change := TRUE
16:     Precede(n) := D
17:   end if
18: until !change

```

checked for a path between the successor nodes and the loop tail. If a path is present then it is part of the loop, otherwise it is not. That is, if there is a path from loop head to a node and if there is path from node to the loop tail then the node is part of the loop. Algorithm 1 explains how the nodes which constitute the loop can be found. It should be noted that, for cache size studies, we are interested in all the loops (strongly connected graphs) as against natural loops which are detected based on dominator tree [49].

For example, if we look at Figure 5.2, there is no path between basic block 5 and loop tail (basic block 4). Hence, the basic block 5 is not part of this loop. Therefore, the basic block list for the loop will be 1, 2, 3 and 4. Once we get the basic block list corresponding to the loop, we can extract the other information such as static instruction count of the loop, dynamic instruction count of the loop, total execution time of

5.2 Loop Profiling

Figure 5.3: Computing the Required Cache Size with Basic Blocks (Each Basic Block is Shown with Start and End Addresses)

the loop, cache size required to contain this loop etc. from the list, as shown in Figure 5.3.

When the loop contains procedures, the software also maintains data structure to calculate the average execution statistics for the procedure, as the procedure can be part of multiple loops.

In instruction cache hierarchy the filter cache is used so that energy savings can be realized by caching the tiny loop in an energy efficient filter cache. And the L1 cache contains the working set of the application so that the power and performance expensive main memory access can be minimized. That is, the filter cache and L1 cache are used to realize different purposes. And, the filter cache can be tuned in isolation from the L1 cache. The next section presents the proposed loop profiler directed methodology for tuning the filter cache and the subsequent section presents the L1 cache tuning methodology.

5.2 Loop Profiling

Figure 5.4: An Example of a CFG with Predefined Minimum Edge Count (Minimum Edge Count of 1000 for the Application “grep”)

5.3 Optimizing Filter Cache Size

5.3 Optimizing Filter Cache Size

The objective in this thesis is to use filter cache hierarchy in order to minimize the energy-delay product in the instruction cache hierarchy, while limiting the performance penalty. As studied earlier, customization of the filter cache architecture also results in minimizing the energy and thereby reduces the energy delay product. Thus, in this section we present a methodology to customize the filter cache.

The energy savings in the filter cache hierarchy comes from the frequent tiny loops that are cached in the filter cache till the next iteration. While executing larger loops, or while executing the instruction stream that does not have immediate temporal locality, filter cache acts as a line buffer. When the filter cache acts as a line buffer, the size of the filter cache does not matter, as a single line itself is sufficient to exploit spatial locality. On the other hand, the per-access energy of the filter cache increases gradually when the filter cache size is increased.

Let us calculate the energy savings (ES) resulting with the spatial locality (Equation 5.1). Assuming all the instructions in a cache line (16 byte for filter cache) execute, 75% of the instruction accesses will be hits in the filter cache, when only spatial locality is possible. Though, not all the instructions in a cache line execute all the time, since we are only interested in relative savings, the error due to this will be applicable for all the filter cache sizes. Filter cache also adds a slight energy overhead in-terms of read/write accesses to the filter cache for each new cache line, which is subtracted as shown in Equation 5.1. Since the spatial locality will be possible throughout the execution of the application, it is being multiplied by 1.0 (100% of the time). Here, E_{L1} and E_{FC} stand for the access energy for the L1 cache and the filter cache respectively.

$$ES = ((0.75) \times E_{L1} - E_{FC}) \times (1.0) \quad (5.1)$$

Where ES stands for the energy savings due to spatial locality.

Let us calculate the energy savings (ET) resulting with the temporal locality (Equation 5.2). The basic requirement of filter cache for energy savings is to completely or partially cache small but frequent loops while they are executed. Thus, the filter cache size should be large enough to cache these frequent loops completely. If the size of the filter cache is increased, more and more frequent loops can be completely cached

5.3 Optimizing Filter Cache Size

in the filter cache, resulting in reduced L1 cache accesses, which in-turn will reduce the overall cache access energy. However, increasing the filter cache size increases the per-access energy and the static energy (due to the larger memory size). Increasing the filter cache beyond a limit would increase the filter cache access energy and negate the advantage possible with the filter cache. Figure 5.5 shows how increasing the cache size changes the access energy for a single loop.

$$ET = \begin{cases} = 0 & \text{if } loop > 2 \times FC \\ = (0.25) \times E_{L1} \times T_{ex} & \text{if } loop \leq FC \\ = (0.25) \times (1 - (\frac{2 \times FC - loop}{loop})) \times E_{L1} \times T_{ex} & \text{if } (loop < 2 \times FC) \&\& (loop > FC) \end{cases} \quad (5.2)$$

Where ET stands for the Energy savings due to spatial locality for a given loop and T_{ex} stands for the fraction of the total execution time spent in this loop, which accounts for the nested loop effect which we have already discussed.

Therefore, we have chosen a set of possible filter cache sizes that have the potential to provide energy savings. In our experiments, we use filter cache sizes of 128 byte, 256 byte, 512 byte, 1024 byte and 2048 byte. Thus, for selecting the appropriate filter cache, we need to calculate the relative energy savings possible for the given application with each filter cache and select the filter cache size with maximum savings. When we have the loop profile, which constitute the loop size and the percentage of the total execution time being spent on these loops, we calculate the relative energy savings for each loop and then normalize it with the percentage execution time for this loop.

For the loop size, we need to use the cache size needed to contain the loop. This is because, the loop might not be placed in the continuous memory location and the basic blocks in the loop have to be aligned to the instruction cache lines. This can be very easily calculated as we have the list of basic blocks that constitutes this loop. Since the line size of our filter cache is 16 bytes, we align the instruction blocks in the loops to 16 byte to calculate the filter cache size needed to cache all of them. For example, if we have a loop shown as in Figure 5.3, we will need a slightly larger filter cache to completely cache the loop. We cache only the basic blocks that are executed at least 20% of the loop iterations. That is, we accommodate only the basic blocks that are

5.3 Optimizing Filter Cache Size

Figure 5.5: Access Energy vs. Cache Size for a Single Loop

executed frequently. In our subsequent discussions in this section, whenever we refer to loop size, we will be referring to the filter cache size needed to cache the loop (with the frequent basic blocks) in the direct mapped filter cache.

We also differentiate the nested loops in energy savings calculations. That is, in the nested loop, though the outer loop executes less frequently, it might have higher execution time (T_{ex}) because of the inner loops. For instance, if we look at Figure 5.6, the outer loop executes less frequently and it is not necessary to increase the filter cache size to accommodate the loop, though the execution time of this loop is high. Thus, when we calculate the execution frequency for outer loops, if the dynamic instruction count (average number of instructions executed in a loop iteration) of the loop is higher than the static instruction count (number of instruction in the loop) of the loop then we multiply the execution time by $\frac{\text{Static.instruction.count}}{\text{Dynamic.instruction.count}}$ of the loop to calculate the effective execution time.

Once we have the loop statistics, we can calculate the energy savings possible for each loop due to the temporal locality. Equation 5.2 shows how the energy savings due to temporal locality (ET) can be calculated for a loop. If the loop size is more than $2 \times \text{FC}$ (where FC stands for filter cache size), then there will not be any temporal

5.3 Optimizing Filter Cache Size

Figure 5.6: Computing the Effective Execution Time of an Outer Loop in Nested Loops

locality possible with this filter cache. This is assuming there is no conflict misses between instruction lines that constitute this loop. Since we are looking into tiny loops that are being cached in the small filter cache, this assumption generally holds true. If the loop size is less than that of the FC, a complete loop can be cached in the filter cache, and then full energy savings will be extracted with the filter cache. On the other hand, if the loop size is between $2 \times \text{FC}$ and FC , partial energy savings will be extracted from the filter cache.

The total relative energy savings for a given filter cache over a conventional filter cache is the addition of ET and ES. Algorithm 3 calculates the energy savings with the given filter cache. Once we calculate the relative energy savings with the different filter cache sizes, the one with the maximal energy savings is selected as the suitable filter cache for the given application. The subsequent section produces the results obtained with the proposed analytical model.

