Profile driven schemes for energy-sensitive cache hierarchy

Santanu Kumar Dash

2009

Santanu, K. D. (2009). Profile driven schemes for energy-sensitive cache hierarchy.
Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/19286>

<https://doi.org/10.32657/10356/19286>

PROFILE DRIVEN SCHEMES FOR ENERGY-SENSITIVE CACHE HIERARCHY

SANTANU KUMAR DASH

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2009

Acknowledgments

I would like to take this opportunity to thank many people who have directly and indirectly helped me find my foothold in research. I would like to express my heartfelt gratitude to my supervisor and mentor Dr. Thambipillai Srikanthan. His guidance during the course of this project has been invaluable. Despite being entrusted with many responsibilities, he has always found time to discuss the directions for this project with me. His patience and understanding in this regard is truly admirable.

I would like to thank Mr. Kugan Vivekanandarajah for helping me get a headstart in this project. His advice and insights truly helped me jumpstart my research process. I would like to thank him for guiding me on the simulation tools available as well.

The administrative support rendered by Ms. Nah, Merilyn and Jeremiah was the best I could have asked for. They always helped me out whenever I needed any equipment or administrative work to be sorted out. They are truly an integral part of the research lab and their presence is absolutely necessary for the day-to-day functioning. I want to thank them for all the help they have provided me with during my time at CHiPES.

This acknowledgement is incomplete without mentioning my dear friends with whom I have shared many wonderful moments. They always cheered me up with their jokes and antics and I will miss them sorely. I want to thank Ashish (Oriya Bhai), Alok (Bihari), Amit (Khodu), Bharath (Motu) and Abhijit (Taklu Bhai) for all the great times. They will all remain my lifelong friends.

Last but not the least, I would like to thank Meethu, Ma, Baba and Ronnie for their endless support. There were times when results were hard to come by and often it was frustrating. But the faith they showed in me spurred me on to give my best regardless of the difficulties I faced. I feel truly blessed to have such a strong support system around myself.

Contents

Acknowledgments	1
List of Figures	7
List of Tables	9
Abstract	10
1 Introduction	12
1.1 Motivation	12
1.2 Major contributions of this work	13
1.3 Organization of the Thesis	14
1.4 Publications resulting from this work	16
2 Literature Survey	18
2.1 Introduction	18
2.2 Principles of Memory Hierarchy	19
2.3 Architectural Level Techniques for High Performance	20
2.3.1 Reducing Hit Time	20
2.3.2 Increasing Hit Rate	22
2.3.3 Reducing Miss Penalty	23
2.4 Architecture Level Techniques for Power Reduction	24
2.4.1 Dynamic Energy Reduction Techniques	25
2.4.2 Static Energy Reduction Techniques	30
2.5 Architectural-level techniques for low power and high performance	30
2.5.1 Predictor based filter cache hierarchy	31
2.5.2 Prediction algorithms	33
2.6 Application-specific customization of cache memories	36

2.6.1	Tunable cache memories	36
2.6.2	Design space exploration for cache tuning	37
2.7	Summary	44
3	Cache tuning for single-task based embedded applications	46
3.1	Introduction	46
3.2	A discussion on locality of reference	48
3.3	Drawbacks of existing profile-driven cache tuning techniques	49
3.3.1	Limiting loop exploration to reducible loops	49
3.3.2	Oversight of function calls in calculation of loop sizes	51
3.3.3	Insensitivity to sparsely populated cache lines	51
3.4	An improved approach to cache tuning	52
3.4.1	Representing the application for loop profiling	53
3.4.2	Methodology for extracting loop characteristics	54
3.5	Estimation of cache hit rates from loop characteristics	60
3.5.1	Estimation of Spatial Hits	60
3.5.2	Estimation of Temporal Hits	61
3.6	Tuning Heuristics for Predictor Based Filter Cache Hierachy	63
3.7	Overview of the cache tuning framework	67
3.8	Experimental results	70
3.8.1	Estimation of cache hit rates	70
3.8.2	Comparison of proposed techniques with exhaustive simulation	72
3.9	Proposed techniques for tuning of block sizes and associativity	74
3.9.1	Evaluation of block sizes	75
3.9.2	Evaluation of associativity	77
3.10	Summary	78
4	Cache tuning for RTOS-driven multitasking applications	80
4.1	Introduction	80
4.2	Analysis of program execution in multitasking applications	82
4.2.1	Execution footprint of a multitasking application	82
4.2.2	Pseudo-random control flow	84

4.3	Impact of pseudo-random control flow on cache accesses	85
4.3.1	Intrinsic and extrinsic code interference	86
4.3.2	Empirical evaluation of cache misses due to code interference . . .	87
4.3.3	Theoretical explanation of empirical observations	89
4.4	Proposed technique for cache tuning for multitasking systems	91
4.4.1	Loop profiling of user tasks	93
4.4.2	Loop profiling of the RTOS	95
4.4.3	Overview of the tuning framework	96
4.5	Experimental results	98
4.5.1	Experimental Setup	98
4.5.2	Comparison of proposed techniques with exhaustive simulation . .	98
4.6	Estimation of misses due to extrinsic interference	101
4.7	Summary	104
5	RTOS models for rapid cache tuning	105
5.1	Introduction	105
5.2	Segmentation of the RTOS into components	107
5.2.1	System-call driven execution of the RTOS	107
5.2.2	Exception-driven execution of the RTOS	107
5.2.3	A multi-entry-multi-exit view of the RTOS	108
5.3	Factors influencing hit rates for RTOS components	109
5.3.1	Static factors - application and cache	109
5.3.2	Dynamic factors - scheduler	110
5.4	Methodology for modeling RTOS components	111
5.4.1	Obtaining hit rates for RTOS components	112
5.4.2	Multivariate models for the estimated hit rates	115
5.5	Overview of the RTOS modeling framework	117
5.6	Experimental results	118
5.7	Summary	119

6	A generic framework for cache tuning	122
6.1	Introduction	122
6.2	Target-independent profiling for target-specific loop characteristic estimation	123
6.2.1	High-level representation of the application	124
6.2.2	Obtaining code frequencies using baseline simulation	124
6.2.3	Estimation of code size for a target instruction set	125
6.2.4	Overview of target-independent profiling	125
6.3	A generic tuning framework for multitasking applications	127
6.4	Summary	128
7	Conclusion and Recommendations	131
7.1	Conclusion	131
7.2	Recommendations for future work	134
	References	137

List of Figures

2.1	Predictor based filter cache	32
2.2	Analytical cache tuning techniques for the filter cache hierarchy	38
3.1	Different DFS walks can identify different loops in the same CFG	50
3.2	Representation of the application using call graph and control flow graphs	53
3.3	Bottom-up call graph processing	54
3.4	Calculation of function sizes	56
3.5	Filter cache prediction methodology	64
3.6	Hit Rate estimation framework	68
3.7	Actual vs. estimated hit rates	71
3.8	Relative EDP of cache configurations reached by estimation and exhaus- tive simulation as compared to a 32 kB L1 cache	73
4.1	Execution footprint of a multitasking application	83
4.2	Flow of control in multitasking systems	86
4.3	Evaluation of intrinsic and extrinsic interference	87
4.4	Optional caption for list of figures	89
4.5	Effect of extrinsic interference	90
4.6	Cache tuning methodology for multitasking systems	94
4.7	Cache tuning framework for multitasking systems	97
4.8	Savings in EDP achieved by estimation algorithm vs. exhaustive simulation	100
4.9	Optional caption for list of figures	103
5.1	Multivariate nature of instruction cache hits in read(x)	110
5.2	Testcase results used for modeling instruction cache performance for timer interrupts	113

5.3	Overview of the modeling framework	118
6.1	Overview of target-independent profiling	126
6.2	Generic Multitasking Cache Tuning Framework	129

List of Tables

3.1	Actual vs. assumed line utilization for different benchmarks	52
3.2	Instruction cache parameters for design space exploration	72
3.3	Filter cache and L1 cache sizes in Bytes chosen by estimation techniques and exhaustive simulation	73
3.4	Comparison of the average runtime for the estimation algorithm and ex- haustion simulation techniques	74
4.1	Average increase in miss rate and energy delay product	88
4.2	Instruction cache parameters for design space exploration	99
4.3	Filter cache and L1 cache sizes chosen by the estimation algorithm and the exhaustive simulation	99
4.4	Comparison of the average runtime for the estimation algorithm and ex- haustion simulation techniques	100
5.1	Mean difference in hit rates between the model based and exhaustive sim- ulation based approach	120

Abstract

With the advent of mobile and handheld devices, power consumption in embedded systems has become a key design issue. Of the components that consume significant amounts of power in an embedded system, cache memories have been reported to consume in excess of 40% of the total power in typical high end embedded processors [38]. Therefore, cache memories are an obvious target of many low-power optimizations. Recently, it has been shown that cache requirements of the applications vary widely [74] and a significant amount of energy spent in cache accesses can be saved by tuning the cache parameters according to the needs of the application[60][62]. However, tuning the cache memory to suit the needs of the application entails identification of optimal cache configurations in the first place. With the large set of configurations to choose from, this process is prohibitively time consuming if done through exhaustive cache hierarchy simulations. Therefore, there exists a need for tools that can rapidly identify optimal cache configurations to tune the cache parameters for any given application.

In this research work, we present novel, lightweight and fast techniques for energy-sensitive tuning of the instruction cache hierarchy. The proposed techniques rely on profiling the application to identify its loop characteristics. The loop characteristics are then used to identify the optimal instruction cache size for any application. The techniques are initially proposed for single-task based systems. Subsequently, they are extended to rapidly identify optimal instruction cache size for multitasking applications too. Cache tuning for RTOS-driven multitasking applications is achieved by intelligently separating the user tasks and RTOS components and profiling them in isolation to identify the nature of loops in them. We apply the proposed techniques to tune a predictor based filter cache hierarchy for instructions for both single-task based applications and RTOS-driven multitasking applications. The proposed techniques are able to identify optimal or

near-optimal filter and L1 cache sizes for all the applications tested and are up to an order of magnitude faster than exhaustive cache hierarchy simulation techniques. The proposed techniques are also highly scalable and can be relied upon to predict the instruction cache hit rate for any range of instruction cache sizes after a one-time simulation and profiling.

To further decrease the time taken to simulate multitasking applications and identify optimal cache sizes, we present strategies for breaking up the RTOS into components to model their effect on the cache tuning process. Once built, these RTOS models save precious simulation time because they can be referred to rapidly estimate instruction cache behaviour of the RTOS components without actually needing to run the RTOS. For modeling the RTOS, we propose and validate effective non-parametric regression strategies to link the instruction cache hit rates for the RTOS components with the factors that influence them.

Since different combinations of RTOS and target platforms exhibit different cache access behaviours, the optimal cache size for each RTOS-target combination will be different. Therefore, in this research work, we also propose a generic cache tuning framework wherein it is possible to identify optimal instruction cache sizes for any RTOS-platform pair after simulating the application on a baseline processor.

Chapter 1

Introduction

1.1 Motivation

Power analysis of embedded processors has indicated that up to 40 % of the processor power is consumed in the cache [38]. With the widespread use of power-sensitive battery-powered devices, cache memories have therefore been an obvious target for low-power optimizations lately. While most microcontroller vendors adopt a one-size-fits-all approach when it comes to cache memories, it has been shown that cache requirements vary from one application to another. A fixed cache size, block size and associativity often results in unnecessary dissipation of energy because an application may end up using a cache that is larger or smaller than what it needs. Using too large a cache results in more energy spent per access while using a smaller cache results in increased number of misses and this leads to additional energy being spent and delay being incurred in accessing the main memory. Based on these observations, cache tuning was recently proposed to reduce the energy expended in accessing the cache [3][34][74]. It is the process of customizing the cache by tuning its parameters like cache size, block size and associativity in an application specific manner and has been shown to significantly reduce power consumption in cache memories [60][62].

While tuning the cache results in considerable reduction in the energy spent in accessing the cache, the identification of optimal cache parameters for modern-day embedded

applications poses significant challenges. Firstly, the search space for cache configurations is huge and exhaustive simulation of the cache hierarchy to identify optimal cache parameters is prohibitively time consuming. Secondly, growing complexity in embedded applications is leading to pervasive use of RTOS-driven multitasking applications. When multiple tasks and the RTOS share the cache, they jointly impact the amount of energy expended in the cache. Therefore, proposed techniques to identify optimal cache configurations for embedded applications needs to be sensitive to how different tasks and the RTOS affect the optimal cache configuration in such multitasking applications. Finally, since different RTOS and target platform combinations exhibit different cache access patterns, the proposed techniques need to be generic enough to be able to identify optimal cache configurations for any application regardless of the RTOS or the target platform chosen to run the application.

Therefore, suitable strategies need to be developed to rapidly identify optimal cache configurations for embedded applications without the need for exhaustive cache simulations. The proposed techniques should be sensitive to issues in tuning the cache for both single-task based systems and multitasking systems. They also be generic enough to be extensible to any RTOS or target platform pair that is chosen to run the embedded application.

1.2 Major contributions of this work

This section gives an overview of the novel techniques and methodologies developed during the course of this research work. For the subsequent part of this thesis, an optimal or ideal cache size refers to a cache that yields the lowest value for the energy-delay product - a metric that considers both the energy spent in cache accesses and the delay incurred in accessing the cache. Details about the energy-delay product can be found in chapter 3. The main contributions of this work can be summarized as follows.