5.3 Optimizing Filter Cache Size

Figure 5.7: Comparing Relative Access Energy with Calculated and Simulated Values for Filter Cache

5.3 Optimizing Filter Cache Size

Algorithm 3 Calculate the energy savings with the filter cache of size FC

```

1:  $savings = (0.75) \times E_{L1} - E_{FC}$ 
2: for  $loop = CriticalLoop_1$  to  $CriticalLoop_N$  do
3:   if  $loop > 2 \times FC$  then
4:      $savings += 0$ 
5:   else
6:     if  $loop \leq FC$  then
7:        $savings += (0.25) \times E_{L1} \times T_{ex}$ 
8:     else
9:       if  $loop < 2 \times FC$  then
10:         $savings += (0.25) \times (1 - (\frac{2 \times FC - loop}{loop})) \times E_{L1} \times T_{ex}$ 
11:      end if
12:    end if
13:  end if
14: end for

```

Figure 5.8: Energy Reduction with Optimal Filter Cache vs. Calculated Filter Cache for MediaBench Applications

5.3 Optimizing Filter Cache Size

Benchmark	Optimal filter cache Size	Calculated filter cache Size
ADPCM-DEC	1024	1024
ADPCM-ENC	1024	1024
EPIC	256	256
UNEPIC	512	512
G721.DEC	512	512
G721.ENC	512	512
GSM.TOAST	2048	2048
GSM.UNTOAST	1024	1024
MIPMAP	128	128
OSDEMO	1024	1024
TEXGEN	1024	1024
MPEG2.DEC	512	512
MPEG2.ENC	1024	1024
PEGWIT	512	256
PEGWIT.DEC	512	256
PEGWIT.ENC	512	256
JPEG.DEC	1024	1024
JPEG.ENC	512	512

Table 5.1: Optimal Filter Cache Size for Energy Reduction

5.3.1 Performance Comparison

Figure 5.7 compares the relative access energy obtained with the analytical model to that obtained with the exhaustive simulation, for a few benchmarks. The analytical model, which was described earlier, produces the average per access energy savings. Thus, it is subtracted from the per access L1 cache energy, to obtain the average per access energy with the filter cache. Both the calculated and simulated values are normalized such that the minimum access energy is one. As can be seen from Figure 5.7, the results obtained and the analytical model match closely.

Table 5.1 compares the optimal filter cache size for energy reduction obtained by the experimental method with that from the analytical model. Figure 5.8 compares the normalized energy for optimal filter cache configuration calculated as compared to using a fixed filter cache configuration of 512 bytes for the complete applications. As it can be seen from Table 5.1, analytical model produces the best filter cache con-

5.4 Optimizing L1 Cache Size

figuration for most of the benchmarks. That is, the analytical model finds the optimal configuration except for PEGWIT, PEGWIT.DEC and PEGWIT.ENC. Even for the cases where it does not produce the correct result, the difference in the energy reduction with the optimal filter cache and the next best filter cache is extremely small. Overall, tuning the filter cache for the given application results in up to 47% energy reduction as compared to using a fixed filter cache size of 512 Byte. The average energy savings for the applications simulated is 5.74% compared to fixed filter cache size of 512 Byte. The next section presents the methodology we used for the L1 cache size optimization.

5.4 Optimizing L1 Cache Size

5.4.1 L1 Cache Size Optimization Methodology

Cache memories work based on the principles of locality. If we look at the instruction accesses, the instructions are accessed consecutively till a control flow instruction (a taken branch instruction) is reached. Thus, the instruction access exhibits high spatial locality. The temporal locality in instruction access comes either from the loops in the instruction stream or, from the function (subroutine) calls to the same routine from different places. Thus, any instruction which is not part of any loop or function call need not be cached once all the instructions in the line are executed, as only spatial locality is possible. Any instruction that is part of loop or function call should be in the cache till they are needed for exploiting temporal locality.

For example, in the Figure 5.9, while executing the basic block 2, we no longer need basic block 1. This is because basic block 1 is not part of any loop. However, basic block 2 must be cached till the basic block 11 is reached. That is, if a basic block is part of a loop, it must be cached till the loop is exited. If a basic block is part of multiple loops, it must be cached till all the loops are exited. In the same way, if a basic block is part of a sub-routine it must be cached till the sub-routine is called for the last time. Our analysis with the embedded applications shows that subroutines that are not part of loops are often executed less frequently. Such sub-routines are called to handle unexpected conditions. And, the subroutines that are part of loops are executed frequently.

5.4 Optimizing L1 Cache Size

Figure 5.9: CFG of a Typical Application

However, in reality, we never design a system that has zero misses, excluding the misses resulting from the first time access, as this implies a huge cache. Thus, we cannot cache all the basic blocks in the loop. Instead, we need to cache the basic blocks in the hot-paths (the basic blocks in the path which are executed frequently) within the loop. For example if we look at the Figure 5.9, in the outer loop, the basic block 3 is executed only twice. Caching the non-frequent basic blocks will increase the cache size and access energy compared to the savings obtained with decreasing the main memory access energy. Thus, we need to use a threshold value to filter out the basic blocks in a loop that are not executed frequently. In our experiments we use a threshold value of $0.5 \times \frac{\text{Dynamic_instruction_count}}{\text{Static_instruction_count}}$. This is based on the observation that on an average, each instruction executes for $\frac{\text{Dynamic_instruction_count}}{\text{Static_instruction_count}}$ times, thus, if an instruction executes less than the average number of times, we need not cache it. In our experiments, we use a threshold that is 50% of the average execution of an instruction. Some instructions in the hot path execute more frequently than the average, and some instructions in the non-frequent path execute much less frequently than the average

5.4 Optimizing L1 Cache Size

value. Thus, in general, changing the threshold in the range of the average value does not affect the filtering of frequent basic blocks. So we use 50% of the average. We also set a minimum limit of 10 for the threshold.

$$Threshold = \begin{cases} 10 & \text{if } 0.5 \times \frac{Dyn_inst_count}{Static_inst_count} < 10 \\ 0.5 \times \frac{Dynamic_inst_count}{Static_inst_count} & \text{Otherwise} \end{cases} \quad (5.3)$$

Our goal is to find the near-optimal cache which will minimize the energy consumed in the L1 cache and the external bus access. Therefore, to calculate the cache size needed for a given application, we have to calculate the relative access energy for each possible L1 cache size and select the L1 cache with the minimal access energy. We use a cost model for calculating the energy for a given L1_size with a particular loop. That is, for a given loop_size, the L1 cache will have different energy cost depending on whether the loop is fully containable, partially containable or not containable at all. Here, we assume that L1 cache miss takes 50 times more energy for cache refill as compared to the L1 cache hit as shown in the literature [85].

Assuming that there are no conflict misses, as our cache is highly associative, an L1 cache of L1_size and loop of loop_size, the Cost (loop_size) can be calculated as follows:

When the complete loop is cache resident : all the instructions will be hit in the L1 cache as temporal locality and spatial locality are possible:

$$C1 = 1.0 \times E_{L1} \times T_{ex} \quad (5.4)$$

Here, T_{ex} stands for the execution spent in this particular loop as a fraction of the total execution time. This includes the effect of nested loops as discussed earlier. Since the complete loop is cached, it is multiplied by 1.0.

When the partial loop is cache resident : When loop_size is greater than the L1_size and less than $2 \times L1_size$, then the portion of the loop which can not be contained (NCR or Non Cache Resident part) is:

$$NCR = \frac{2 \times L1_size - loop_size}{loop_size} \quad (5.5)$$

5.4 Optimizing L1 Cache Size

Temporal locality is possible for the part which can be contained in the cache (1 - NCR times). For the rest (NCR), only spatial locality is possible. And also, off-chip accesses have to be made with 50 times more energy. Since the L1 cache has a 32byte line size and the instruction width is 4bytes, even when temporal locality is not possible, a maximum of seven out of eight instructions will hit in the L1 cache which is equal to 87.5%. That is for the part which can not be contained we still can have a maximum of 87.5% of the instruction accesses being a hit in the L1 cache.

$$C2 = (1 - NCR) \times E_{L1} \times T_{ex} + (0.875 \times NCR + 0.125 \times 50 \times NCR) \times E_{L1} \times T_{ex} \quad (5.6)$$

When no temporal locality is possible : If the loop_size is greater than $2 \times L1_size$, there will not be any temporal locality possible. Since the complete loop is not cached, only spatial locality is possible so a maximum of 87.5% of the instructions will be hit in the L1 cache and 12.5% of the time the main memory access has to be made with 50 times more energy.

$$C3 = (0.875 \times 1.0 + 0.125 \times 50 \times 1.0) \times E_{L1} \times T_{ex} \quad (5.7)$$

Therefore, the cost for a cache size with a given loop is:

$$Cost(loop_size) = \begin{cases} C1 & \text{if}(loop_size < L1_size) \\ C2 & \text{if}(loop_size > L1_size) \\ & \text{and } loop_size < 2 \times L1_size \\ C3 & \text{if}(loop_size > 2 \times L1_size) \end{cases} \quad (5.8)$$

With the loop profile and cost model, we can calculate the cost for each L1 cache size and then select the L1 cache with the lowest cost as the near-optimal L1 cache size. When we calculate the L1 cache size based on the loop statistics, we are essentially calculating the L1 cache size needed such that the capacity misses are reduced. In the next section we will provide the results for the L1 cache size optimization.