- (i) A loop profiling framework for accurate identification of both reducible and irreducible loops in the application using its weighted function call graph and weighted control flow graphs of the functions in it. Subsequent usage of this information to predict the instruction cache hit rates for a range of cache sizes. (Chapter 3)
- (ii) Techniques to tune the predictor-based filter cache hierarchy for instructions using loop profiling of the application and subsequent hit rate estimation. (Chapter 3)
- (iii) Analysis of the invocation of RTOS components in multitasking applications and intelligent segmentation of the RTOS for the purpose of loop profiling based on the analysis. (Chapter 4)
- (iv) A technique to conduct energy-centric instruction cache tuning for RTOS-driven multitasking applications using isolated loop profiling of user tasks and RTOS executions. (Chapter 4)
- (v) Estimation of context switch misses for RTOS-driven multitasking applications using loop profiling. (Chapter 4)
- (vi) Techniques to separate the RTOS into components based on the nature of RTOS invocations and statistical models to relate RTOS hit rates with the factors that influence the hit rates for these components. (Chapter 5)
- (vii) A generic methodology to perform cache tuning for any multitasking application regardless of the RTOS and the target platform used for running the application after a one-time simulation on any baseline processor. (Chapter 6)

1.3 Organization of the Thesis

In this section, we outline the contents of the thesis.

The motivation behind this research work and its main contributions are highlighted in chapter 1.

In chapter 2, we present a survey of the techniques suggested in the literature to improve cache performance and decrease power consumption of the cache hierarchy. In this chapter, we discuss architectural optimization of cache memories aimed at improving performance and decreasing power consumption. In this context, we discuss tunable cache memories in particular and various analytical and exhaustive techniques suggested in the literature for identifying optimal cache parameters in an application-specific manner for tuning the cache.

In chapter 3, we review existing profile driven schemes for identifying optimal cache parameters in an application-specific manner. We address the shortcomings in these techniques and propose an improved techniques for loop profiling and subsequent cache tuning. To validate our propositions, we successfully apply the proposed techniques to tune a predictor-based filter cache hierarchy for instructions. While the proposed techniques identify the optimal cache size for an application, we also discuss how our techniques can be easily tweaked to identify the optimal block size and associativity for a cache configuration.

In chapter 4, we extend the proposed techniques for cache tuning from single-task based systems to RTOS-driven multitasking systems. Herein, we develop strategies to intelligently profile multitasking applications so as to accurately derive information about the loops inherent in the application. We successfully apply the proposed techniques to tune the predictor-based filter cache hierarchy for instructions for multitasking applications in this chapter. Finally, we propose and validate techniques for estimating instruction cache misses due to cache sharing among participating tasks and the RTOS using profiling techniques.

In chapter 5, we present strategies for breaking up the RTOS into components so that their effect on the instruction cache hit rate and consequently on cache tuning can be

modeled. We also propose effective non-parametric regression modeling strategies to link the instruction cache hit rates for the RTOS components with the factors that influence them. These models save precious simulation time because once the models are built for a given RTOS, they can be subsequently referenced to rapidly estimate instruction cache behaviour of the RTOS components without actually needing to run the RTOS.

In chapter 6, we present techniques to estimate the optimal cache parameters for any single-task based or multi-task based application for any RTOS or target platform used. The techniques presented in this chapter extend the strategies presented in chapters 3 and 5 to make the techniques generic and offer a powerful and scalable way forward to tune the cache regardless of target platform or RTOS chosen to run the application.

We conclude the thesis and present directions for future research in chapter 7.

1.4 Publications resulting from this work

Journal Papers

- Santanu Kumar Dash and Thambipillai Srikanthan. "Instruction Cache Tuning for Single-Task based Embedded Applications", submitted to IET Computers & Digital Techniques. (Under Review)
- Santanu Kumar Dash and Thambipillai Srikanthan. "Instruction Cache Tuning for RTOS-driven Embedded Multitasking Applications", submitted to IET Computers & Digital Techniques. (Under Review)

Conference Papers

- Santanu Kumar Dash and Thambipillai Srikanthan. "Rapid Estimation of Instruction Cache Hit Rates Using Loop Profiling", IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 263-268, Belgium, July 2008.

CHAPTER 1. INTRODUCTION

- Santanu Kumar Dash and Thambipillai Srikanthan. "Modelling RTOS Components for Instruction Cache Hit Rate Estimation", IEEE International Symposium on Circuits and Systems (ISCAS), Taiwan, May 2009.
- Santanu Kumar Dash and Thambipillai Srikanthan. "Rapid Cache Tuning Techniques for Embedded Multitasking Systems", IFIP/IEEE Symposium on Rapid System Prototyping, France, June 2009.

Chapter 2

Literature Survey

2.1 Introduction

With the ever-rising complexity in applications, there is a constant need for faster processing. While processors have been able to scale the speed requirements imposed by modern day applications, memory systems have not been as successful. High performance computing has relied on cache memories to bridge the disparate growth in the speed of processors and memory sub-systems. Sitting in between the fast processor and slow main memory, cache memories hold frequently used program and data segments and cater to the processor when it makes a memory reference.

With the increased reliance on cache memories, several architectural-level optimizations have been proposed for the cache memory. Initially efforts in this direction were geared towards decreasing average cache memory access time so that computations could memory access could proceed even faster. However, with the recent advent of mobile and handheld devices, power consumption has taken centre stage and architectural-level optimizations are being proposed to reduce the power consumption of the cache hierarchy.

In this chapter, we discuss some of these architectural-level optimizations proposed for increasing performance and/or decreasing power consumption of cache memories. We also discuss techniques suggested for application-specific customization of the cache memories for optimal performance and low-energy consumption. The rest of the chapter

is organized as follows. In section 2.2, we talk about the principles of memory hierarchy. In section 2.3, we discuss the major architectural level techniques aimed at improving the access time, increasing the hit rate and reducing the miss penalty of the cache hierarchy. In section 2.4, we discuss the important power reduction techniques proposed for the cache hierarchy. In section 2.5, we present techniques proposed in the literature for decreasing the energy expended in accessing the cache hierarchy while mitigating the associated performance overheads. We discuss customizable cache memories in section 2.6. In that section, we also discuss techniques presented in the literature for design space exploration for identifying optimal cache configurations in an application-specific manner. The contents of the chapter are summarized in section 2.7.

2.2 Principles of Memory Hierarchy

Applications tend to reuse data and instructions that they have referenced in the recent past. Empirical observations have shown that a program typically spends 90% of the time in 10% of its code [24]. It makes perfect sense therefore to hold these frequently used data and instructions in a small high-speed memory. Cache hierarchy uses this underlying concept of program locality to bridge the high speed of processors with the low speed of memory sub-systems. Cache hierarchies tend to offset the lower speed of lower-level memories with the higher speed of expensive high-speed memories. They buffer recently accessed data or instructions to minimize the cache misses which can be very expensive in terms of latency and energy spent to retrieve the data.

Cache misses can be categorized into three major categories - compulsory, capacity and conflict misses. Compulsory misses occur when the first few accesses are made to the cache. This is when no relevant instruction or data is present in it and any access to the cache results in a miss. Capacity misses occur if the cache is not large enough contain all the blocks necessary for the execution of the program. Conflict misses are due to cache

line eviction because too many program blocks map to the same cache block. Different applications have different reference patterns and subsequently different number of conflict, capacity and compulsory misses. The amount of cache memory required for optimal performance is application specific. Since cache misses impose significant performance and energy overheads, many architectural-level optimizations aimed at improving the cache hit rate tend to reduce one of these three kind of misses and consequently, the cache access time and energy.

2.3 Architectural Level Techniques for High Performance

The performance of the memory hierarchy is measured in terms of average memory access time (AMAT) which is shown in [Eq. 2.1](#) and is defined as the average amount of time (or clock cycles) taken to access a single piece of data or instructions from the memory sub-system. In this equation, HT is the cache hit time or the time taken to retrieve data from a cache when the data is cache resident, MR is the cache miss rate and MP is the miss penalty which is the average amount of time taken to fetch data from a lower level of memory hierarchy in the event of a miss.

$$AMAT = AT + MR * MP \quad (\text{Eq. 2.1})$$

As evidenced by the AMAT equation, all efforts towards improving the performance of memory hierarchy have targeted improving the cache access time, increasing (reducing) the hit (miss) rate or reducing the miss penalty. We discuss all three classes of techniques in the subsections below.

2.3.1 Reducing Hit Time

Reducing the hit time for a cache can be simply achieved by using small and non-complicated cache structure as the first level of the cache. In this regard, the direct

CHAPTER 2. LITERATURE SURVEY

mapped cache offers a distinct advantage because the tag check in a direct-mapped cache can be overlapped with the transmission of the data [24].

Another technique that has been used to reduce the cache hit time in multitasking systems is the use of a virtual cache [24]. In a virtual cache, virtual addresses are used for indexing the cache. This does away with the need for virtual to physical address translation thereby decreasing the hit time of the cache. Although this offers a viable solution for improving the hit rate, there are challenges that need to be tackled.

One such difficulty in using a fully virtual addressed cache is that page-level protection is checked as part of the virtual to physical address translation and the check being mandatory, it must be performed. Another problem is that on a context switch, the same virtual address could point to different locations in physical memory because it belongs to two different processes. For circumventing this problem, the cache has to be flushed every time there is a context switch. This is wasteful as it may evict useful information from the cache which may be reused again when the task is executed again. Another solution that has been suggested for this problem is using PIDs or process identifiers in conjunction with the virtual addresses to identify cache blocks. When the PID for a virtual address is changed, it indicates that the line needs to be flushed. A problem of aliasing [37] also exists in virtually addressed caches, wherein two different virtual addresses of the operating system and the user task may map to the same physical address. A workaround to this problem is called anti-aliasing which guarantees a unique physical address for every cache block [37].

Trace caches [24] also reduce cache hit time effectively. Instead of limiting the instructions in a cache block to spatial locality by storing physically adjacent instructions only in a cache block, a trace cache finds a dynamic sequence of instructions including taken branches to load a cache block. The downside of a trace cache is that it may store the same instructions multiple times because conditional branches in programs may result in the same instructions being part of separate traces.

2.3.2 Increasing Hit Rate

The hit rate of an application can be increased using a number of techniques. The first among such techniques is using a larger block size. The advantage of using a larger block is that it exploits the concept of spatial locality meaning that data adjacent to the current reference have a high likelihood of being referenced as well. However, if the block sizes are increased beyond a certain limit, they reduce the amount of cache line utilization. Consequently, a large part of the cache line contains irrelevant data which increases the miss rate because relevant data may not be able to reside in the cache.

A powerful miss rate reduction strategy is to increase the associativity of the cache. Increasing the associativity decreases the number of conflict misses in the cache because a cache line brought in from the memory could reside in any block inside a set. This provides for a strategy to replace unused lines or less frequently used lines with the more useful ones. If multiple program lines map to the same cache line in a direct mapped cache, making the cache set-associative ensures a unique cache for all of those program lines by evicting other less frequently used cache lines. It has been shown in empirical studies that the miss-rate of a direct mapped cache of size N is equal to the miss rate of a 2-way set associative cache of size $N/2$ [24]. This is known as the 2:1 cache rule. As with large block sizes, increasing associativity beyond a certain point may not yield increasing returns. This fact has been observed in studies which have shown that the miss rate rarely drops significantly if the associativity is increased beyond 8 [24].

Set-associative cache memories achieve better hit rates than direct mapped caches but they have problems of their own. They are slower than direct mapped caches because of the more complicated cache line (or way as they are called in set-associative caches) selection logic. This bottleneck in set-associative cache memories can be circumvented by using way-prediction caches [26] wherein extra bits are maintained in the cache to predict the way in which the next instruction or data referenced could be found.

Another approach to reducing the miss rate is compiler assisted optimizations. Loop interchange [4] and blocking [70] are two common techniques used by compilers to reduce the number of misses when the application executes. While loop interchange improves the hit rate by reordering nested loops so that the data blocks brought into the cache can be fully utilized before being evicted. Blocking on the other hand tries to reduce misses by improving temporal locality. Block algorithms are usually used for matrix manipulations. Instead of operating on entire rows or entire columns of a matrix, blocked algorithms work on sub-matrices or blocks.

2.3.3 Reducing Miss Penalty

Using multilevel caches has been a widespread technique for reducing the miss penalty. The first level of the cache can be small enough to match the CPU clock speed while the second level of the cache can be large enough to hold many accesses and prevent them from going through to the main memory [24].

Whenever an access is made to a cache, if it is a cache miss, the entire line is brought into the cache from the memory and the requested word is forwarded to the processor. This is obviously wasteful because the processor has to wait longer than what it normally should. There are two kinds of optimizations suggested to deal with this problem. In early restart [24], as soon as the requested word is brought into the cache, it is forwarded to the processor. In another technique called critical word first [24], the requested word is brought from the memory first and forwarded to the processor. As the processor continues its operation, the rest of the line can be filled simultaneously.

Another commonly used technique is to give reads priority over writes. This means letting read misses continue by fetching data from the write buffer if it is present there instead of making the read miss wait until the write buffer is empty. In case of write-back caches, when a read miss leads to the replacement of a cache line that is dirty, the dirty

line could be copied to a write buffer instead of being copied to the memory and the read miss can proceed. This technique leads to considerable savings in miss penalty.

Some processors minimize the miss penalty by merging write buffers. In this technique known as write merging [24], the write buffer is checked for an address match to decide whether the address being written to already exists in the write buffer. If it does, then the old value is overwritten. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though they can fit in a single entry of the buffer because they are sequential. This optimization reduces stalls due to the write buffer being full.

The last optimization that we discuss here is the use of victim caches [27]. A victim cache is a small fully-associative cache between the cache and its refill path. The idea behind a victim cache is to reuse cache lines evicted from the cache at a small cost. The victim cache holds data that is discarded from the main cache recently because of a miss. If the data item is referenced again the victim cache is searched first before going to the memory for fetching the data.

2.4 Architecture Level Techniques for Power Reduction

Increasing the cache capacity often leads to increased hit rates. This is useful in conserving power because off-chip accesses could typically consume about 50 times the power required to access the on-chip cache [59]. However, there is a trade-off to this. Accessing a larger on-chip cache can itself increase the amount of energy spent to get the instruction. For example, on-chip cache for the Strong ARM SA110 consumes approximately 43% of the total chip power [38]. Therefore, reduction of energy expended in accessing cache memories has been an active area of research lately.

Energy dissipation in integrated circuits can be divided into two components - static and dynamic. Dynamic component of energy dissipation depends on the charging and discharging of capacitive gates and wires. This in turn is dependent on the amount of activity in the component. Static energy consumption on the other hand is due to the sub-threshold leakage of reverse biased junctions and depends on the process technology used to manufacture the component and various other factors. The interesting thing about static energy consumption is that it occurs even when there is no activity in the component as long as the supply voltage for the transistors are on. Static consumption of energy in integrated circuits is proportional to the number of transistors used.

In typical high end processors, cache memories account for a large portion of the total transistor count. Subsequently, they account for a large amount of both static and dynamic energy consumption. Therefore, there exists a strong need for power-sensitive cache hierarchies. In the next few subsections, we discuss the various techniques suggested in the literature for power-sensitive cache hierarchies.

2.4.1 Dynamic Energy Reduction Techniques

The equation which is commonly used for modeling the average energy spent per access to a single level cache hierarchy is as follows:

$$E_{AMAE} = E_{Hit} + M \times E_{Mem} \quad (\text{Eq. 2.2})$$

Where E_{AMAE} , E_{Hit} , M and E_{Mem} are the average memory access energy (the average amount of energy spent per access to the memory hierarchy), access energy when the instruction/data access is a hit in the cache, miss rate of the cache and energy spent to access the main memory respectively. In this work, we are interested in the cache memory hierarchy only. So, the techniques for reducing energy consumption in the main

memory are not discussed. Instead we discuss techniques suggested in the literature for reduction of power consumption in cache memories.

The dynamic power consumption in CMOS circuits is represented by the following equation:

$$E_{Dynamic} = 0.5 \times C \times V_{dd}^2 \quad (\text{Eq. 2.3})$$

In this equation, $E_{Dynamic}$, C , V_{dd} represent the dynamic energy consumption of the CMOS circuit, the total effective capacitive load and the supply voltage of the circuit respectively. For a constant miss rate, to reduce the cache access energy, the effective capacitance C has to be reduced.

2.4.1.1 Using Compressed Caches

Instructions and data in a reference stream usually exhibit a large amount of redundancy. The idea behind compressed cache memory is to compress the contents of the cache to exploit this redundancy. The compressed information can then be stored on a much smaller cache. The decompression must be done on the fly as data or instructions are referenced. This imposes performance requirements on the decompression algorithm. Also, instruction and data usually exhibit different kinds of redundancy. The compression schemes for data and instruction thus have to be different.

Compilers use syntax directed translation to compile high level languages. This creates redundancy in an instruction stream which could have same instruction blocks repeated at many places. Compression schemes targeted at using compressed cache memories usually break the program into blocks of size n - the cache line size - which is compressed individually. To facilitate the decompression process, a Line Address Table (LAT) is used to record the address of each compressed cache line within the compressed program [66]. A Cache Lookaside Buffer (CLB) is used to reduce the decompression overheads.

The compression scheme for data caches is slightly different. For data caches, 70% of the bits that are read from or written to are zeroes [58]. To reduce cache access energy, all zero values are represented by a single bit and accessed. Similar schemes like the sign compression and cache line bisection take advantage of the occurrence of both all zeroes and all ones.

Although using compressed caches results in a large saving in cache access energy, they have their own drawbacks. The additional hardware logic required for decompression scheme is significant. Also, the decompression scheme imposes significant performance penalty.

2.4.1.2 Partitioning the Cache Array

Cache array partitioning targets bit line energy dissipation to reduce the cache access energy. Bit line energy dissipation occurs due to switching of the cache line data and tag. Partitioning activates only a section of the entire cache array to service the processor's request. Cache partitioning is done using two techniques - horizontal and vertical partitioning [46].

In vertical partitioning techniques, the basic idea is to minimize the energy expended in accessing the cache by enabling access to one cache block at a time without activating the entire cache. This is often achieved by having a line buffer between the processor and the cache. This enables the processor to access adjacent memory locations from the line buffer which has a much lesser capacitance value as compared to the cache arrays. Cache sub-banking, on the other hand, is a horizontal partitioning scheme in which the data array is partitioned into sub-banks. Using the offset part of the address, sub-banking activates only the data item being referenced and not the entire data array. Therefore, it aids in considerable power savings. As far as performance is concerned, sub-banking does not impose any performance overheads while buffered cache access in the case of

vertical partitioning imposes performance overheads due to the additional level of cache hierarchy.

2.4.1.3 Behavioral Optimizations for Cache Accesses

For performance reasons, all the ways in a set-associative cache are searched in parallel. However, this is wasteful from the energy perspective because a lot of cache lines are unnecessarily enabled. Vertical partitioning somewhat mitigates this energy overhead by adopting a line-buffered scheme for accessing the cache. However, the additional level of cache hierarchy imposes performance and hardware area overheads. To circumvent this shortcoming of vertical partitioning schemes, way-prediction caches and way-memorization cache memories can be used. In way-prediction schemes, a cache-way is speculatively selected based on most-recently-used (MRU) information [26]. However, this also imposes performance overheads in the event that the prediction is wrong. Luckily, empirical studies have shown that for commonly used applications, the prediction is correct for 85 % of the times[11]. Other techniques for behavioral optimization of the cache have suggested doing away with tag comparisons for every access to the cache. Tag comparison is sometimes wasteful because if two consecutive instructions belong to the same cache line, the tag comparison can be avoided. This fact is used by way memorization schemes [41].

Another technique that has been proposed for reducing the energy expended in accessing the cache is to perform tag comparison and data access in series so that unnecessary data array accesses are avoided [23]. Such cache memories are known as Phased Cache. While such a scheme decreases the energy expended in accessing the cache, the sequential access imposes a performance penalty on every access.

2.4.1.4 Auxiliary Cache Based Memory Hierarchy

Another approach to reduce the average cache access energy is to use an unusually small cache (typically less than 1 kB) in conjunction with the L1 cache. Such auxiliary cache structures proposed for use in the instruction cache hierarchy exploit the spatial and temporal locality in small loops within applications. The underlying idea behind energy conservation using this kind of a scheme is that access energy from a smaller cache is much lower than that of a larger cache. Line buffer [24], loop cache [7][33] and filter cache [29] are examples of these kind of arrangements. While such arrangements are typically for instruction cache memories, scratch pad memory [6] has been proposed for the data cache.

In a loop cache, the presence of a loop is detected by short backward branches (*sbb*). If a short backward branch is detected, it is assumed to be the backward branch of a loop and the loop cache is filled up unobtrusively with instructions of the loop during the next iteration. For the subsequent iterations, references are directed to the loop cache. The limitation of a loop cache which is filled up dynamically is that it assumes a straight line execution for the loop with the absence of any branch instructions. In reality however, this is not always the case and a preloaded loop cache was discussed in [20] to deal with loops that contain branches.

A filter cache is another viable method for saving energy expended in the cache hierarchy. It is a tiny cache (less than 1kB) which sits between the processor core and the L1 cache. It has been shown to achieve a significant reduction in expended energy at the cost of a minimal performance overhead [29]. The performance overhead is largely due to the hierarchical nature of the filter cache hierarchy. To circumvent this, a predictor based filter cache hierarchy has been proposed as well [52][51][59]. The predictor has the capability to direct the access to the L1 cache if it thinks the line is missing in the filter

cache. Predictors have been shown to be highly accurate [59] but their disadvantage is that they occupy extra hardware area.

2.4.2 Static Energy Reduction Techniques

Lately with deep sub-micron process technologies, static energy dissipation has taken a front seat in power-sensitive designs. To meet this challenge of leakage energy dissipation, numerous techniques have been suggested. One such technique is gated-Vdd wherein portions of the cache are shut down dynamically [42]. Another such technique is called cache decay [28]. It works on the assumption that when a cache line is brought in, it will be used frequently over the next few accesses and if it has not been used, it is probably not needed anymore. In such a case, the cache line is turned off. For both gated-Vdd and cache decay, hardware logic overheads are significant. Another added consideration in these schemes is that the gain in energy savings should be reasonable enough to justify the additional accesses to lower levels of the memory in case a block is turned off.

Another set of techniques achieves energy savings by putting the cache lines into a **drowsy** mode which is a low power state [14]. There are different schemes employed for instruction and data cache for putting the cache lines into drowsy state. For the data cache, cache blocks are periodically put into a drowsy state regardless of their access pattern. Whereas for instruction cache, the entire cache is divided into sub-banks with only one sub-bank being active at any given time. The disadvantage of such an arrangement is again the energy overhead for re-fetching a drowsy block from lower memory levels.

2.5 Architectural-level techniques for low power and high performance

In the preceeding sections, we described different solutions proposed in the literature for low power cache memories as well as solutions for high performance cache memories.

Using a smaller cache may result in lower energy expended per access to the cache memory but may result in additional miss rates and a performance penalty may be incurred. For most cache optimizations, performance and power are indeed inversely related. To circumvent this, solutions have been proposed in the literature that use predictors to direct accesses to smaller cache memories when the working set of the program is small and to a larger cache otherwise. In this section, we discuss predictor based cache hierarchies with a focus on the filter cache. As discussed earlier, the filter cache is a tiny auxiliary cache (usually less than 1kB) structure which sits between the processor and the L1 cache. Embedded applications typically spend a significant amount of time in small loops which makes the filter cache an ideal energy saving initiative option for embedded system because accessing data from a small cache is definitely more energy efficient than accessing it from a larger cache. However, there is a downside to this arrangement. The filter cache structure imposes performance overhead by introducing another level of cache memory between the processor and the L1 cache. Since not all the loops can be contained in the filter cache, there is a significant miss penalty every time anything other than a small loop is executed because of the small size of the filter cache. This is demonstrated with the fact that although the filter cache is able to achieve a 58 % energy savings for the MediaBench benchmark suite, it also makes the cache hierarchy 21% slower [59]. To deal with this problem, predictor based filter cache hierarchy was introduced which is discussed in the next section.

2.5.1 Predictor based filter cache hierarchy

In the predictor based filter cache hierarchy, predictive parallel path exists from the processor core to the filter cache and L1 cache as shown in figure 2.1. Since this arrangement was originally proposed and validated for the instruction cache, we discuss the predictor-based cache hierarchy from the perspective of instruction references. Due

Figure 2.1: Predictor based filter cache

to the presence of the predictor, there can be four operating cases for every access to the predictor-based filter cache hierarchy for instructions. Each of these cases have different energy and performance trade-offs.

- (i) Predicted Location:FC, Actual Location:FC - In this case, there will be no performance penalty and we will gain substantially in terms of energy. The instruction can be fetched in a single cycle.
- (ii) Predicted Location:FC, Actual Location:L1 - Since the instruction was predicted to be in the filter cache but was not found there, there will be energy overheads as well as performance penalty of one cycle as compared to the case when the filter cache was not there. This case involves a false access to the filter cache. Therefore, a design priority of the predictor is to prevent such wrong predictions.
- (iii) Predicted Location:L1, Actual Location:FC - The predictor directed the instruction access to the L1 when the instruction requested is already present in the filter cache (the filter cache is an inclusive cache so such a case is entirely possible). In this case, the potential savings in energy and access time due to getting the instruction

from the filter cache is lost due to inaccuracy of the predictor. The access time is one cycle in this case. This is another case where the potential savings may be lost due to the inaccuracy of the predictor.

- (iv) Predicted Location:L1, Actual Location:L1 - The access time in this case is one cycle as well. The access energy in this case will be lesser than that of a conventional filter cache hierarchy which does not have a predictor as the access need not be directed to the filter cache first and then on a miss, the requisite instruction fetched from the L1 cache. There will be not performance penalty in this case.

2.5.2 Prediction algorithms

As evident from the preceding discussion, a good prediction scheme can result in significant energy savings. In the next few sub-sections, we discuss some of the prediction algorithms that have been proposed for the predictor based instruction cache hierarchy.

2.5.2.1 NFPT Prediction Algorithm

The next fetch prediction table (NFPT) algorithm was proposed in [51] and [52]. It was shown to limit the performance degradation to only 1.5 % while providing energy savings of about 31.5 % for the spec95 benchmarks [51]. The results were reported for a wide issue superscalar processor whose instruction fetch width was equal to the instruction cache line size (16 bytes). For the NFPT predictor, the prediction process for the next cycle begins in the current cycle wherein the locality within small loops is utilized to predict whether the next access will be from the filter cache. The predictor relies on the next fetch entry in the NFPT for making this prediction. The next-fetch entries in the NFPT are updated based on control paths taken in the past. If the current instruction and the next-fetch address entry have the same tag, then the instruction is predicted to be in the filter cache otherwise the L1 cache is accessed.

The NFPT prediction algorithm was modified to suit single issue processors as well. For the prediction purposes, the logic that was employed was that if the current fetch address is PC , the most likely next fetch address would be $PC + 4$. Therefore, if PC and $PC + 4$ map to the same cache line then the next fetch would be from the filter cache otherwise the NFPT is referenced to determine which cache to access when the next instruction is executed.

There are some problems inherent in the NFPT predictor though. If there is a function call inside a small loop, the predictor does not direct accesses to the filter cache even though the small loop can be fully contained in the filter cache. This happens because the predictor uses tag information to direct accesses to the L1 cache or the filter cache. On a function call, assuming that the compiler does not compile the called function inline, there will be a tag change and the subsequent access (even those instructions belonging to the small loop) would be directed to the L1 cache. To overcome this shortcoming of the NFPT prediction algorithm, the Pattern Prediction Algorithm was used in [61]. This algorithm is discussed next.