5.4.2 Performance Comparison

Figure 5.11 compares the relative access energy obtained with the analytical model to that obtained with the simulation, for a few benchmarks. Both the calculated and

5.4 Optimizing L1 Cache Size

Figure 5.10: Energy Reduction with Optimal vs. Calculated L1 Cache Sizes for MediaBench Applications

simulated values are normalized such that the minimum access energy is one. As can be seen from Figure 5.11, the results obtained matches the analytical model quite closely. However, in developing these models, we assumed that there are no conflicts between the instruction lines in the loop. We also select the basic blocks from the hot-paths to keep in the cache. For doing this, we use an approximate threshold based filtering technique. As a result, for some cases, the model does not follow the actual values closely for smaller L1 cache sizes, as it is evident from JPEG.DEC. As a result, when the difference between the optimal and next one is minimum, the model might find the next best configuration.

Table 5.2 compares the optimal cache size which is obtained by simulation with that obtained with the loop statistics. Here, as explained earlier, the optimal L1 cache sizes are calculated with the 32byte line size and 8-way associativity with various cache sizes. For calculating the energy consumption in the L1 cache and off-chip bus access, we used the method used in [85]. And also, since our focus in this section is L1 cache, we have excluded the filter cache altogether in the calculation.

Figure 5.10 compares the energy savings obtained with the optimal L1 cache size with that from the analytical model. As can be seen from the Figure 5.10, the analytical

5.4 Optimizing L1 Cache Size

Figure 5.11: Comparing Relative Access Energy with Calculated and Simulated Values for L1 Cache

5.4 Optimizing L1 Cache Size

Benchmark	Calculated cache size	optimal cache size
adpcm.enc	1	1
adpcm.dec	1	1
gsm.untoast	4	1
gsm.toast	4	4
mpeg2.enc	2	4
mpeg2.dec	4	8
jpeg.dec	4	8
jpeg.enc	8	4
mipmap	2	1
osdemo	16	16
texgen	16	16
pegwit	8	8
pegwit.dec	8	8
pegwit.enc	16	16
epic	1	1
unepic	1	2
g721.enc	8	8
g721.dec	8	8

Table 5.2: Comparison of Optimal vs. Calculated L1 Cache Sizes for MediaBench Applications (in KB)

model settles for a cache size which results in energy reduction close to the optimal configuration. Overall, tuning the cache size with the proposed methodology results in up to 18% energy reduction as compared to using a fixed cache size of 32KB. The average energy gain for the applications simulated is 12% as compared to the optimal value of 13%.

Although, the values calculated with the loop statistics are not always optimal, the energy savings are very close to those for the optimal case. Therefore, this along with the configurable cache architecture (discussed earlier) can be easily applied to get energy reduction in the instruction cache hierarchy. Hence, a methodology directed by simple loop statistics can be effectively used to deduce the cache size in an application

5.5 Summary

Figure 5.12: Leakage Power (W) for Various L1 Cache Sizes

specific manner.

It should be noted here that we do not include the leakage energy in the cost model we presented in this section. This is because, it is understood that using a custom L1 cache as against using a universal big cache will result in reduction in the leakage energy. As can be seen from the Figure 5.12, the leakage power is proportional to the L1 cache. Therefore, having a smaller L1 cache will reduce the leakage power. To keep the model simple in the L1 cache calculations, we just focus on the dynamic energy consumed because of the cache access and off-chip access. However, it is clear that the leakage energy also will be saved as compared to using a fixed L1 cache, which is large enough to contain the working set of all the applications.

5.5 Summary

Profiling applications is essential to optimize the applications for improving the performance, particularly for embedded systems. Although, loop profiling is often used in embedded systems for hardware software partitioning and feedback directed compiler optimization, there is no readily available loop profiler for extraction of the loop statistics needed for cache hierarchy optimizations. Thus, we presented a loop profiling methodology for studying the applications. In addition, we have presented novel loop profile directed methodologies for tuning the filter cache size and the L1 cache size.

5.5 Summary

The proposed methodologies have been shown to produce the near-optimal instruction filter cache size and L1 cache size without time consuming exhaustive simulations. In the next chapter we will present a dynamic filter cache management scheme to further reduce the energy-delay product of a filter cache hierarchy that consists of optimized filter cache and L1 cache sizes.

Chapter 6

Dynamic Filter Cache Management Scheme

6.1 Introduction

Although we have established techniques to optimize the size of the filter cache and the L1 cache, there will be instances when the filter cache may not be beneficial due to run time characteristics of the applications. The filter cache depends on the temporal locality in loops which can be cached till their next iteration. If the size of the loop is larger than what can be accommodated in the filter cache, it will not have any real benefit other than acting as a line buffer. A suitable dynamic management scheme has therefore been devised to further reduce the energy-delay product. L1 cache on the other hand does not benefit from this approach. Hence, this enhancement is not extended to the L1 cache.

6.2 The Rationale for Dynamic Management of Filter Cache

Branch prediction confidence based enabling/disabling of the filter cache is studied in [11]. A branch that is predicted taken with high confidence is expected to be taken during the particular phase of the program. If it is not taken, it is considered to be behaving unusually. If the program behaves unusually, it will probably drive the thread

6.2 The Rationale for Dynamic Management of Filter Cache

Figure 6.1: Simple Architecture for Dynamic Filter Cache Management

of control to a portion of the code that is not very frequently executed. And, the portion of the program which is executed need not be cached in the filter cache. Only the basic blocks that are executed frequently will be accommodated in the filter cache. The most frequent basic blocks such as those from loops will be cached in the filter cache. Since this scheme relies on sophisticated branch prediction schemes for enabling and disabling the filter cache, it cannot be used in embedded processors, where these sophisticated branch prediction schemes are often not used. In addition, it has been shown that the performance of this scheme is inferior to the predictive filter cache [68].

In this report we are interested in using dynamic filter cache management in single issue embedded processors. Hence, we use a filter cache management scheme based on loop detection. The scheme should be able to detect loops that cannot be contained fully or partially in the filter cache. The logic also should decide whether the loop has been exited. Since our interest is on embedded processors, this scheme should be able to detect loops with minimal hardware complexity. When the filter cache is disabled, we also need to have a line buffer to exploit the spatial locality.

6.2 The Rationale for Dynamic Management of Filter Cache

Figure 6.1 shows the simplified dynamic filter cache management architecture, where the loop detection is used to enable and disable the filter cache. When the filter cache is enabled, the hit/miss predictor predicts whether the instruction should be fetched from filter cache or the L1 cache. This is done so that the filter cache can exploit energy savings from partially containable loops. In addition, we also use a line buffer to exploit spatial locality when the filter cache is disabled. Thus, at any point of time, either the filter cache or the line buffer will be active. Now we will discuss each of these components and their operation briefly in the subsequent section.

6.2.1 Line Buffer

We use a line buffer with a 16 Byte line size (equivalent to the filter cache line size) for exploiting the spatial locality when the filter cache is disabled. Line buffers have been researched quite extensively for energy reduction in the cache hierarchy. Su and Despain proposed in-cache two-level hierarchies [61] in which a single line buffer, serves as the first level cache. This design is essentially a single-entry filter cache within the original cache. Consequently, a line buffer miss requires additional cycles to access the main cache, degrading the overall program performance. Moreover, the overall energy consumption may increase if the line buffer miss rate is high.

Ghose and Kamble [25] introduced a concurrent version of line buffers for pipelined caches in which cache lines in the same set are organized in one Wider Line Buffer (WLB), and they keep multiple such buffers. Buffering multiple WLBs aims at improving the WLB hit rate. The proposed WLB is effectively a fully associative cache placed on the side of the level one cache. Both caches are inquired simultaneously, and a hit in WLB cancels the ongoing access in the main cache assuming that the former is resolved earlier. The major overhead of this design comes from (1) the power dissipated in the (fully-associative) WLB on buffer misses, including miss detection and the line replacement, (2) the power spent in main cache activation during WLB hits, and (3) the comparisons of both set index and tag in the WLB on each potential hit.

Our focus is on embedded processors where L1 cache is typically accessed in a single cycle. In addition, our focus is on the instruction cache. Thus we employ a simple predictive line buffer for exploiting temporal locality when filter cache is disabled. The line buffer we use works as follows: when the filter cache is disabled,

6.2 The Rationale for Dynamic Management of Filter Cache

Figure 6.2: Line Buffer Enable Signal Generation with Previous Address Bits A_3 And A_2

if the PC and PC+4 (here each instruction is 32-bits wide and so that next instruction address is the current instruction address plus four bytes) map to the same line, then the line buffer is used for the next access. Otherwise, L1 cache will be accessed and the same line will be transferred to the line buffer. In addition to this, when in the decode stage of the processor pipeline, if a branch instruction is detected, we always bypass the line buffer and fetch the instruction from the L1 cache. This is done irrespective of whether the branch is taken or not taken. Thus, the processor core has to signal it to the line buffer. This simple scheme, as explained in Figure 6.2, results in a line buffer where the instruction accessed is always present.