2.5.2.2 Pattern Prediction Algorithm

The pattern prediction algorithm uses two levels of cache access history information to make meaningful predictions on where the next instruction can be found. The first level is the history of the last N cache line changes encountered wherein it records the outcome of the prediction like in the normal branch prediction schemes [24]. This information is stored in a shift register (of length N) for the pattern prediction algorithm. If the shift register is N bits, the number of entries in the pattern table is 2^N . The next level records the hit/miss behaviour of these patterns for the last k times these patterns were encountered. This information about hit/miss behaviour is stored in a pattern table and is captured in saturating counters for each entry in the pattern table. The value of

each entry is decided by the value of a 2-bit saturating counter which is updated with hit/miss result of the prediction whenever the pattern table is referenced for prediction purposes[59]. When the pattern table is accessed, it is indexed using the shift register contents. Then the value of the saturating counter for that entry is read and depending on this value a decision is taken on whether the filter cache will be accessed or the L1 cache will be accessed.

Like the NFPT predictor algorithm, the predictor in this case is not invoked on every instruction reference. Instead, it is invoked only when cache line changes are encountered. If the current instruction (PC) and the next instruction (which is assumed to be $PC + 4$) map to the same cache line then the instruction is accessed from the filter cache. Otherwise, the predictor is invoked to predict where the instruction could potentially be found and subsequently, the shift register and the pattern history table are updated with the hit/miss result.

2.5.2.3 Static Pattern Prediction Algorithm

Although the pattern predictor solves the problems of the NFPT predictor, it still requires a large area to be implemented due to the counters and the update logic. To tackle this, the static pattern prediction was suggested in [63]. In this predictor, instead of using saturating counters to update the entries in the pattern table, the entries are preset to 0 or 1. The concept behind this optimization is explained below.

It was shown in [59] that the shift register values in the pattern prediction scheme for a given application usually shows an inclination toward either all 0s or all 1s or a series of 0s followed by a series of 1s. It rarely shows values where 0s are randomly interspersed with 1s. This happens because if a loop is fully cache resident, it will exhibit cache hits consistently (represented by a series of 1s). If the loop is partially cache resident because of either inadequate cache capacity or cache line conflicts, then a small part of it will

still be cache-resident. That small part would be responsible for series of hits and then there will be a series of misses of instruction lines because of the non-cache resident part of the loop. In such a case, we will have a pattern which will be a series of 1s followed by a series of 0s and so on. Finally, if the loop is so large that no fraction of it can be cache resident across iterations then we will always have instruction line misses.

Considering this fact, it is possible to assign default values to the pattern table for different patterns of the shift register. The static pattern predictor trades off a little bit of accuracy though it is still more accurate than the NFPT predictor [59]. However, its main advantage lies in significantly reducing the hardware area required by the prediction logic because we do away with the saturating counter and the logic for updating them.

2.6 Application-specific customization of cache memories

Cache requirements of applications vary widely [74]. Some may need a larger cache for the best performance while others may need to save on energy and it may be reasonable to use a smaller cache for such applications. When it comes to block sizes too, the amount of spatial locality may be different in different applications and using a one-size-fits-all approach may not be ideal. Tuning cache parameters in an application specific manner has been shown to aid in considerable savings in energy and improvement in performance [57][72]. With these requirements in mind, configurable or tunable cache memories were proposed to tweak cache parameters in an application specific manner. In this section, we discuss tunable cache memories and also present techniques proposed in the literature for application-specific cache tuning.

2.6.1 Tunable cache memories

The concept of configurable or tunable cache memories has been prevalent in both softcore and hardcore processors. In softcore processors like Microblaze and Nios II [5][69][55],

the cache configuration is chosen at synthesis-time. On the other hand, in hard-core processors [3][34][74] the cache is configured at run-time. For soft-core processors, the user usually chooses a cache configuration before synthesis and generates a customized cache. In hard-core processors, the desired cache configuration can be obtained by initializing registers on the chip that control multiplexers and the cache controllers using software code at run-time.

Techniques have also been proposed in the literature wherein the cache adjusts its parameters automatically to the executing application. This is achieved by observing special counters at run-time at regular intervals and efficiently exploring the configuration space to select the best cache configuration [72][21]. Such an approach is known as the self-tuning approach.

While self-tuning is a neat method for cache tuning, the infrastructure for the special counters required in this method may not be always available. Even if the infrastructure is available, the overheads associated the control circuit may defeat the purpose of cache tuning. These problems are addressed by one-shot software-reconfigurable tuning wherein optimal cache parameters for the applications are determined statically and instructions are added to the application to tune the cache at run-time.

2.6.2 Design space exploration for cache tuning

Cache tuning is usually achieved by obtaining cache hit rates for a variety of cache configurations and then identifying the optimal cache configuration using user-defined tuning heuristics as shown in figure 2.2. The first step in this process is to obtain the hit/miss rates for a variety of cache configurations. In the next step, the multitude of cache configurations for which the hit rate has been measured using cache memory simulation or estimated using profiling is fed to a cache energy-delay model database like CACTI [65]. CACTI is able to estimate the energy per cache access and the delay

Figure 2.2: Analytical cache tuning techniques for the filter cache hierarchy

per cache access for the input cache configurations. The hit rates, per access energy and delay values are then used to identify an optimal cache configuration using a set of user defined heuristics. A common heuristic searches for a cache configuration that results in the lowest energy consumption [21][60][74].

While the heuristics for selection of an optimal cache is based on system constraints and is left for the system designer to specify, there are many techniques to measure or estimate cache hit/miss rates for multiple cache configurations. The techniques for measuring the hit rates rely primarily on cache simulation of the desired cache configuration by using the program trace as an input for simulation. On the other hand, techniques to estimate the cache hit rates are based on analyzing the locality in the programs either with the help of a compiler or through a profiler. The profiler is fed with runtime information about the execution of the program and identifies potential zones of locality of reference in the application. In this sub-section, we discuss the various options available for design space exploration to identify optimal cache parameters for an application.

2.6.2.1 Design space exploration using exhaustive cache simulation

In exhaustive techniques for cache tuning, simulation based search methods are used to find optimal cache configurations. In such techniques, multiple cache configurations are simulated to identify the optimal cache configuration. The number of combinations that need to be searched in this approach is very large. This entails simulating the application multiple times and can be prohibitively time consuming even for small applications. Such brute-force search techniques are ineffective in projects with time-to-market pressures.

Therefore, a significant amount of work has focussed on reducing the number of cache configurations that need to be examined to arrive at an optimal configuration. Several tools do exist for assisting designers in tuning a single level of cache. Platune[18] is a framework for tuning configurable system-on-a-chip (SOC) platforms. Platune prunes the search space by isolating interdependent parameters from independent parameters. However, cache parameters like cache size, block size, associativity, etc being inter-dependent, are explored exhaustively which makes this approach prohibitively time consuming.

Heuristic-based methods were proposed to prune the search space of the configurable cache. A genetic algorithm was proposed in [40] to improve upon the exhaustive search used in Platune to produce comparable results in less time. Another technique was presented in [] wherein a cache configuration exploration methodology wherein a cache exploration component searches configurations in order of their impact on energy, which reduces the number of combinations that need to be evaluated. Iterative heuristics were proposed in [71] so that near-optimal cache configuration is reached without actually simulating the entire configuration.

2.6.2.2 Design space exploration using static control flow graphs

Optimizing compilers have long relied on analysis of program locality to detect data reuse and estimate cache misses. The analysis of data dependencies and reference patterns has been used in a variety of loop transformations to improve locality of reference

in the program and reduce accesses to the main memory. Researchers have worked on numerous such re-ordering or transformation techniques to improve the locality of reference in applications like loop interchange [16][36], loop fusion/fission [36] and loop tiling [10][13][31].

In [36], a cost model was proposed to find desirable loop organizations by computing both the spatial and temporal reuse of a cache line. The cost model was shown to be highly effective in deriving loop structures that result in the fewest number of accesses to the main memory. A technique to estimate conflict misses by linearizing array references and then evaluating the amount of cache conflict resulting from memory references was presented in [44]. Cache miss equations were presented in [17] to describe the cache performance of applications analytically. In this technique, deciding whether a reference causes a hit or miss for a given iteration of a loop was equivalent to deciding whether it belonged to a polyhedra defined by the cache miss equations.

Design space exploration using control flow graphs has low overheads because it can be done at compile time without needing to run the application. However, it is insensitive to loops whose boundary is known only at runtime and therefore, offers inaccurate measures of locality for applications that contain such loops.

2.6.2.3 Design space exploration using runtime trace

Design space exploration techniques that have relied on the runtime trace generally focus on extracting features from the trace that quantify the level of locality in the application. The techniques proposed in the literature rely on these extracted features to predict the hit/miss rates for a range of cache configurations. One of the earliest of such techniques was proposed in [12]. In that work, it was proposed that the miss rate of the cache size of C could be modeled by the equation $MR(C) = C^{-\alpha}$. The parameter α could be found based on the available program trace. Based on the empirical model in [12], a fractal

CHAPTER 2. LITERATURE SURVEY

model of locality was also proposed [56]. The argument put forth in [56] was that the sequence of memory references represented a random walk through memory. The jumps from one region of the memory to another are often short and occasionally long. It was shown that such a random walk is a special case of fractals and therefore, existing work in fractal theory can be relied upon to model the locality of the trace. The miss rate for a cache of size C is modeled as $MR(C) = AC^\theta$ in this work. The different values for A and θ can be obtained using existing data. These models were effective in shedding some light on the nature of locality of the program. However, they focussed only on the cache size and were not detailed enough to observe the effect of changing the block size and associativity on the hit rate. Moreover, for the fractal model of locality, while the instruction stream showed characteristics of a random walk necessary for a fractal model, most of the data stream lacked this.

A detailed model for modeling temporal and spatial locality of the reference stream after breaking the reference stream into time granules was proposed in [1]. Apart from considering the cache size, the model also takes into account the block size, associativity and cold start effects. To estimate the miss rate, this model has four locality parameters. The first parameter is the number of unique references that are made in a time granule. The second parameter is the collision rate of different cache blocks in the cache. The last two parameters are inputs to a two-stage markov model for analysing the effect of block sizes on the collision rate. The collision rate is derived from the number of unique references and the inputs to the markov model using an expression. These input values to the model are measured from the trace of the application. While this technique offered a feasible way to model a variety of parameters affecting the miss rate, the average error rate in miss rate estimation occasionally reached as high as 23 % for certain benchmarks and configurations. Measurement of locality based on reuse distance, which is discussed subsequently, offered a solution to this problem by reducing the maximum error rate to about 5 to 10 %.

CHAPTER 2. LITERATURE SURVEY

The analysis of program locality based on reuse-distance analysis was initially proposed in [35]. In this technique, the number of distinct data accesses are counted between two accesses to the same data element. After obtaining information about the reuse distance for the data elements, the whole program can be represented as histogram describing reuse distance distribution, where each interval showing the portion of memory references which falls into the same reuse distance range. This reuse distance information can then be used to estimate the hit/miss rate for multiple cache sizes. While the reuse-distance analysis offered a feasible technique to explore multiple cache sizes and offered fairly accurate results when compared with other techniques, there were issues with its scalability. The time complexity of a naive stack distance algorithm is in the order of $O(NM)$ where N is the length of the trace and M is the size of the program data. The space complexity of the distance analysis data structure is $O(M)$ in this case. Indeed, this was the complexity reported in early research work in this area [35]. As the size of programs increased, this complexity was found unacceptable, therefore further research was done to deal with this and an m-ary tree built on a vector was used for analysis in [8]. The same work also showed how to use blocked-hashing in a pre-pass. The first tree-based organization of the data access trace using an AVL tree was discussed in [39]. In the same work, techniques were also proposed to compress the trace vector used in [8]. In [48], it was shown that a splay tree has better performance than an AVL tree. Vector based techniques reduce the time complexity to $O(N\log N)$ but the space complexity remained very high at $O(N)$. Tree-based techniques reduced the time complexity to $O(N\log M)$ with space complexity being $O(M)$.

Relying on the program trace to determine a suitable cache configuration was feasible as long as the program traces were small. The ever-increasing complexity of modern-day applications forced researchers to investigate on efficient program trace collections techniques. Some techniques were also reported to reduce the trace generated by the

simulated application and/or simulate multiple configurations in one go [25] [47]. To improve the simulation speed, techniques were formulated to find approximate trace [30] or lossless trace reduction [43] [64] [68].

2.6.2.4 Design space exploration using dynamic control flow graphs

Lately, cache tuning based on loop profiling has been shown to achieve significant speed-up in the tuning process [62][59]. It relies on tuning the cache based on results from a one-time loop profiling of the weighted or dynamic control flow graph (DCFG) of the application. DCFGs are edge-weighted versions of the static control flow graph. The DCFG of an application can be obtained by instrumenting the application with counters so that the frequency of a branch can be obtained. Alternatively, a weighted control flow graph can also be obtained by observing the trace of the application and then deducing the frequency of the edges.

Compared to the static control flow graph based design space exploration, the dynamic control flow graph approach is more accurate as it contains runtime information. The runtime information is useful in identifying the frequent code segments like loops whose boundaries are known only at runtime. Identification of frequent code segments is essential to calculating the working set size for the program and then selecting a suitable cache for the application.

Compared to stack distance based techniques, the dynamic control flow graph offers a faster and scalable solution. Building, maintaining and analyzing a dynamic control flow graph is simpler than computing locality based on stack distance probabilities. Firstly, because the DCFG is at the basic block level, it is simpler to obtain. When the edge weights are computed, it is just a simple increment operation. For example, when a basic block is executed, only a counter needs to be incremented to keep track of the execution frequencies. However, for the stack distance method, the stack has to be searched and the

probabilities updated for every reference made. This is more involved than building the DCFG. On the flip side, due to a higher level of abstraction, the DCFG based technique is not as accurate as stack distance based measures.