6.2.2 Filter Cache Enable/Disable Mechanism

For the dynamic filter cache scheme to be effective, the dynamic filter cache controller should detect loops in the program execution dynamically (which can be fully or partially contained in the filter cache till the next iteration). It is imperative that the algorithm for detecting the loop be simple and energy efficient so as to present minimal overhead to the architecture. Therefore, we devised a very simple backward branch based loop prediction algorithm. That is, loops in dynamic instruction access could

6.2 The Rationale for Dynamic Management of Filter Cache

Figure 6.3: Simple Loop Detection Based Filter Cache Enable/Disable FSM for Dynamic Filter Cache Management

6.2 The Rationale for Dynamic Management of Filter Cache

be determined by a backward branch. In general, a backward branch occurs due to an iteration of a loop. Though this might not always be true, it keeps the algorithm simple and VLSI efficient. The loops which are miss-predicted are small in number, and it is envisaged that they shall not accumulate to a high energy/performance penalty. Hence, while executing the instruction stream, any backward branch instruction would enable the filter cache.

The filter cache enable/disable logic consists primarily of the FSM shown in Figure 6.3. Once the filter cache is enabled, the value $\frac{N}{2}$ (the number consecutive filter cache line misses that will disable the filter cache) is loaded into the counter as shown in the FSM of Figure 6.3. The FSM will then track the hits and misses of the instruction lines. It will disable the filter cache when continuous instruction line accesses results in misses. The number of stages (N) through which the FSM will go before the filter cache gets disabled is kept variable depending on the application and its requirements. For obtaining maximal benefit with the largest possible cacheable loop, N should be large. On the other hand, if N is very large, it will take time for the filter cache to get disabled after it comes out of the loop. Hence, the value of N needs to be decided based on the application's requirement. We term the value N , the miss sensitivity factor, as it controls utilization of the filter cache thereby controlling the energy-delay measures, especially its performance.

For instance, if we consider a filter cache with 16 lines, the N value should be at least 15 to exploit the temporal locality with a loop which will have one instruction line hit in the filter cache. However, if the value of N is large, there will be large number of misses in the filter cache before it gets disabled for smaller loops. This results in performance penalty and increased energy consumption. However, earlier research on loops as well as our profiling of the embedded applications shows that the majority of the frequent loops in embedded applications consist of tiny loops. As shown earlier, in embedded applications, nearly all the time spent in loops is spent in loops of 256 instructions or less. In addition 77% of the execution time of the application is spent in loops with 32 instruction or less [72]. Thus, a small value of N should suffice to obtain improved energy-delay as shall be seen later in the results.

The FSM shown in Figure 6.3 can be easily implemented in hardware as a counter. The counter will count down (count up) when it encounters a miss (hit). When the

6.3 Energy-Delay Tradeoff

counter value becomes zero, the filter cache is disabled. A backward taken branch will enable the filter cache. This will also load the value $\frac{N}{2}$ into the counter.

6.2.3 Filter Cache Hit/Miss Predictor

The dynamic filter cache scheme uses *pattern predictor* as the hit/miss predictor as before. The proposed dynamic filter cache scheme works concurrently with the filter cache hit/miss predictor. The predictor predicts whether a particular instruction access is going to be a hit (or a miss) in the filter cache (when the filter cache is enabled). Therefore, unlike the loop cache, the proposed scheme utilizes the filter cache even when only a part of the loop can be contained in the filter cache. The predictor directs the fetch unit to either filter cache or L1 cache based on the hit/miss predictions in the filter cache. In contrast, the loop cache caches instructions only if the complete loop can be cached. Hence, the proposed scheme leads to a solution superior to the loop cache. The subsequent sections compare the energy and delay measures of the proposed dynamic filter cache scheme with the pattern predictor.

As explained in Algorithm 4, the filter cache is used if temporal locality is possible in the filter cache. Otherwise, the filter cache is disabled and line buffer is used to exploit spatial locality along with the L1 cache.

6.3 Energy-Delay Tradeoff

This section presents the energy, delay and performance comparisons for the filter cache hierarchies incorporating just the pattern predictor versus the hierarchy with both the pattern predictor and dynamic management scheme for the filter cache. To illustrate the effectiveness of the dynamic filter cache management technique in leveraging on the pattern predictor, the relative improvements of the combined scheme is presented in the sections that follow. The net result is improved performance at lower energy requirements.

6.3 Energy-Delay Tradeoff

Algorithm 4 Algorithm for Dynamic Filter Cache Management

```

if FilterCacheEnabled = TRUE then
  if PredictedHitInFilterCache = TRUE then
    Access from Filter Cache
    if WrongFilterCaheHitPrediction then
      Access from L1 Cache
      Transfer Line to Filter Cache
    end if
    if UpdatePHTandSR = TRUE then
      Update Hit/Miss Result in PHT & SR
    end if
  else
    Access from L1 Cache
    Transfer Line to Filter Cache
    if UpdatePHTandSR = TRUE then
      Update Hit/Miss Result in PHT & SR
    end if
  end if
  if FilterCacheLineHit = TRUE then
    Increment the Filter Cache Disable Saturating Counter
  else
    Decrement the Filter Cache Disable Saturating Counter
    If the Filter Cache Disable Counter is Zero, Disable the Filter Cache
  end if
else
  if UseLineBuffer = TRUE then
    Get the Instruction from the Line Buffer
  else
    Access from L1 Cache
    Transfer Line to Line Buffer
  end if
end if
if BackwardBranch = TRUE then
  Enable the Filter Cache
  Load Value ( $\frac{N}{2}$ ) in to the Counter
end if

```

6.3 Energy-Delay Tradeoff

Figure 6.4: AMAC Comparison for Dynamic Filter Cache. (AMAC of One Depicts No Performance Penalty)

6.3 Energy-Delay Tradeoff

6.3.1 Average Memory Access Cycles

Figure 6.4 compares the average memory access cycles or performance, obtained with dynamic filter cache management scheme. Here, the average memory access cycle is obtained by restricting our analysis only to the filter cache and L1 cache accesses. This average memory access cycle will be one if filter cache is not employed. It is clearly evident from Figure 6.4 that the dynamic filter cache improves the average memory access cycles as compared to the predictive filter cache. The average memory access cycles of the dynamic filter cache is very close to the case when filter cache is not present.

It is also evident that the optimal miss sensitivity (N) where dynamic filter cache results in minimum average memory access cycle depends on the application. For instance, benchmark ADPCM.DEC results in better average memory access cycles with an N value of 4 whereas benchmarks MPEG2.DEC results in better average memory access cycles with N value of 6. This is because, if there is a critical cacheable loop with a bigger loop size, then the N value needed to exploit temporal locality is higher. If we use a smaller N value, the filter cache will get disabled while executing this critical loop; this will degrade the predictor performance. For such benchmarks when the value of N is increased, the performance improves. The next section provides the partially cacheable loop statistics for the benchmarks simulated and their effect on miss sensitivity.

6.3.2 Normalized Energy

Figure 6.5 compares the normalized energy for various filter cache sizes with different N values. Here, the normalized energy is obtained by restricting our analysis only to filter cache and L1 cache accesses. This normalized energy will be one if filter cache is not employed. It is clear from Figure 6.5 that dynamic filter cache results in comparable energy savings as compared to the predictive filter cache. In most cases, it results in improved energy savings. It is also evident from Figure 6.5 that the optimal value of N at which the energy reduction is maximum depending on the applications loop characteristics. This is again because of the correlation between the critical loop size and the value of N as explained in the previous section. However, the energy

6.3 Energy-Delay Tradeoff

Figure 6.5: Normalized Energy Consumption for Dynamic Filter Cache (Includes the Overhead for the Pattern Predictor Logic)

6.3 Energy-Delay Tradeoff

Figure 6.6: Normalized Energy-Delay Product Comparison for Dynamic Filter Cache

6.3 Energy-Delay Tradeoff

reduction also depends on the energy cost for keeping the filter cache active and the temporal locality possible with the high N value.

For the benchmarks simulated, Table 6.1 shows the loops which can be partially cacheable in the filter cache and, still temporal locality is possible as only part of the loop is accessed from the filter cache during the next iteration. For a filter cache size of 256 Byte, the loop size of 64 instructions (256Byte) to 128 instructions (512Byte) will be in the partially cacheable range. As for 512 Byte, this range is 128 instructions (512Byte) to 256 instructions (1024Byte). Here, the loop sizes refer to the filter cache size needed to contain the loop.

As can be seen from Figure 6.5, for ADPCM.DEC with the filter cache size of 256Byte, the loop with 128Byte is the maximum cacheable limit. This is assuming that there are no conflicts between the instructions in the loop. But, in reality, few of the instructions will conflict with others. And also, the instruction line misses will not be continuous and, it will be followed by hits (The miss sensitivity is an up down saturating counter, where hit will count-up and miss will count-down). Thus, as can be seen from the Figure 6.5, for ADPCM.DEC with the 256 Byte filter cache, when the N value is increased, the filter cache gets disabled in the middle. Thus, the energy consumption also increases, as possible temporal locality is not exploited. When the filter cache size is increased to 512 Byte, this loop can be completely contained, thus there is not much change in the energy savings with the change in N value. However, the performance will suffer if the loops are executed in intervals rather than continuously as in ADPCM.DEC. Similar inferences can be made for other benchmarks based on the loop characteristics of the applications.

6.3.3 Normalized Energy-delay Product

Figure 6.6 compares the energy-delay product reduction of predictive filter cache with the dynamic filter cache. It can be clearly seen that the dynamic filter cache results in improved energy-delay reduction as compared to predictive filter cache for most of the benchmarks.

Table 6.2 summarizes the average percentage energy reduction and average percentage cycle time degradation for the filter cache. Thus, it is clear that the dynamic

6.3 Energy-Delay Tradeoff

Benchmark	Loop Size (inst.)	Loop Size (Byte)	Percentage of Time Executed
ADPCM-DEC	128	512	99
ADPCM-ENC	165	660	99
EPIC	68-152	272-608	2-5
UNEPIC	70-160	280-640	14-31
G721.DEC	137	548	45
G721.ENC	145	580	37
GSM.TOAST	99-217	396-868	5-35
GSM.UNTOAST	133-168	532-672	3-77
MIPMAP	168	672	9
OSDEMO	79-239	316-956	64-85
TEXGEN	79-139	316-556	4-34
MPEG2.DEC	90-113	360-452	36-36
MPEG2.ENC	118-167	472-668	1-59
PEGWIT	69	276	33
PEGWIT.DEC	69	276	32
PEGWIT.ENC	69	276	31
JPEG.DEC	65-185	250-740	10-27
JPEG.ENC	81-121	324-484	3-5

Table 6.1: Loop of Application That Cannot be Fully Represented in Filter Cache

Filter cache	Predictive filter cache		DYN-4		DYN-6		DYN-8	
	APACD	APER	APACD	APER	APACD	APER	APACD	APER
256	2.47%	42.60%	0.96%	46.48%	1.23%	46.40%	1.39%	46.36%
512	1.68%	53.76%	0.95%	54.45%	1.06%	53.36%	1.14%	53.43%

Table 6.2: Average Percentage Access Cycles Degradation (APACD) and Average Percentage Energy Reduction (APER)

6.4 Summary

filter cache substantially reduces the energy-delay product in the filter cache based hierarchy, while limiting the performance penalty.

6.4 Summary

Filter cache depends on the temporal locality of loops which can be cached till their next iteration. If the size of the loop is larger than that can be accommodated in the filter cache, then it will not have any real benefit other than acting as a line buffer. In this chapter we have shown that, by using simple loop detection based dynamic filter cache management scheme, the performance penalty associated with the filter cache hierarchy can be reduced to a negligible level. The proposed novel dynamic filter cache management scheme is simple and leads to further reduction in the energy-delay product when combined with the pattern prediction scheme. Moreover, the dynamic filter cache management technique allows for configuration through its miss-sensitivity parameter. This provides for an avenue to further tweak the dynamic loop management mechanism. In the next chapter we propose a unified framework for generating a filter cache based custom instruction cache hierarchy that provides for an overall reduction in the energy-delay product through systematic methods to identify optimal cache sizes and to configure both the predictor and dynamic management units of the filter cache hierarchy.

Chapter 7

A Framework for Customizing Predictor Based Instruction Hierarchy

7.1 Introduction

In the previous chapters, we have reported the effectiveness of the filter cache hierarchy for energy-delay product reduction by limiting the performance penalty in the filter cache hierarchy to negligible levels. In addition, we have shown the potential benefit of tuning the cache hierarchy in an application centric manner and proposed efficient methodologies to customize the various components of the instruction hierarchy. This provides for a custom filter cache based instruction hierarchy to be realized without increasing the design time. In this chapter, we present a systematic framework that incorporates these various components to tune the filter cache hierarchy.

7.2 Components of the Framework

As explained in the earlier chapter, filter cache based instruction hierarchy reduces the overall energy-delay product of the instruction memory hierarchy. And also, optimizing the filter cache configuration in an application specific manner can lead to a further reduction in the energy-delay product. Thus, there exists a need to customize the filter cache hierarchy to the needs of the application in a systematic manner to maximize the energy-delay product reduction. Figure 7.1 illustrates a general structure in which the

7.2 Components of the Framework

Figure 7.1: Proposed Techniques for Custom Low-Energy Instruction Hierarchy

various techniques contribute towards realizing an energy efficient instruction hierarchy are highlighted.

In the filter cache based instruction hierarchy, the configurations that influence the overall energy-delay product reduction are:

- The Filter Cache Configuration
- Predictor Tuning
- The L1 Cache Configuration

7.2 Components of the Framework

- Dynamic Cache Management Configuration

Thus, these techniques and configurations have to be invoked in an application centric manner to realize the energy-delay product reduction in the filter cache hierarchy. As shown in Figure 7.1, loop profiling information is used in various stages of the design to generate near-optimal filter cache hierarchy in an application centric manner. The following subsections summarize the techniques for configuring the filter cache hierarchy.

7.2.1 Selecting The Filter Cache Size

We have reported in Chapter 3 and 4 that the filter cache is effective in reducing the dynamic energy in the instruction memory hierarchy. And also, the filter cache results in maximal energy savings when the cache hierarchy is tuned for a given dedicated application. Thus, we presented a methodology to tune the filter cache in an application specific manner using the loop profiler output. In the proposed loop profiler directed analytical model for calculating the optimal filter cache, we specify whether we want an aggressive or relaxed filter cache size selection. That is, while selecting the near-optimal filter cache, we designate only the hot paths within the loop as filter cache resident. To do this, we set a threshold, which is a fraction of the loop iteration. Thus, by changing this value, we can either go for aggressive or relaxed filter cache. This will in turn affect the overall reduction achieved in the energy-delay product.

Figure 7.2 shows the methodology for custom filter cache hierarchy generation, where the custom filter cache synthesis is done first. As can be seen from the Figure 7.2, the filter cache size is then fed into the custom predictor configuration selection and dynamic filter cache management scheme selection. The next subsection presents the tuning of the filter cache predictor. This is independent of the L1 cache tuning process, which can proceed alongside.

7.2.2 Predictor Selection and Configuration

Once we have selected the filter cache configuration, the filter cache predictor and the predictor configurations have to be decided for the application. As we have reported earlier, the NFPT predictor often results in inferior prediction accuracy compared to

7.2 Components of the Framework

pattern predictor and the static pattern predictor. In addition, the hardware complexity of the NFPT is determined by the filter cache size and often takes more VLSI area. Thus, we rely on the pattern predictor and the static pattern predictor for predicting the hit and misses in the filter cache and for reducing the energy-delay product. Therefore, we need to choose between pattern predictor and the static pattern predictor. In addition, the number of entries in the PHT/PT also needs to be decided.

The PHT/PT entries maintain the hit/miss patterns of the instruction lines, thus, it depends on the loop size of the frequent loops and the conflicts of patterns between different loops. The choice is also influenced by the filter cache size. This is because, if the frequent loops can be completely cacheable in the filter cache, the hit/miss patterns can easily be represented with smaller shift_register and PHT/PT entries. Thus, firstly, we select the PHT entries based on the application's requirements. And, we go for the pattern predictor with the calculated number of PHT entries. If that satisfies the hardware cost, we select this as the predictor configuration. If the hardware cost is not satisfied, we select the static pattern predictor with the calculated number of PT entries. If this also does not satisfies the hardware cost, we reduce the PT entries till the hardware cost is satisfied. This is shown in the Figure 7.2.

Once we select the predictor configuration, we check whether this configuration satisfies the performance limit. If the performance is not satisfactory, then we will have to use the dynamic filter cache management scheme to improve the performance further.

7.2.3 Selecting The L1 Cache Parameters

As reported in Chapter 5, optimizing the L1 cache parameters for the application is also essential as it can result in a reduction of the dynamic as well as static energy, without scarifying the performance. In this thesis we have presented a methodology to optimize the L1 cache size such that the energy consumed is minimized. As discussed earlier, the proposed methodology finds the near optimal L1 cache parameter from the loop profiler output. As can be seen from the Figure 7.2, we also provide possible L1 cache sizes, from which the technique finds the near optimal L1 cache. Thus, when we have the VLSI area budget for the L1 cache, we can select the near-optimal cache configuration for maximal energy savings. In addition, we can also specify whether we

7.3 Summary

want an aggressive or relaxed L1 cache size selection, in our model, while calculating the optimal L1 cache. That is, while selecting the near-optimal L1 cache, we designate only the hot paths within the loop for caching in the L1 cache. For doing this we set a threshold, which is a fraction of $\frac{\text{Static_instruction_count}}{\text{Dynamic_instruction_count}}$. Thus, by changing this value, we can either go for aggressive or relaxed L1. This will in turn affect the overall reduction achieved in the energy-delay product.