There are three steps in using DCFGs to obtain estimates for suitable cache sizes for the application. First the application is simulated in an instruction set simulator or the actual platform if available. After simulation, the DCFG is available for further processing. Loop exploration algorithms are then applied to obtain the frequent code segments in the application. In the final step, the relative frequency of these loops and their size is taken into account in obtaining an estimate for a suitable cache size for the application. The heuristics used for selecting the cache size are usually defined by the designer and usually measures for energy spent per access to the cache are used to identify an optimal cache size.

2.7 Summary

With the rising complexity of applications, performance improvement was the original focus of architectural-level cache optimizations. However, the advent of mobile and handheld devices triggered research into architectural-level optimizations for low-power cache memories. In this chapter, we discussed numerous architectural-level optimizations proposed for the cache hierarchy to improve its performance and decrease its energy consumption. We further discussed how optimizations to reduce energy expended in cache accesses often led to a deterioration in system performance and presented optimizations like the predictor-based cache hierarchy aimed at decreasing cache access energy without significantly impacting performance .

Apart from architectural-level optimizations, we discussed how application-specific cache customization results in considerable improvements in energy expended in the cache hierarchy as well. In this context, we discussed tunable cache architectures where

CHAPTER 2. LITERATURE SURVEY

the cache parameters can be tuned in an application-specific manner to save on energy. While tunable cache architectures offer a viable way of further reducing energy expended in the cache hierarchy, identification of optimal cache parameters for tuning purposes is a difficult task and an area of active research. Therefore, we presented analytical and exhaustive techniques proposed in the literature aimed at identifying optimal cache parameters for a given application as well.

While profile-driven cache tuning is an attractive option for tuning the cache to suit the needs of the application, it suffers from some inherent shortcomings. Firstly, it fails to identify all the loops in the application due to the choice of a loop profiling algorithm which cannot identify irreducible loops (loops with multiple entry points). Secondly, it does not take into account the cascading nature of function calls and sparsely populated cache lines when calculating loop sizes. This leads to misappropriation of the nature of the loops in the application which affects the cache tuning process. Also, existing techniques do not address how to tune the cache for multitasking applications. They do not describe how can the cache be re-tuned for an application if the underlying RTOS and target platform combination changes. In this research work, we seek to investigate and develop fast, scalable and generic techniques for tuning the cache for both single-task based and RTOS-driven multitasking embedded applications. In order to do so, we first examine existing techniques for cache tuning for single-task based applications in detail in order to address any shortcomings in them.

Chapter 3

Cache tuning for single-task based embedded applications

3.1 Introduction

Traditionally microcontrollers and microprocessors vendors have relied on a one-size-fits-all approach when it comes to cache memories. However, cache requirements of applications vary widely and such an approach has serious consequences when it comes to the energy expended in cache memories. For an application, using too small a cache results in increased number of cache misses and consequently, the main memory needs to be accessed frequently. This is wasteful and increases the average cache access energy considerable. On the other hand, using a large cache may result in fewer number of cache misses but the energy spent in accessing a large cache is higher as compared to cache memories of smaller size. Therefore, tuning the cache to the needs of the application can result in a significant reduction in the energy expended in cache accesses. In wake of this, tunable cache memories have been proposed that enable tuning the cache parameters to suit the needs of the application [74]. They have been shown to substantially reducing the energy and delay overheads associated with using a non-tunable or static cache normally available in commercial microcontrollers and microprocessors [60][62].

Although tunable cache memories offered an efficient way of reducing the energy and

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

delay incurred in running the application, identification of optimal cache parameters remained a difficult task. Various techniques were proposed to identify these parameters for any given application. These techniques are discussed in detail in chapter 2. Prominent among these techniques were ways to tune a cache based on the loop profile of an application. The loop characteristics of an application could be obtained after a one-time simulation and offered a fast and reasonably accurate alternative to identifying optimal cache parameters using exhaustive simulation. However, the proposed techniques for loop-profile driven cache tuning have some inherent shortcomings. In this chapter, we address these shortcomings and propose a fast, lightweight, accurate and highly scalable technique to tune the cache in an application specific manner. We apply the proposed techniques to tune the predictor based filter cache hierarchy for instructions - a common solution for power-sensitive embedded systems. We also discuss how the proposed techniques can be tweaked to include exploration for optimal cache block sizes and the level of associativity as well.

The rest of the chapter is organized as follows. In section 3.2, we define some of the terms that we use in our estimation framework. We discuss the shortcomings in existing profile-driven schemes for cache tuning in section 3.3. An improved approach to representing the application and subsequent loop profiling using this representation is discussed in section 3.4. We present expressions for hit rate estimation from the loop profile of the application in section 3.5. We discuss the predictor based filter cache hierarchy and propose heuristics for energy-centric cache tuning of the hierarchy in section 3.6. The overview of the cache tuning framework for tuning the predictor based instruction cache hierarchy is discussed in section 3.7. Our results for cache size tuning using the proposed techniques are presented in section 3.8. We discuss how the proposed techniques can be tweaked to include tuning methodologies for the block size and associativity in section 3.9 and conclude the chapter in section 3.10.

3.2 A discussion on locality of reference

The concept of locality in computer science pertains to the same storage locations being frequently accessed during the execution of a program. Temporal locality is a concept that a memory location referenced at one point in time will be referenced again in the near future. Spatial locality on the other hand is a concept that if a memory location is referenced, it is very likely that nearby memory locations will be referenced in the future.

For our modeling purposes, we model locality at the cache line level and define some terms used in our estimation framework below.

- **Cache Line Utilization:** This is the percentage of instructions executed from each cache line. Cache accesses by the application are seldom aligned with cache line borders. The flow of control may jump to a memory reference stored in another cache line before executing all the instructions in the current line. Therefore, when a line is brought into the cache, not all the instructions in it may be executed. There can be potentially unused instructions in every cache line fetched from the memory. So, the ratio of the number of instruction actually executed to the number of instructions brought in from the memory for a filling a cache line expressed as a percentage forms the Cache Line Utilization.
- **Spatial Locality:** Spatial locality is defined at the level of the cache block. It is the number of additional instructions (beyond the first instruction executed from the cache line) executed from the same cache line after it is brought into the cache from the memory. For example, for a cache line with 32 bytes block size, 4 bytes per instruction and a 50% cache line utilization, there would be 4 useful instructions in the line. When we access the first instruction it might be a miss but the subsequent instructions from the same line will definitely be hits due to spatial locality as

we have defined it. Assuming that the rest of the 3 instructions are accessed subsequently, we have a spatial hit rate of $3/4$ or 75 %.

- **Temporal Locality:** Spatial Locality accounts for most of the hits when accessing a cache line. However, it sheds no light on whether the first instruction in the line will be a hit or a miss. This depends on whether the line that is currently being referenced is cache resident or has been evicted from the cache. If program is currently executing in a loop that can be contained in the cache, the first instruction will be a hit as well when the loop iterates because that particular cache line can be retained in the cache. If we know the size and frequency of the loop and the size of the cache, we can estimate the number of possible hits of the very first useful instruction in a cache line. This is how we define temporal locality at the cache block level. Temporal locality as we measure it is therefore a function of the cache size and the nature of the loops in the application.

3.3 Drawbacks of existing profile-driven cache tuning techniques

Using the loop characteristics of the application to estimate the optimal cache size i.e. the one that gives the lowest energy delay product was proposed in [62][59]. The techniques offered the unique advantage of arriving at the optimal cache size after a one-time simulation without the need for exhaustive simulation of the cache hierarchy. However, there were some inherent shortcomings in the proposed approach which are discussed in this section.

3.3.1 Limiting loop exploration to reducible loops

Existing techniques used Tarjan’s algorithm for finding loops in the application [53][54]. Tarjan’s algorithm relies on depth first search for finding loops in the application. This

3.1.a: Correct loop identification -
 {Code Seg, BB5, BB4, BB3}

3.1.b: Incorrect loop identification -
 {Code Seg, BB2, BB3, BB4}

Figure 3.1: Different DFS walks can identify different loops in the same CFG

technique is good for finding simple reducible loops (loops with a single entry point into the loop body) in the application. However, the algorithm fails when there are irreducible loops (multiple entry points to the loop body) in the application.

This subtle shortcoming becomes obvious in figure 3.1. In this diagram, BBx stands for basic block x. BB1 to BB5 are the basic blocks in any given function. Basic block 1 (BB1) and basic block 4 (BB4) call the same code segment - which could be another function for example. In reality, BB3, BB4 and BB5 (and the code segment called by BB4) are part of a loop as is evident by the backward branch from BB5 to BB3. However, due to the return path from the code segment, the depth first walk can go in two different directions as shown in the depth first numbers in the two control flow graphs. This can lead to the identification of two different loops - code seg, BB5, BB4, BB3 and Code Seg,

BB2, BB3, BB4.

A significant portion of irreducible loops occur due to the same function call happening from basic blocks within loops and basic blocks outside loops. For example, in figure 3.1, the code segment that is called twice (once from BB1 and again from BB4) is invariably a function call eg. `printf`, `scanf`, `fread`, etc. Therefore, return paths from function calls should be ignored in loop profiling because they give rise to irreducible loops as shown in figure 3.1 and function calls should be abstracted out of loop bodies for an accurate representation of the loops.

3.3.2 Oversight of function calls in calculation of loop sizes

The existing techniques do not elaborate on the methodology for calculation of loop sizes that call functions. This case deserves special attention because if a loop is calling a function, calculation of the loop size becomes trickier. This is because the called function in turn may be calling other functions and so on. Consider figure 3.2 as an example. In this example, the loop in function calls function B directly. Function B in turn calls function D. Therefore it becomes necessary to include the sizes for both function B and D in the calculating the loop size. Existing loop profiling techniques for cache tuning fail to address this problem in calculation of effective sizes for loops that call functions.

3.3.3 Insensitivity to sparsely populated cache lines

While calculating loop sizes and the amount of locality in a program, existing techniques ignore the fact that all the instructions in a cache line may not be executed. For example, a basic block may be smaller than a cache line and the control flow may reference an instruction from another cache line before executing all the instructions from the same cache line. In other words, existing techniques have assumed a perfect alignment of basic blocks with cache line boundaries or a cent-percent cache line utilization.

We measured the line utilization ratios for lines sizes of 16B and 32B for some benchmark programs from the Mediabench [32] and MiBench [22] benchmark suite by running them on an instruction set simulator. As demonstrated in table 3.1, the line utilization is clearly short of the line utilization assumed by existing techniques ($ALUR$). The difference between LUR and $ALUR$ is as ERR or error in assumed values.

Benchmarks	16B block size			32B block size		
	LUR	ALUR	ERR	LUR	ALUR	ERR
adpcm.dec	0.93	1.00	0.07	0.83	1.00	0.17
adpcm.enc	0.93	1.00	0.07	0.83	1.00	0.17
bitcount	0.74	1.00	0.26	0.54	1.00	0.46
dijkstra	0.71	1.00	0.29	0.80	1.00	0.20
patricia	0.76	1.00	0.24	0.63	1.00	0.37
qsort	0.76	1.00	0.24	0.57	1.00	0.43
rijndael.enc	0.89	1.00	0.11	0.74	1.00	0.26
rijndael.dec	0.86	1.00	0.14	0.83	1.00	0.17
sha	0.96	1.00	0.04	0.89	1.00	0.11

Table 3.1: Actual vs. assumed line utilization for different benchmarks

These results show some instructions in a cache line may not be executed at all before the control flow shifts to another cache line. This may lead to a misrepresentation of the number of cache lines that the working set of the program will actually span because basic block boundaries may not be perfectly aligned with cache block boundaries. Consequently, it will lead to an inaccurate estimation of the hit rate and energy delay product that can be achieved for a given cache size and a line size assuming an ideal level of associativity.

3.4 An improved approach to cache tuning

In this section, we present an improved and simplified approach to loop profiling for cache tuning purposes. First, we discuss the techniques used to represent the application for the purpose of loop profiling. Then, we discuss techniques to identify loops and calculate loop sizes using this representation.

Figure 3.2: Representation of the application using call graph and control flow graphs

3.4.1 Representing the application for loop profiling

To deal with the issues in existing techniques for loop profiling and subsequent cache tuning, we introduce levels of abstraction in the representation of the application. Borrowing a concept from optimizing compilers [2][15], we represent the control flow in the application as a combination of a inter-functional call graph and intra-functional control flow graph. The call graph gives a snapshot of the control flow dependencies between functions while the control flow graph shows the control flow dependencies between basic blocks inside each function as shown in figure 3.2. While there is only one call graph for the entire application, there is a control flow graph for each function in the application. For the case of function calls, we disregard any return paths because they may lead to the formation of irreducible loops as has been dicussed in the preceeding section.

Representing the application in the aforementioned manner greatly simplifies loop ex-

Figure 3.3: Bottom-up call graph processing

ploration by limiting it to the body of a single function. Using a call graph for representing inter-functional control flow dependencies enables us to identify all the functions that could potentially be invoked directly or indirectly if any given function is called. This is especially useful in calculating the size of loops that call functions. Using such a representation, we can find all the functions that a loop can directly or indirectly invoke. This information can be obtained from first examining the control flow graph of a function that contains the loop to determine what functions are directly called by it. Then, the call graph can be relied upon to determine what functions can in turn be called by those functions invoked directly by the loop.

3.4.2 Methodology for extracting loop characteristics

The first stage in calculating loop sizes is to calculate the effective size of the function set directly or indirectly invoked by it. This information can be obtained from the call graph of the application.

For this purpose, we do a bottom-up processing of the call graph to determine the effective function sizes as shown in figure 3.3. The bottom-up processing ensures that the size of a called function is calculated prior to the calculating the size of the calling function and the size of the called function can just be added to the size of the calling function while calculating the working set size of an invocation of the latter. In the

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

bottom-up processing, we calculate the sizes of leaf functions first. They do not call any other functions and their size is the easiest to calculate. Then, we move up level by level until we reach the top of the call graph. To achieve this process, we put the functions in a call stack in that the functions that are called the last are at the top of the stack as shown in figure 3.3. Then we pop each function and calculate its size as described above.

The function sizes are calculated by adding up a frequency weighted sum of all the basic blocks in it. If the function calls another function directly or indirectly, we add the frequency weighted size of those functions too. The process of calculating the size of any given function (the expected number of distinct instructions executed per invocation of the function) using its control flow graph is shown in figure 3.4.

In the first stage in calculating the function size, we calculate the size of the basic block by counting the number instructions in it. If the basic blocks invokes a function, we add the working set of the function call to the basic block as well. The working set of the function call is the relative frequency weighted sum of the sizes of all functions directly or indirectly invoked by the function call. This enables us to give a higher weightage to any code segment is executed more frequently than one that is executed less frequently. As shown in figure 3.4, BB3 (which is executed thrice) calls function B thrice directly and function D only once indirectly. So, the size of BB3 is the total sum of size of the basic block, the size of function B and a third of the size of function D (because it is called only once).

In the second stage, we deal with any loops that might be present in the function body. This is necessary because a loop may increase frequency of basic blocks of the function which are a part of the loop body. This increased frequency may in turn skew the frequency weighted sum of the basic blocks which is the size of the function. It must be noted here that the second step also identifies loops in the application. The loops thus identified are used for subsequent cache tuning purposes at a later stage.

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

Figure 3.4: Calculation of function sizes

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

To identify a loop, we explore the control flow graphs of each function in the application. We use the dominator-join algorithm for the extraction purpose [45]. After applying this algorithm to the control flow graph of a function, we are able to identify all loops in the function body and the corresponding basic blocks for those loops. If those basic blocks call any functions, then the working set of the function would have already been added to the size of the basic block after the first stage. For calculation of loop sizes, we utilize the information about frequency of the loop and the size and execution frequency of the basic blocks in it. We use a few variables for this purpose which are described below.

- **Loop-Iteration Count:** This is the number of times a loop iterates. This variable is referred to as *LITC* subsequently.
- **Basic-Block Call Ratio:** This is the ratio of the number of times a basic block executes in the loop to *LITC*. Subsequently, this is referred to as *BBCR*.
- **Basic-Block Instruction Count:** This is number of distinct instructions executed per basic block for each call to the basic block. This can be obtained by dividing the total number of instructions executed as a part of a basic block by the total number of times the basic block is called. Henceforth, this is referred to as *BBIC*.
- **Normalized Loop Instruction Count:** This is the number of distinct instructions executed as a part of the loop body per iteration of the loop or the size of the loop. Henceforth, this is referred to as *NLIC*.

Using the variables described above, the value for *NLIC* of the loop can be determined by adding instructions from the basic blocks that are a part of the loop as shown in equation Eq. 3.1. After the value for *NLIC* has been obtained, the loop can be collapsed

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

and be represented by a single node as shown in figure 3.4 and for subsequent processing, $NLIC$ is representative of the number of instructions in the collapsed node.

$$NLIC = \Sigma(NBBIC \times BBCR) \quad (\text{Eq. 3.1})$$

In the final stage, we calculate the size of the function by adding a relative frequency weighted sum of the sizes of the basic blocks and loop nodes in it. This process is illustrated in figure 3.4. The complete algorithm for calculating the function sizes is shown in algorithm 1.

Algorithm 1 Calculating size of function F and finding loops in it

```

fbbset  $\leftarrow$  basic blocks of  $F$ 
for  $BB \in fbbset$  do
  if  $BB$  calls function then
     $BBsize \leftarrow SizeofInstinBB + FuncSetSize \times NormFreq$ 
  else
     $BBsize \leftarrow SizeofInstinBB$ 
  end if
end for
floopset  $\leftarrow$  loops in  $F$ 
for  $L \in flopset$  do
  loopBBSet  $\leftarrow$  basic blocks in  $L$ 
  for  $BB \in loopBBSet$  do
     $BBsize \leftarrow$  size of  $BB$ 
     $loopSize \leftarrow loopSize + BBsize \times NormFreq$ 
  end for
  CollapseLoopIntoNode
  SaveLoopInfoForTuning
end for
fbbnodeset  $\leftarrow$  basic blocks and loop nodes of  $F$ 
for  $BBLN \in fbbnodeset$  do
   $BBLNsize \leftarrow$  size of basic-blocks/loop-nodes in  $F$ 
   $funcSize \leftarrow funcSize + BBLNsize \times NormFreq$ 
end for

```

In algorithm 1, the major task is the identification of loops before the effective function sizes can be calculated. Therefore, the computational complexity of the entire process

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

hinges on the complexity of the loop identification algorithm. For our experiments, we use the Dominator-Join algorithm to identify the loops in the function [45]. This algorithm identifies both reducible and irreducible loops in the application. During the initial depth-first walk of the control flow graph as a part of this algorithm, the basic block sizes can be computed as the basic blocks are visited during the walk. After the depth-first walk is completed, the loop identification phase begins. If there are no irreducible loops in the application, then the computational complexity of the dominator-join algorithm is $O(|E|\alpha(|E|, |N|))$ where $|E|$ is the edge count of the control flow graph, $|N|$ is the node count of the control flow graph of each function and $\alpha(m, n)$ is the inverse Ackermann function. If there are irreducible loops in the functions, then assuming that irreducible loops are discovered at each level of loop depth, the computational complexity becomes $O(|E|\alpha(|E|, |N|) + k|E|)$ where k is the maximum depth of loop nesting in the function. Since this exercise is repeated for each function in our case, the complexity of the entire loop profiling process is $O(P(|E|\alpha(|E|, |N|) + k|E|))$ where P is the number of functions in the application.

As has been discussed earlier, due to the nature of code layout, some bytes from the memory may be brought into the cache even if they are not needed by the application. In other words, cache accesses by the application may not be aligned with cache line borders. So there can be potentially unused instructions in every cache line fetched from the memory. For incorporating such sparsely populated cache lines in our estimation of the size that a loop will occupy in the cache, we measure the cache line utilization (LUR) of the application by simulating it. The arrangement to measure the line utilization ratio is similar to a line buffer with a counter. The counter is incremented whenever the flow of control requires another cache line to be brought into the line buffer. This way we keep tabs on the total number of times the control flow jumps from the current cache line to another one. The estimation technique for Line Utilization also keeps count of the total

number of instructions. At the end of the simulation, the total number of instructions divided by the total number of lines accessed gives the number of useful instructions per line.

After obtaining the value for LUR , we multiply $NLIC$ with the instruction size (IS) and divide the result with LUR to obtain the size of the memory that the loop will occupy when it is stored in a cache. This variable is referred to as $LSIC$ (acronym for loop size in cache) subsequently and equation Eq. 3.2 shows how to calculate it.

$$LSIC = LS \div LUR \quad (\text{Eq. 3.2})$$

3.5 Estimation of cache hit rates from loop characteristics

In this section, we present equations to estimate the number of spatial hits and the number of temporal hits from the loop profile of the application. These equations are used by the hit rate estimation module to meaningfully estimate hit rates for multiple cache sizes for the application. The estimated hit rates are used for subsequent cache tuning.

3.5.1 Estimation of Spatial Hits

As per our definition of spatial locality, spatial hits are subsequent accesses (beyond the first instruction) to the instructions in a cache line after it is brought into the cache. So, the first instruction in the line does not contribute to spatial locality. Instead, it contributes to temporal locality if it is reused subsequently. When the cache line is reused subsequently, accesses to instructions in the line beyond the first instructions are again captured as spatial hits in our model. The first step in calculating the spatial hit rate for the application therefore is to calculate the line utilization ratio and then using

the block size of the cache and the size of each instruction, find the number of useful instructions per line. Out of these useful instructions, regardless of how many times the cache line is reused, the reuse of the first instruction will boost the temporal hit rate and the rest of the useful instructions in the line will contribute to the spatial hit rate. So, the spatial hit rate can be calculated from the line utilization ratio (LUR) and the instruction per cache block (IPB) using the equation [Eq. 3.3](#). Here, $(LUR \times IPB)$ represents the number of instructions executed from each cache block.

$$SHR = \frac{LUR \times IPB - 1}{LUR \times IPB} \quad (\text{Eq. 3.3})$$

3.5.2 Estimation of Temporal Hits

Unlike the benefit from the spatial hits, which is a function of cache block size, the possible benefit from temporal hits (reuse of the first instruction in the cache line) is a function of the cache size and the size of the loops. This is because whether the cache line belonging to a loop will be resident in the cache or flushed out in the subsequent iterations of the loop is a function of how big a cache we use. To estimate the approximate number of temporal hits, we have calculate the number of lines that the loop will span in the cache and the number of lines in the cache. This is done as shown in equation [Eq. 3.4](#) and [Eq. 3.5](#). Here C is the size of the cache, LL and CL are the number of cache lines that the loop can occupy and the total number of lines in the cache respectively.

$$CL = \frac{C}{IPB \times IS} \quad (\text{Eq. 3.4})$$

$$LL = \frac{LSIC}{IPB \times IS} \quad (\text{Eq. 3.5})$$

Once we know the number of lines that a loop spans and the total number of lines available in the cache, we calculate the cache residency (CR) of the loop - the average

number of lines retained in the cache per iteration of the loop. There can be the following three cases when a loop is executing.

- **When the complete loop is cache resident:** This happens when the cache size is larger than the loop size. In this case, the cache resident part of the loop is actually the entire loop body.

$$CR = LL \quad \text{if } LL < CL \quad (\text{Eq. 3.6})$$

- **When the partial loop is cache resident:** When the loop size is greater than the cache size but smaller than twice the cache size, then a certain portion of the loop will be flushed from the loop with every iteration. The part that remains behind in the cache is given by [Eq. 3.7](#).

$$CR = 2 \times CL - LL \quad \text{if } CL \leq LL < 2 \times CL \quad (\text{Eq. 3.7})$$

- **When no temporal locality is possible:** This happens when the loop size is greater than twice the cache size. In such a case, not a single line is able to remain in the cache because of being flushed out by other lines during the same iteration. In such a case, CR is zero.

$$CR = 0 \quad \text{if } LL \geq 2 \times CL \quad (\text{Eq. 3.8})$$

Once we know the cache resident part of a loop in terms of the number of lines, we can just multiply this value by $LITC$ to get the number of temporal hits (because temporal locality corresponds to only the first instruction in the cache line) and sum this up for every loop to get the total number of temporal hits for the application. This number divided by the total instruction count (TIC) would give us the temporal hit rate (THR) as shown in [Eq. 3.9](#).

$$THR = \frac{\Sigma(CR \times LITC)}{TIC} \quad (\text{Eq. 3.9})$$

The final step is to calculate the total hit rate (HR) of the cache. This is done by adding up SHR and THR as shown in [Eq. 3.10](#).

$$HR = SHR + THR \quad (\text{Eq. 3.10})$$

3.6 Tuning Heuristics for Predictor Based Filter Cache Hierarchy

To test our techniques for profile-driven cache tuning, we chose the predictor based filter cache hierarchy as a testcase. As discussed in chapter 2, this is a common solution for low power cache hierarchies in the embedded systems domain. In this section, we give an overview of the predictor based filter cache hierarchy and also discuss energy-centric heuristics for tuning the same.

The filter cache is a tiny auxiliary instruction cache whose purpose is to hold the instructions of the many small loops inherent in embedded applications. While these tiny loops execute, access to the small filter cache reduces the energy expended in the cache hierarchy. However, the filter cache could be a potential liability for larger loop sizes as they cannot be contained in the small cache. This is where the predictor in the predictor-based filter cache hierarchy becomes useful. The predictor minimizes accesses to the filter cache when the requisite instruction is not expected to be there by directing those accesses to the larger L1 cache instead. The predictor in a filter cache hierarchy works in a two stage manner as shown in [3.5](#). The first level is the cache line level. It assumes that the address of the next instruction to be accessed will be the current value of the program counter plus the instruction size. If the current instruction and the future instruction map to the same cache line, instruction access is directed to the filter cache. If they do not map to the same cache line, a pattern predictor is accessed to decide whether the next access will be from the L1 cache or the filter cache [\[61\]](#).

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

Figure 3.5: Filter cache prediction methodology

The important thing to note here is that when an instruction is accessed from the L1 cache, a full filter cache line containing the instruction is moved to the filter cache too. Subsequent access to instructions from the same line are done from the filter cache instead of the L1 cache. So in an ideal case, the predictor can sense that the instruction will be a miss in the filter cache and direct it to the L1 cache. The proportion of the instructions directed to the L1 cache will therefore be the application wide miss rate for the filter cache.

For evaluating the trade-offs in energy-centric cache tuning, we use energy-delay product as a metric for comparing the combined energy and performance metrics [29]. This metric has been used extensively in CAD tools and circuit designs to evaluate both energy and delay at the same time. There are numerous approaches which can be used to reduce the power consumption but these sacrifice arbitrary amounts of performance. For

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

example, by simply running the system at a lower clock frequency, the amount of power spent can be reduced because the amount of dynamic energy expended is proportional to the clock frequency. However, this would mean a drop in performance because of the slower clock. The relevance of this metric to cache tuning comes from the fact that including a smaller cache between the processor and a larger cache might reduce the energy consumed because of access to a smaller cache but it might also impose performance penalties because increased number of misses in the smaller cache as compared to the larger cache. For this reason, energy-delay product serves as a meaningful metric for comparing designs from the perspective of energy-centric cache tuning.