Figure 7.2 shows the methodology for custom L1 cache hierarchy generation, where the custom L1 cache hierarchy synthesis is done with the input from the loop profiler output.

7.2.4 Criteria for Invoking Dynamic Cache Management Scheme

Finally, if the performance of the predictive filter cache is not satisfactory, the dynamic management scheme also needs to be applied to reduce the energy-delay product further. The dynamic filter cache management scheme is used so that when the filter cache is not of any use, the filter cache is disabled, thus eliminating the overhead. A simple loop detection based methodology is used to decide whether enabling filter cache is useful or not.

However, as discussed in Chapter 7, for the dynamic filter cache management scheme to be effective, the processing core has to signal when a branch is taken in the backward direction. In addition, we also need to know whether the current instruction can change the flow of the execution. Thus, if this information is not available from the processing core, the dynamic filter cache management scheme cannot be applied.

As explained earlier in chapter 7, the miss sensitivity factor influences the time for which the filter cache is enabled and thereby the energy-delay product. The optimal energy-delay product reduction depends on the frequent loop characteristics of the application. Thus the sensitivity value has to be selected based on the application and filter cache size, as shown in Figure 7.2.

7.3 Summary

Filter cache based custom instruction hierarchy is effective in reducing the energy consumed in the filter cache. The optimal filter cache based instruction hierarchy, which

7.3 Summary

Figure 7.2: Proposed Framework for Customizable Filter Cache Hierarchy Generation

7.3 Summary

results in maximal energy savings depends on the loop characteristics of the application. Thus, the loop profiling information can be used in tuning the filter cache hierarchy in an application centric manner. In this chapter, as a review, we have presented a combined approach to realize the custom filter cache hierarchy. Figure 7.2 summarizes the various stages of this customizable instruction cache hierarchy generation process.

Chapter 8

Conclusion and Recommendations

8.1 Conclusion

We have proposed a number of novel methods to reduce the energy delay product through customization of a filter cache based instruction cache hierarchy. Methods devised for customization are compute efficient and yield comparable results to that possible with exhaustive computations.

The NFPT predictor employed in filter cache based cache hierarchy was examined closely and its limitations were reported. Based on these findings, a new predictor called the ‘pattern predictor’ was proposed. Our investigations on a Simple Scalar platform using the MediaBench benchmark suite confirm that, when compared with NFPT implementation, the proposed predictor is superior both in terms of VLSI cost and prediction accuracy. Unlike NFPT whose VLSI complexity increases with filter cache size, the PHT size is independent of the filter cache size. It is evident that a relatively small PHT with a 5-bit shift register (i.e. 32 entries) is sufficient to contain the patterns exhibited by instruction lines. For example, for a 512-byte filter cache configuration, the pattern predictor with 32-entry PHT occupies only 1802 gates and outperforms NFPT, which requires 2798 gates to implement. Moreover, for the MediaBench benchmark suite, increasing the filter cache size beyond 512 bytes does not necessarily lead to a reduction in the energy delay product. This implies that most of the embedded applications benefit from a small filter cache. Further improvements to the pattern predictor architecture were made by examining the loop access patterns.

8.1 Conclusion

This has led to the proposal of a ‘static pattern predictor’ to further reduce the access energy and VLSI costs of the prediction unit due to the elimination of saturation counters.

It became evident that the filter cache size must be tailored for each application to maximize the reduction in the energy delay product. This led to the development of a loop profiler directed methodology to tune the filter cache hierarchy. To facilitate this, we have developed a tool to analyze loop statistics of applications. The mathematical models developed to tune the filter cache have been fully validated with the results from exhaustive simulations on the Simple Scalar platform. Our investigations show that, on an average, the proposed model based cache tuning process is within 1% of the ideal case. Similar results were also observed for the models developed to tune the L1 cache. The techniques devised for tuning the filter cache and the L1 cache substantially reduce the design space exploration time to compute the optimal cache sizes that provide for maximum reduction in the energy delay product. The proposed customization techniques were applied to the MediaBench benchmark suite to evaluate their effectiveness towards reducing overall access energy for both caches. Our investigations reveal that the proposed methodologies for tuning both the cache sizes can provide for up to 47% and up to 18% energy savings for the filter cache and L1 cache respectively when compared with typical configurations.

The proposed dynamic filter cache management scheme contributes to further reduction in the energy delay product when combined with the pattern prediction based cache hierarchy. Simulations confirm that the proposed dynamic management scheme can be incorporated into the pattern predictor based instruction cache hierarchy to further reduce the energy delay product by 4.9%. The proposed scheme also lends well for application specific optimization as it provides for selective tuning of the miss-sensitivity parameter to turn-on/turn-off the filter cache.

For the MediaBench benchmark suite, the proposed techniques for tuning both the pattern predictor based cache hierarchy and dynamic management scheme have been shown to reduce the overall access energy by 54.45%, while limiting the performance degradation to a mere 0.95%. We have also proposed a framework that provides an integral environment for the systematic tuning of the predictive filter cache hierarchy. With this, efficient tuning of the predictor based instruction cache hierarchy can be

8.2 Recommendations for Further Research

realized without time consuming exhaustive simulations to yield notable reduction in the overall energy delay product.

8.2 Recommendations for Further Research

Below, we have identified a number of areas that could be delved into to further this research work:

- **Evaluation with Other Benchmarks and Data Set**

Although a number of novel techniques to customize the instruction cache hierarchy have been proposed and validated with the MediaBench benchmark suite, it will be of interest to evaluate the performance of the proposed techniques with other benchmarks. Also, investigation into how the data set and their access patterns influence the energy delay product measure would be useful as this may provide for additional information to tune the cache hierarchy for data intensive applications.

- **Computing the Optimal Pattern History Table (PHT) Size**

In this research work, we have clearly established the effect of altering the PHT size in an application centric manner. In particular, we were able to relate the loop sizes to the PHT size so as to best represent the hit-miss pattern profile of the loops. Hence, it should be possible to establish a systematic framework to compute the optimal PHT size for a given spectrum of loops in an application. In addition, further research could be undertaken to devise suitable methods to represent bigger loops with a relatively small PHT by establishing a fixed offset based identification of the cache lines that need be maintained in the PHT. This may help to establish a means to further compress the PHT size.

- **Profile Directed Techniques for Tuning Cache Associativity and Line Size**

The techniques developed in this research work were based on a pre-determined associativity and line size for both filter and L1 caches. Thus, it will be of interest to further explore the potential benefits of tweaking the associativity and line

8.2 Recommendations for Further Research

sizes as part of the cache hierarchy tuning process. The challenge here would be to devise efficient techniques that yield near optimal reduction in the energy delay product without resorting to exhaustive simulations. This will be of particular interest as it may help to establish a relationship between the miss penalty and access energy with reference to the distribution profile of active loops that are similar in size and associativity.

- **Dynamic Schemes for the Leakage Power Reduction in Cache Hierarchy**

Since the leakage power is becoming a formidable issue with deep sub-micron technologies and that cache memories are transistor intensive, it would be of immense interest to explore for ways to combine advanced techniques for leakage power reduction with the proposed techniques. This is particularly so when 50% of the time is spent in about 1% of the code size in typical embedded applications. It is envisaged that the profile data used for tuning the cache hierarchy could also be employed to establish a possible link especially when most embedded applications tend to be governed by a set of mini loops. For example, the activity levels of filter cache could be used to identify and manage the inactive regions of the code in a dynamic manner by employing established techniques such as the drowsy cache schemes.

- **Predictor Based Cache Hierarchy for General Purpose Computations**

The proposed techniques show promising results for use in embedded applications. However, it would be interesting to examine the effectiveness of the pattern predictor based instruction hierarchy for general-purpose computations. Since the NFPT based approach has been demonstrated to yield improvements when applied to computations on super-scalar processor architectures, it seems appropriate to explore for an approach that could be morphed using heuristic techniques to improve the overall reduction in the energy delay product when compared with static assignments of cache hierarchy. This may also lead to a proposition in which the cache structure may be made to morph at run-time to provide for added benefits.

Appendix A

MediaBench Benchmark Suite

- **JPEG:** JPEG is a standardized compression method for full color and gray-scale images. JPEG is lossy, meaning that the output image is not exactly identical to the input image. Two applications are derived from the JPEG source code; cjpeg does image compression and djpeg, which does decompression.