For calculating the energy-delay product of the filter cache and the L1 cache in the predictor based instruction cache hierarchy, we look at the following properties of the cache.

- **Cache Access Time:** This is average amount of time spent per cache reference (assuming it to be a hit) to retrieve the requisite instruction from it. In the following equations, this is represented as AT_{FC} for the filter cache and AT_{L1} for the L1 cache.
- **Cache Access Energy:** This is average amount of energy spent per cache reference (assuming it to be a hit) to retrieve the requisite instruction from it. In the following equations, this is represented as AE_{FC} for the filter cache and AE_{L1} for the L1 cache.
- **Cache Miss Rate:** This is the application wide miss rate for a given cache size. In the following equations, it is represented as MR_{FC} for the filter cache and MR_{L1} for the L1 cache.
- **Average Memory Access Time:** An ideal predictor can identify if a reference will be a miss in the filter cache and direct it to the L1 bypassing the filter cache

altogether. This way, when the instruction can be a hit in the filter cache, we get the latency of the filter cache and when it will be a miss in the filter cache, we get the latency of the L1 cache. In a predictorless filter cache hierarchy, the latency on a miss would be the access time of the filter cache plus the access time of the L1 cache. The average memory access time for the filter cache and the L1 cache is calculated as shown in equation Eq. 3.11 and Eq. 3.12. In these set of equations, $AMAT_{FC}$ is average memory access time for the filter cache and $AMAT_{L1}$ is the average memory access time for the L1 cache. HR_{FC} is the application-wide hit rate for the filter cache and AT_{L2} is the access time of the L2 cache.

$$\begin{aligned} AMAT_{FC} &= HR_{FC} \times AT_{FC} \\ &+ MR_{FC} \times AMAT_{L1} \end{aligned} \quad (\text{Eq. 3.11})$$

$$AMAT_{L1} = AT_{L1} + MR_{L1} \times AT_{L2} \quad (\text{Eq. 3.12})$$

- **Average Memory Access Energy:** The case for average memory access energy calculation in a predictor based filter cache hierarchy is different from that of the average memory access time calculations. When an instruction is accessed from the L1 cache, the line is transferred to the filter cache and subsequent accesses from the same line are made from the filter cache. So, for every access to the L1 cache, the filter cache is also accessed once. The average memory access energy for the filter cache and the L1 cache is calculated as shown in equations Eq. 3.13 and Eq. 3.14. In these set of equations, $AMAE_{FC}$ is average memory access energy for the filter cache and $AMAE_{L1}$ is the average memory access energy for the L1 cache. AE_{L2} is the access energy of the L2 cache.

$$AMAE_{FC} = AE_{FC} + MR_{FC} \times AMAE_{L1} \quad (\text{Eq. 3.13})$$

$$AMAE_{L1} = AE_{L1} + MR_{L1} \times AE_{L2} \quad (\text{Eq. 3.14})$$

- **Energy-Delay Product:** The energy-delay product is just the product of the average memory access time and the average memory access energy and is calculated as shown in equations [Eq. 3.15](#) and [Eq. 3.16](#)

$$EDP_{FC} = AMAE_{FC} \times AMAT_{FC} \quad (\text{Eq. 3.15})$$

$$EDP_{L1} = AMAE_{L1} \times AMAT_{L1} \quad (\text{Eq. 3.16})$$

After we obtain the values for the energy-delay product for a given cache configuration, we use these values to identify optimal (configuration with the lowest energy-delay product) L1 cache size and optimal filter cache size. In our exploration algorithm, we first find the L1 cache size that gives us the lowest energy delay product or $\min(EDP_{L1})$ in the set of L1 cache sizes that we explore. Once we know the optimal (lowest EDP) L1 cache size, we know the values for $AMAE_{L1}$ and $AMAT_{L1}$. Then, we repeat the same process to find the optimal filter cache size.

3.7 Overview of the cache tuning framework

In this section, we present the overview of the cache tuning framework based on the ideas discussed in this chapter. This framework is shown in figure [3.6](#) and adopts a three stage approach to identifying the optimal cache configuration for a given application.

The first stage in this framework is a one-time simulation of the application for which the cache needs to be tuned. The frequency-weighted function call graph of the application, the frequency-weighted function-specific control flow graphs and an estimate for the line utilization ratio (LUR) are obtained from the one-time simulation of the application.

In the second stage, the cache size values, call graph, control flow graphs and LUR are used to estimate the temporal hit rate using the techniques described in the preceeding sections. The LUR value is also used to estimate the spatial hit rate for the application.

Figure 3.6: Hit Rate estimation framework

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

The sum of the spatial and temporal hit rate gives an estimate of the total hit rate which is used in the final stage for cache tuning.

In the final stage, we identify the optimal cache configuration i.e. the optimal L1 and filter cache size for the application. After obtaining the hit/miss rates for multiple cache sizes using one-time simulation, we obtain the energy spent per access to these cache memories using the open-source CACTI tool.

CACTI [65] is an integrated cache access time, cycle time, area, leakage, and dynamic power model. Heuristics based on these models allow the user to observe the trade-offs in time, power and area for different cache sizes and associativities. The models take in cache capacity, block size, associativity as well as the process technology as inputs. The model has an optimization function which divides the data and tag array into number of sub arrays such that the configuration has the best access time and least energy consumption. The results obtained by CACTI were verified in [73] and were found to be close to the values obtained from post-layout simulations.

After obtaining the values for energy spent per access using CACTI, we calculate values for average memory access energy for a given cache size using equations described in section 3.6. We also obtain average memory access delay similarly. The product of the average memory access energy and average memory access delay gives us the value of energy-delay product for the given cache size. In this work, we search for cache configurations that result in the lowest energy-delay product of all cache sizes considered. We first search the entire range of cache sizes to identify the one that yields the lowest energy-delay product. This is identified as the optimal L1 cache. With the L1 cache size fixed, we then repeat the experiment to find the optimal filter cache - a smaller cache intended to hold small loops inherent in embedded applications - which would further reduce the energy-delay product. While searching for the optimal filter cache size, the upper limit of the range of cache sizes considered is set to be equal to the L1 cache size

thereby ensuring that the filter cache holds small loops inherent in embedded applications and saves on accesses to the larger L1 cache. At the end of this exercise, we obtain the optimal L1 cache size and filter cache size for the application under consideration.

3.8 Experimental results

For testing the proposed techniques, we chose benchmark programs from the Mibench [22] and the Mediabench [32] benchmark suite and executed them on the M5 simulator [9]. MiBench and Mediabench programs have been used extensively in studies on embedded system optimization. The M5 simulator is an instruction set simulator which can simulate a number of microprocessor instruction sets. For our experiments, we used the Alpha DEC instruction set.

For tuning the instruction cache hierarchy, we first estimate the hit rates for these cache sizes using the techniques described in this chapter. Then, we use those hit rate values to tune the predictor-based filter cache hierarchy. In the subsequent subsections, we first present results on the estimation of cache hit rate for multiple cache sizes and then show the efficacy of the proposed techniques for cache tuning.

3.8.1 Estimation of cache hit rates

We compared the estimated hit rates with actual hit rates obtained using the Dinero cache simulator for a direct mapped cache with a block size of 16B. The instruction size was assumed to be 4 B. So, every cache block was capable of holding up to 4 instructions. The range of cache sizes for which the hit rate estimation was done was from 256 Bytes to 32 kB. As can be seen in figure 3.7, the estimated hit rate values (shown as a dotted line) follows the actual hit rate values (shown as a solid line) closely for a wide range of cache sizes.

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

3.7.a: Patricia

3.7.b: SHA

3.7.c: Rawcaudio

3.7.d: Rawdaudio

3.7.e: Cjpeg

3.7.f: Djpeg

3.7.g: Epic

3.7.h: Unepic

Figure 3.7: Actual vs. estimated hit rates

Cache parameter	Value or range
FC size range	256B to L1 Size
FC block size	16B
FC associativity	1
FC access time	1 cycle
L1 size range	1kB to 32kB
L1 block size	32B
L1 associativity	4-way
L1 access time	1 cycle
L2 size	512kB
L2 block size	32B
L2 associativity	4-way
L2 access time	6 cycle

Table 3.2: Instruction cache parameters for design space exploration

3.8.2 Comparison of proposed techniques with exhaustive simulation

The cache hierarchy for our experiments consisted of a split instruction-data cache connected to a unified L2 cache of 512kB with a 32B block size and 4-way set associativity. For the instruction cache tuning experiments, we assume a predictor-based filter cache hierarchy for instructions. The measurements of access energy per access for the different cache configurations were obtained using the CACTI tool [65]. The selection space for the design space exploration and the instruction cache parameters used for the tuning experiments are shown in table 3.2.

Table 3.3 shows the filter cache and L1 cache sizes chosen through exhaustive simulation (sim) and through the estimation heuristics (est). The exhaustive simulation approach is always able to identify the optimal configuration - the one with the least EDP - because it resorts to brute force search. However, our estimation heuristics are also able to identify the optimal configuration or a near-optimal one in all cases. The EDP of the configurations chosen by the two techniques is shown (in figure 3.8) next to the EDP of a base 32kB, 4-way set associative and 32B line size L1 cache which is the

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

Benchmarks	Estimate (FC,L1)	Exhaustive (FC,L1)
pat	(1kB,32kB)	(2kB,32kB)
sha	(256B,1kB)	(256B,2kB)
rawc	(512B,1kB)	(512B,1kB)
rawd	(512B,1kB)	(512B,1kB)
cjpeg	(512B,4kB)	(1kB,4kB)
djpeg	(1kB,4kB)	(2kB,8kB)
epic	(256B,1kB)	(512B,1kB)
unepic	(256B,1kB)	(1kB,4kB)

Table 3.3: Filter cache and L1 cache sizes in Bytes chosen by estimation techniques and exhaustive simulation

norm in many processors. As is evident from the figure, the EDP of the configuration obtained through the estimation techniques is optimal in almost all cases and near-optimal for the rest.

Figure 3.8: Relative EDP of cache configurations reached by estimation and exhaustive simulation as compared to a 32 kB L1 cache

While the estimation technique has been shown to identify optimal or near-optimal cache configurations in almost all cases, the real value of the proposition comes through in the runtime of the algorithm and its scalability. Table 3.4 shows the time that was required to run the estimation algorithms as compared to an exhaustive simulation. As

Benchmark	Time(est)	Time(sim)	Savings
pat	247s	1421s	83%
sha	121s	350s	65%
rawc	40s	166s	76%
rawd	45s	131s	66%
cjpeg	91s	314s	71%
djpeg	44s	120s	63%
epic	302s	1043s	71%
unepic	54s	205s	74%

Table 3.4: Comparison of the average runtime for the estimation algorithm and exhaustion simulation techniques

can be seen, a significant amount of time is saved if the design space exploration is done using the estimation heuristics.

The time complexity of the exhaustive simulation techniques is $O(n)$ where n is the number of configurations tested. On the other hand, the time complexity of the estimation heuristics is constant irrespective of the number of cache sizes that the estimation is required for. This constant time is the time taken to simulate the application, build its call graph and control flow graph and to estimate the EDP for different cache sizes through a one-time loop profiling. Therefore, the proposed techniques offer a highly scalable solution for the cache tuning problem as compared to exhaustive cache hierarchy simulation.

3.9 Proposed techniques for tuning of block sizes and associativity

In the preceeding sections, we proposed techniques to identify the optimal filter and L1 cache size given an application. For our tuning experiments, we fixed the block size to 16B and the associativity to one for the filter cache. Similarly, we also fixed a block size of 32B and an associativity of four for the L1 cache. While it is true the cache size has the maximum and over-ruling effect on the energy delay product, tuning other cache

parameters can result in further albeit smaller reduction in the energy-delay product. Therefore, in this section, we discuss techniques to extend the dynamic control flow graph based cache tuning to include other cache parameters like block size and associativity as well.

3.9.1 Evaluation of block sizes

Cache blocks capture spatial locality in an application. They are supposed to store memory references adjacent to the current reference in anticipation that these memory locations may be referenced in the future. Therefore, the ideal block size should be able to capture as many *runs* (sequence of accesses to consecutive memory locations) as possible. However, care must be taken not to choose too big a block size as it may unnecessarily store memory locations which may never be referenced because a typical run may be smaller than the block size.

To evaluate the suitability of a block size, we leverage on existing work done to evaluate the effect of cache block size on the miss rate of the cache [1]. These techniques need information on the distribution of the run length to determine the effect of changing the block size. In this section, we briefly explain the techniques proposed in [1]. We show that our profiling framework can be extended to obtain the input variables required by these techniques and that the result from these techniques can be applied to block size tuning.

The expression for the number of unique cache blocks referenced ($u(B)$) corresponding to the number of unique memory locations ($u(1)$) was derived in [1] and is shown in equation Eq. 3.17. In this equation, l_{av} is the average run length, $R(l)$ is the probability of a run being of length l and B is the block size in terms of the number of references it can hold.

$$u(B) = \frac{u(1)}{l_{av}} \sum_{l=1}^{l=\infty} R(l) \left(1 + \frac{l-1}{B} \right) \quad (\text{Eq. 3.17})$$

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

Equation [Eq. 3.17](#) can be explained as follows. If l_{av} is the average run length, then the probability that a reference will start a new run is $\frac{1}{l_{av}}$. Therefore, if there are $u(1)$ unique references, there will be approximately $\frac{u(1)}{l_{av}}$ unique runs. Now, we do not know how long these runs will be or how many blocks those runs will span. The first step to tackle this is to determine how many cache blocks a run of length l will span. If there is ideal alignment (the run starts on a cache block boundary), then the number of blocks it will span will be $\lceil \frac{l}{B} \rceil$. However, the alignment is seldom ideal and is often random. It can be approximated by $1 + \frac{l-1}{B}$. The average number of blocks that a run will span can be approximated by multiplying this value with the probability of a run being of length l which is $R(l)$ and then summing it up for all l . This value when multiplied with the number of unique runs, gives us the value for the number of unique cache blocks that the execution will span as shown in equation [Eq. 3.17](#).