- **MPEG:** MPEG2 is a dominant standard for high quality digital video transmission. The important computing kernel is a discrete cosine transform for coding and the inverse transform for decoding. The two applications used are mpeg2enc and mpeg2dec for encoding and decoding respectively.
- **GSM:** European GSM 06.10 provisional standard for full rate speech transcoding, prI-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits.
- **G.721 Voice Compression:** Reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions.

- **PEGWIT:** A program for public key encryption and authentication. It uses an elliptic curve over $GF(2255)$, SHA1 for hashing, and the symmetric block cipher square.
- **Mesa:** Mesa is a 3-D graphics library clone of OpenGL. All display output functions were removed from the library and demo programs included in the package. Three applications are used: mipmap which executes fast texture mapping using pre computed filter results, osdemo which executes a standard rendering pipeline, and texgen which generates a texture mapped version of the Utah teapot.
- **EPIC:** An experimental image compression utility. The compression algorithms are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding without floating-point hardware.
- **ADPCM:** Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding.

Bibliography

- [1] Nios embedded processor system development. <http://www.altera.com/>, 2004. 23
- [2] Tensilica, xtensa processor generator. <http://www.tensilica.com/>, 2004. 23
- [3] *The GNU Profiler*. <http://www.gnu.org>, 2005. 74
- [4] SimpleScalar. <http://www.simplescalar.com/>, 2005. 28
- [5] TOM VANDER AA, MURALI JAYAPALA, FRANCISCO BARAT, GEERT DECONINCK, RUDY LAUWEREINS, FRANCKY CATTHOOR, AND HENK CORPORAAAL. Instruction buffering exploration for low energy vliws with instruction clusters. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 824–829, 2004. 2, 18
- [6] ANANT AGARWAL AND STEPHEN D. PUDAR. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 179–190. ACM Press, 1993. 12
- [7] YAZDAN AGHAGHIRI, FARZAN FALLAH, AND MASSOUD PEDRAM. Irredundant address bus encoding for low power. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 182–187. ACM Press, 2001. 22
- [8] DAVID H. ALBONESI. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259. IEEE Computer Society, 1999. 23

BIBLIOGRAPHY

-
- [9] RAJESHWARI BANAKAR, STEFAN STEINKE, BO-SIK LEE, M. BALAKRISHNAN, AND PETER MARWEDEL. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM Press, 2002. 18
 - [10] BRANNON BATSON AND T. N. VIJAYKUMAR. Reactive-associative caches. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–60. IEEE Computer Society, 2001. 12
 - [11] NIKOLAOS BELLAS, IBRAHIM HAJJ, AND CONSTANTINE POLYCHRONOPOULOS. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 64–69. ACM Press, 1999. 20, 96
 - [12] NIKOLAOS BELLAS, IBRAHIM HAJJ, CONSTANTINE POLYCHRONOPOULOS, AND GEORGE STAMOULIS. Energy and performance improvements in microprocessor design using a loop cache. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 378–383. IEEE Computer Society, 1999. 18, 19
 - [13] L. BENINI, G. DE MICHELI, E. MACII, D. SCIUTO, AND C. SILVANO. Address bus encoding techniques for system-level power optimization. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 861–867. IEEE Computer Society, 1998. 22
 - [14] LUCA BENINI AND GIOVANNI DE MICHELI. System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, **5**(2):115–192, 2000. 2, 19
 - [15] LUCA BENINI, GIOVANNI DE MICHELI, ENRICO MACII, DONATELLA SCIUTO, AND CRISTINA SILVANO. Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems. In *GLS '97:*

BIBLIOGRAPHY

- Proceedings of the 7th Great Lakes Symposium on VLSI*, pages 77–82. IEEE Computer Society, 1997. 22
- [16] DOUG BURGER, TODD M. AUSTIN, AND STEVE BENNETT. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996. 28, 74
- [17] DAVID CALLAHAN, KEN KENNEDY, AND ALLAN PORTERFIELD. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52. ACM Press, 1991. 12
- [18] J. H. CHANG, H. CHAO, AND K. SO. Cache design of a sub-micron cmos system/370. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 208–213. ACM Press, 1987. 10, 17, 18
- [19] NAEHYUCK CHANG, KWANHO KIM, AND JINSUNG CHO. Bus encoding for low-power high-performance memory systems. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 800–805. ACM Press, 2000. 22
- [20] TIEN-FU CHEN AND JEAN-LOUP BAER. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 51–61. ACM Press, 1992. 12
- [21] BRUCE R. CHILDERS AND JACK W. DAVIDSON. Custom wide counterflow pipelines for high-performance embedded applications. *IEEE Trans. Computers*, **53**(2):141–158, 2004. 18, 32
- [22] ROBERT F. CMELIK. Spixtools, introduction and user's manual. Technical Report TR-93-06, Sun Microsystems Inc, 1993. 74
- [23] CATTHOOR F, K. DANCKAERT, C. KULKARNI, E. BROCKMEYER, P. KJELDSBERG, T. VAN ACHTEREN, AND T OMNES. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002. 2, 19

BIBLIOGRAPHY

- [24] KRISZTIN FLAUTNER, NAM SUNG KIM, STEVE MARTIN, DAVID BLAAUW, AND TREVOR MUDGE. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 148–157. IEEE Computer Society, 2002. 23
- [25] KANAD GHOSE AND MILIND B. KAMBLE. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 70–75. ACM Press, 1999. 17, 98
- [26] ARIJIT GHOSH AND TONY GIVARGIS. Analytical design space exploration of caches for embedded systems. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 10650–10656. IEEE Computer Society, 2003. 25
- [27] ANN GORDON-ROSS, SUSAN COTTERELL, AND FRANK VAHID. Tiny instruction caches for low power embedded systems. *Trans. on Embedded Computing Sys.*, **2**(4):449–481, 2003. 20
- [28] A. HASEGAWA, I.KAWASAKI, K.YAMADA, S.YOSHIOKA, S. KAWASAKI, AND P. BISWAS. Sh3: High code density, low power. In *IEEE Micro*, pages 11–19. IEEE Press, 1995. 17
- [29] M. D. HILL AND A. J. SMITH. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, **38**(12):1612–1630, 1989. 24
- [30] K. INOUE, K. KAI, AND K. MURAKAMI. Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged dram/logic lsis. In *HPCA '99: Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, pages 218–222. IEEE Computer Society, 1999. 11
- [31] KOJI INOUE, TOHRU ISHIHARA, AND KAZUAKI MURAKAMI. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 273–275. ACM Press, 1999. 17

BIBLIOGRAPHY

-
- [32] KOJI INOUE, VASILY MOSHNYAGA, AND KAZUAKI MURAKAMI. Bus encoding for low-power high-performance memory systems. In *IEICE Transactions on Electronics*, pages 304–314, Feb. 2002. 17
 - [33] NORMAN P. JOUPPI. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373. ACM Press, 1990. 12
 - [34] MILIND B. KAMBLE AND KANAD GHOSE. Analytical energy dissipation models for low-power caches. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 143–148. ACM Press, 1997. 17
 - [35] STEFANOS KAXIRAS, ZHIGANG HU, AND MARGARET MARTONOSI. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251. ACM Press, 2001. 23
 - [36] R.E. KESSLER, R. JOOSS, A. LEBECK, AND M.D. HILL. Inexpensive implementations of set-associativity. In *Proc. 16th Int. Symp. on Computer Architecture (ISCA '89)*, pages 131–139. ACM Press, May/June 1989. 10
 - [37] NAM SUNG KIM, TODD AUSTIN, AND TREVOR MUDGE. Low-energy data cache using sign compression and cache line bisection. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues WMPI'02*, May 2002. 16
 - [38] NAM SUNG KIM, KRISZTIN FLAUTNER, DAVID BLAAUW, AND TREVOR MUDGE. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 219–230. IEEE Computer Society Press, 2002. 23
 - [39] JOHNSON KIN, MUNISH GUPTA, AND WILLIAM H. MANGIONE-SMITH. The filter cache: an energy efficient memory structure. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184–193. IEEE Computer Society, 1997. 14, 18, 20, 29, 30, 33, 34, 50, 53

BIBLIOGRAPHY

- [40] ALEXANDER C. KLAIBER AND HENRY M. LEVY. An architecture for software-controlled data prefetching. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 43–53. ACM Press, 1991. 12
- [41] MAKOTO KOBAYASHI. Dynamic characteristics of loops. In *IEEE Trans. Computers*, **33**, pages 125–132. IEEE Press, 1984. 73
- [42] S. LAHA, J. H. PATEL, AND R. K. IYER. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, **37**(11):1325–1336, 1988. 24
- [43] CHUNHO LEE, MIODRAG POTKONJAK, AND WILLIAM H. MANGIONE-SMITH. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997. 31, 32, 33
- [44] LEA HWANG LEE, BILL MOYER, AND JOHN ARENDS. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 267–269. ACM Press, 1999. 18, 19
- [45] HARIS LEKATSAS AND WAYNE WOLF. Random access decompression using binary arithmetic coding. In *DCC '99: Proceedings of the Conference on Data Compression*, pages 306–315. IEEE Computer Society, 1999. 16
- [46] LISHING LIU. Cache designs with partial address matching. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 128–136. ACM Press, 1994. 10
- [47] SCOTT MCFARLING. Combining branch predictors. Technical Report TN-36, June 1993. 45, 46
- [48] JAMES MONTANARO, RICHARD T. WITEK, KRISHNA ANNE, ANDREW J. BLACK, ELIZABETH M. COOPER, DANIEL W. DOBBERPUHL, PAUL M. DONAHUE, JIM ENO, GREGORY W. HOEPPNER, DAVID KRUCKEMYER, THOMAS H. LEE, PETER C. M. LIN, LIAM MADDEN, DANIEL MURRAY,

BIBLIOGRAPHY

- MARK H. PEARCE, SRIBALAN SANTHANAM, KATHRYN J. SNYDER, RAY STEPHANY, AND STEPHEN C. THIERAUF. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *Digital Tech. J.*, **9**(1):49–62, 1997. 13, 28, 34
- [49] STEVEN S. MUCHNICK. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997. 77
- [50] STEPHEN C. NORTH AND ELEFTHERIOS KOUTSOFIOS. Applications of graph visualization. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1994. 75
- [51] P. R. PANDA, N. D. DUTT, AND A. NICOLAU. Data cache sizing for embedded processor applications. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 925–926. IEEE Computer Society, 1998. 25
- [52] PREETI RANJAN PANDA, NIKIL D. DUTT, AND ALEXANDRU NICOLAU. Architectural exploration and optimization of local memory in embedded systems. In *ISSS '97: Proceedings of the 10th international symposium on System synthesis*, pages 90–97. IEEE Computer Society, 1997. 25
- [53] PREETI RANJAN PANDA, NIKIL D. DUTT, AND ALEXANDRU NICOLAU. Memory data organization for improved cache performance in embedded processor applications. *ACM Trans. Des. Autom. Electron. Syst.*, **2**(4):384–409, 1997. 2, 19
- [54] RAMESH PANWAR AND DAVID RENNELS. Reducing the frequency of tag compares for low power i-cache design. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 57–62. ACM Press, 1995. 18
- [55] DAVID A. PATTERSON AND JOHN L. HENNESSY. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 1990. 7, 8, 12, 17
- [56] JIH-KWON PEIR, YONGJOON LEE, AND WINDSOR W. HSU. Capturing dynamic memory reference behavior with adaptive cache topology. *SIGPLAN*, **33**(11):240–250, 1998. 12

BIBLIOGRAPHY

- [57] MICHAEL POWELL, SE-HYUN YANG, BABAK FALSAFI, KAUSHIK ROY, AND T. N. VIJAYKUMAR. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 90–95. ACM Press, 2000. 22
- [58] THOMAS ROBERTS PUZAK. *Analysis of cache replacement-algorithms*. PhD thesis, 1985. 24
- [59] AMITABH SRIVASTAVA AND ALAN EUSTACE. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, **29**(6):196–205, 1994. 74
- [60] MIRCEA R. STAN AND WAYNE P. BURLESON. Bus-invert coding for low-power i/o. *IEEE Trans. Very Large Scale Integr. Syst.*, **3**(1):49–58, 1995. 22
- [61] CHING-LONG SU AND ALVIN M. DESPAIN. Cache design trade-offs for power and performance optimization: a case study. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 63–68. ACM Press, 1995. 17, 18, 98
- [62] CHING-LONG SU, CHI-YING TSUI, AND ALVIN M. DESPAIN. Saving power in the control path of embedded processors. *IEEE Des. Test*, **11**(4):24–30, 1994. 22
- [63] R. A. SUGUMAR AND S. G. ABRAHAM. efficient simulation of multiple cache configurations using binomial trees. *Technical Report CSE-TR-111-91*, 1991. 24
- [64] DINESH C. SURESH, WALID A. NAJJAR, FRANK VAHID, JASON R. VILLARREAL, AND GREG STITT. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 189–198. ACM Press, 2003. 74
- [65] Synopsys Inc., www.synopsys.com. *Synopsys CoCentric Compiler Behavioral User and Modeling Guide*, 2004. 66
- [66] Synopsys Inc., <http://www.systemc.org>. *SystemC Reference Manual*, 2004. 66

BIBLIOGRAPHY

- [67] W. TANG, R. GUPTA, AND A. NICOLAU. Power savings in embedded processors through decode filter cache. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, pages 443–448. IEEE Computer Society, 2002. 20, 30, 37
- [68] WEIYU TANG, RAJESH K. GUPTA, AND ALEXANDRU NICOLAU. Design of a predictive filter cache for energy savings in high performance processor architectures. In *ICCD*, pages 68–75, 2001. 20, 21, 36, 37, 50, 97
- [69] Texas Instruments Inc., <http://www.ti.com>. *TMS320C6000 Power Consumption Summary*, 2004. 1, 18
- [70] J. TUBELLA AND A. GONZLEZ. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA '98: Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture*, pages 14–23. IEEE Computer Society, 1998. 74
- [71] LUIS VILLA, MICHAEL ZHANG, AND KRSTE ASANOVIC;. Dynamic zero compression for cache energy reduction. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 214–220. ACM Press, 2000. 16
- [72] JASON VILLARREAL, ROMAN LYSECKY, SUSAN COTTERELL, AND FRANK VAHID. Loop behavior of embedded programs. Technical Report UCR-CSE-01-03, 2001. 18, 32, 74, 101
- [73] WEN-HANN WANG AND JEAN-LOUP BAER. Efficient trace-driven simulation method for cache performance analysis. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 27–36. ACM Press, 1990. 24
- [74] S. WILTON AND N. JOUPPI. Cacti: An enhanced cache access and cycle time model. Technical report, 1996. 28

BIBLIOGRAPHY

-
- [75] EMMETT WITCHEL, SAM LARSEN, C. SCOTT ANANIAN, AND KRSTE ASANOVI;. Direct addressed caches for reduced power consumption. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 124–133. IEEE Computer Society, 2001. 18
 - [76] ANDREW WOLFE AND ALEX CHANIN. Executing compressed programs on an embedded risc architecture. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91. IEEE Computer Society Press, 1992. 16
 - [77] ZHAO WU AND WAYNE WOLF. Iterative cache simulation of embedded cpus with trace stripping. In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 95–99. ACM Press, 1999. 24
 - [78] WM. A. WULF AND SALLY A. MCKEE. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, **23**(1):20–24, 1995. 7
 - [79] SE-HYUN YANG, BABAK FALSAFI, MICHAEL D. POWELL, KAUSHIK ROY, AND T. N. VIJAYKUMAR. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *HPCA '01: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 147–158. IEEE Computer Society, 2001. 22
 - [80] TSE-YU YEH AND YALE N. PATT. Alternative implementations of two-level adaptive branch prediction. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 451–461. ACM Press, 1998. 45
 - [81] ADI YOAZ, MATTAN EREZ, RONNY RONEN, AND STEPHAN JOURDAN. Speculation techniques for improving load related instruction scheduling. *SIGARCH Comput. Archit. News*, **27**(2):42–53, 1999. 45
 - [82] YUKIHIRO YOSHIDA, BAO-YU SONG, HIROYUKI OKUHATA, TAKAO ONOYE, AND ISAO SHIRAKAWA. An object code compression approach to embedded processors. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 265–268. ACM Press, 1997. 16

BIBLIOGRAPHY

- [83] CHENXI ZHANG, XIAODONG ZHANG, AND YONG YAN. Two fast and high-associativity cache schemes. *IEEE Micro*, **17**(5):40–49, 1997. 12
- [84] CHUANJUN ZHANG AND FRANK VAHID. Cache configuration exploration on prototyping platforms. In *RSP '03: Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, page 164. IEEE Computer Society, 2003. 24, 73
- [85] CHUANJUN ZHANG, FRANK VAHID, AND WALID NAJJAR. A highly configurable cache architecture for embedded systems. *SIGARCH Comput. Archit. News*, **31**(2):136–146, 2003. 23, 24, 29, 72, 89, 91
- [86] YUTAO ZHONG, STEVEN G. DROPSHO, AND CHEN DING. Miss rate prediction across all program inputs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 79–90. IEEE Computer Society, 2003. 25
- [87] DANIEL F. ZUCKER, MICHAEL J. FLYNN, AND RUBY B. LEE. A comparison of hardware prefetching techniques for multimedia benchmarks. In *ICMCS*, pages 236–244, 1996. 12