Tweaking our profiling framework to obtain the values for $u(1)$ and l_{av} is straightforward. The value for $u(1)$ can be obtained from basic blocks in the weighted control flow graph. In this case, care must be taken to get a frequency weighted value for the number of unique reference so that only the more frequent runs influence the block size that is ultimately chosen. The value for the average run length can be obtained while simulating the application. For this purpose, the number of times the memory reference is non-sequential must be counted. The number of instructions simulated divided by this value would give an indication of the average run length l_{av} .

Obtaining values for $R(l)$ is slightly more involved. In the subsequent part of this discussion, let f_{li} be the probability that the next reference will be sequential given the last i addresses are sequential. It was shown in [\[1\]](#), that for $i > 1$ the values for f_{li} were similar and whether the subsequent access would be sequential does not depend on the number of references in the run. In other words, sequential accesses showed a memoryless property. Therefore, the values for f_{lk} could be approximated to be equal to f_{l2} for $k > 2$.

This gave the following expressions for $R(l)$.

$$R(l) = \begin{cases} 1 - f_{l1} & \text{if } l = 1 \\ f_{l1}f_{l2}^{l-2}(1 - f_{l2}) & \text{if } l > 1 \end{cases} \quad (\text{Eq. 3.18})$$

These expressions pose the added question of computing values for f_{l1} and f_{l2} . The probability of a run being of unit length ($R(1)$) can be calculated while simulating the application in a way similar to measuring the average run length. From this, the value for f_{l1} can be calculated. Furthermore, using the expressions for $R(l)$, an expression for the average run length can be calculated which is shown in equation [Eq. 3.19](#). Using this equation and the measured values for f_{l1} and l_{av} , we can calculate the value for f_{l2} .

$$l_{av} = \frac{1 + f_{l1} - f_{l2}}{1 - f_{l2}} \quad (\text{Eq. 3.19})$$

At the end of this exercise, it is possible to quantify how many unique cache blocks will be required for the execution of the application given a block size. After the optimal cache size is identified using the tuning techniques discussed earlier in this chapter, the size of the cache block can be tweaked. The tweaking should be done in such a manner that the product of the the number of unique blocks time the block size ($u(B) * B$) should be roughly equivalent to the cache size C .

3.9.2 Evaluation of associativity

Finding the optimal level of associativity for a given application involves minimizing the number of cache line conflicts due to two different references occupying the same cache location. It must be noted here that it is sufficient to just consider loops in an application while exploring to find the optimal level of associativity. The references made while executing a loop are the ones which will be referenced again in the future. If these references are evicted from the cache, overheads will be incurred in fetching them again from memory into the cache. Since we identify all loops in the application as a part of

our tuning process, we know the distribution of references that loops in the application may make. This enables us to compute how many different blocks will map to the same cache set.

To identify the optimal level of associativity for an application, we first identify the right cache size and block size using techniques described in this chapter. Then, for a given loop and a given level of associativity, the number of cache blocks from the loop that will map to each of the cache sets can be determined. Care must be taken at this stage to consider the frequency of execution of these blocks as well in order to give the more frequent blocks more weightage. Therefore, a frequency weighted approach to estimate the number of cache blocks that will map to the cache set is recommended. For the loop, if the number of blocks mapped to the same cache set exceeds the set size D , then blocks will be evicted from the cache set as the loop iterates. The total number of misses resulting from the eviction of cache blocks will depend on the frequency of the loop. Adding up the potential misses for a given set size for all loops in the application will give an indication of the penalty of using that particular set size. Then, the associativity can be increased or decreased to reduce the number of conflict misses and subsequently reduce the energy-delay product of the cache as well.

3.10 Summary

In this chapter, we discussed the shortcomings in existing loop-profile driven schemes for cache tuning and also proposed solutions for the same. In existing techniques, there was no mention of how to calculate the working set for loops that called functions. Existing techniques were only able to identify reducible loops and were insensitive to the sparsely populated cache lines that occur due to non-ideal alignment of basic block boundaries with cache block boundaries. We developed novel techniques to calculate loop sizes and

CHAPTER 3. CACHE TUNING FOR SINGLE-TASK BASED EMBEDDED APPLICATIONS

frequencies accurately by representing the application as a combination of a function-level call graph and control flow graphs of individual functions. These loop profiles were successfully applied to estimate the cache hit rate for a large range of cache sizes and the estimated hit rates were shown to follow the measured hit rates closely. The estimated hit rates were also used to tune the predictor based filter cache hierarchy - a common solution for low-power embedded systems. The proposed heuristics were able to accurately predict the optimal filter and L1 cache sizes for the application. The proposed techniques were also shown to be much faster than exhaustive simulations of the memory hierarchy and at the same equally reliable in choosing the optimal cache size. After demonstrating the efficiency of the proposed techniques in identifying optimal filter and L1 cache sizes for a given application, we showed how the proposed techniques can be extended to identify the optimal level of associativity and block size as well. The techniques proposed in this chapter are fast, accurate and highly scalable for the purpose of cache tuning and can be relied upon to tune cache parameters for any application.

Chapter 4

Cache tuning for RTOS-driven multitasking applications

4.1 Introduction

Loop profiling enabled cache tuning has been shown to achieve considerable energy savings for standalone embedded applications. While such loop profiling techniques work well for single-task based applications, there has been no work done to extend them to multitasking systems running on a real-time operating system. Therefore, we develop and validate profile driven schemes for cache tuning for multitasking applications in this chapter.

The challenge in applying loop profiling algorithms to tuning multitasking systems lies in the inherent pseudo-random nature of control flow in multitasking systems. In a stand-alone application or single-task-based application, there is no sharing of system resources like processor, memory, I/O, etc. In such systems, the functions that need to be performed by the stand-alone application proceed without interruptions because it is the sole master of all resources available to it. However, in a multi-task based system, tasks compete with each other for system resources and end-up pre-empting each other as a result of mutual competition. This pre-emption makes the control flow of multitasking systems pseudo-random and prohibits direct extension of loop profiling techniques developed for stand-alone applications to include multitasking systems.

With the increasing complexity of modern day embedded applications, RTOS based multitasking systems are fast becoming the norm. There exists a need to extend existing cache tuning techniques developed for standalone embedded applications to multitasking systems. In this chapter, we present a framework for energy centric-tuning of the instruction cache for embedded multitasking applications. Our framework is built upon an analysis of control flow of multitasking system and how the pseudo-random flow of control affects execution of loops and consequently, the size of the optimal cache for the multitasking application. We validate our proposed techniques by using it to tune a predictor based filter cache hierarchy for instructions. For all the multitasking programs tested, our techniques are able to successfully predict configurations that are optimal or near-optimal. The proposed methods are also able to achieve speed-ups of up to an order of magnitude compared to exhaustive design space exploration techniques.

The rest of the chapter is organized as follows. In section 4.2, we analyze the nature of control flow in multitasking application and comment on its determinism. We show that the control flow in multitasking systems is part deterministic and part non-deterministic resulting in a pseudo-random flow. We discuss the impact of this pseudo-random control flow on cache accesses in section 4.3. We show that the pseudo-random control flow results in two kinds of misses - misses due to instructions of the same tasks or intrinsic interference and misses due to competition between different tasks to occupy the instruction cache or extrinsic interference. We show that misses due to intrinsic interference are much lesser as compared to extrinsic interference through empirical experiments and explain the observation theoretically as well. Based on our observations on the relative impacts of intrinsic and extrinsic cache misses, we propose our tuning framework in section 4.4. We present our experimental results on instruction cache tuning for multitasking systems in section 4.5. We present a method to estimate the number of extrinsic instruction cache misses in section 4.6 and summarize the chapter in section 4.7.

4.2 Analysis of program execution in multitasking applications

In this section, we analyze the nature of control flow in multitasking systems. We start off by examining the execution footprint of a typical multitasking application. Then, we make deductions on the level of determinism in control flow of multitasking applications.

4.2.1 Execution footprint of a multitasking application

There are inherent differences in the way a task is executed in a single-task based system and a multi-task based system. In single task systems, the system has to do perform only one task and therefore, the lone task runs on the processor all the time. There is no other task to pre-empt the execution of the single-task based application. On the other hand, in multitasking systems, there are multiple tasks being serviced by the processor. They compete with each other for resources like the processor, memory, etc. This creates a complicated situation where there is a need for task arbitration to decide which task gets to execute and a task is often required to be pre-empted before it can finish execution. Such arbitrations to allocate system resources are usually performed by an operating system.

A typical execution footprint of a multitasking system is shown in figure 4.1. As can be seen, execution of task A is divided into four sections - A1, A2, A3 and A4. There could be many reasons due to which the control flow may shift to the RTOS or other tasks. The suspension of execution of the tasks could be because it may request to perform a privileged action which is usually achieved through system calls. In this example, task A may request to read from a file through system call SA1 and the request is serviced by the RTOS. The suspension of task A could also be because of the occurrence of a trap or interrupt which needs urgent servicing from the RTOS. An example of such an interrupt due to which the execution of task A needed to be suspended is IT3. Another reason

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.1: Execution footprint of a multitasking application

for the suspension of the execution of a task could be because it has already run for a while and the RTOS feels that it should yield the processor to other tasks with higher priorities which are waiting to execute. The RTOS ensures that each task gets a fair share of processor time by keep tabs on the amount of time a process executes. A process is allowed to execute one time-quantum at a time called a *timeslice*. After expiration of a timeslice, the scheduler in the RTOS picks the task with the highest priority that is ready to run. The scheduler is also responsible for putting processes to sleep if they request a resource currently held by a higher priority task and also to wake up tasks if a resource that they requested becomes available. In figure 4.1, the RTOS switches to the higher priority task B after the expiration of the *timeslice* for A at the end of A2.

4.2.2 Pseudo-random control flow

When an application executes in a single-task based system, the sequence of execution within the application can be analyzed. It is possible to know what are possible sequences of basic blocks that can be executed during the execution of the task based on analysis of the control flow and data flow in the task. For example for task A in 4.1 , the executions of segments A1, A2, A3, A4 and the system call SA1 are all part of a possible sequence of execution for task A which can be captured using compile time analysis of task A.

Commenting on the sequence of execution in multi-task based systems is however trickier and not always possible. It has already been discussed that in a multitasking system, the execution of individual tasks is interspersed with the execution of other tasks. The order in which the execution is interspersed depends on circumstances which can only be determined at runtime. The pattern of switching amongst tasks is therefore difficult to predict inducing a sense on non-determinism in the execution sequence. A good example of such non-determinism would be the occasional execution of interrupt or trap service routines or switching to other user tasks. The execution of these interrupt

or trap service routines is not specified explicitly but they need to be executed every once in a while because they are critical to the execution of the multitasking system. For example, IT3 and IT5 could very well be critical interrupts that require immediate servicing by the RTOS and therefore require that the current task be abandoned and the interrupt be serviced.

This element of non-determinism combined with the level of confidence about the expected flow of control in individual tasks makes the execution sequence in multitasking system pseudo-random. Any technique for analysis of multitasking systems must therefore be sensitive to the pseudo-random nature of control flow in multitasking systems.

4.3 Impact of pseudo-random control flow on cache accesses

Execution of any application as a single-task based system is very different from its execution in a multitasking system. While a task is executed without interruptions in a single-task based system, its deterministic control flow is sometimes suspended in a multitasking system. This could be because of a plethora of reasons over which the task may or may not have any control. These reasons could be the unavailability of a system resource, competition for processor time with a task having a higher priority, processing of an asynchronous event like an interrupt, trap, etc by the operating system. In the case the task execution is suspended, the multitasking application takes a non-deterministic control path and switches to another task or interrupt service routine. Both deterministic and random control paths in multi-tasking applications have implications on the cache behaviour of the application. The nature and impact of this implication is discussed in this section.

Figure 4.2: Flow of control in multitasking systems

4.3.1 Intrinsic and extrinsic code interference

Corresponding to the two different types of control flow in multitasking systems, there are two ways in which a cache line could be evicted out of the cache resulting in a cache miss for future references to the same location. We discuss these two kinds misses named as intrinsic misses and extrinsic misses below. Intrinsic interference is the competition amongst the instructions of the same task to occupy the cache. For example, in figure 4.2, it is possible that BB1 and BB5 occupy the same cache line. In such a case, the contents of BB1 will be evicted from the cache by BB5. This will result in a cache miss the next time BB1 is referenced. Such misses are categorized as missed due to intrinsic interference.

Extrinsic interference on the other hand is the competition amongst different tasks and interrupt service routines to fill the cache with their own instructions. Extrinsic interference is a direct result of the interspersed execution of the tasks in multitasking systems. To explain the cause for extrinsic interference, take the case of figure 4.2 as an

Figure 4.3: Evaluation of intrinsic and extrinsic interference

example. In this case, it is likely that some instruction of BB1 that are already present in the cache may be evicted out of the cache if task 'X' is executed. This may lead to cache misses when those instructions are executed again. Such misses are categorized as misses due to extrinsic interference and are in addition to the misses due to intrinsic interference that the application may suffer.

4.3.2 Empirical evaluation of cache misses due to code interference

To evaluate the relative frequency of intrinsic and extrinsic misses, we simulated a few multitasking applications and measured the miss rates. In our experimental setup, shown in 4.3, we connected dedicated cache for each task and housekeeping functions that the RTOS may perform. The number of misses incurred in each of these cache memories

Multitasking benchmark	MR increase %	EDP increase %
dijk + patr + sha	0.08	0.37
rawd + cjpeg + sha	0.02	1.01
epic + unepic + rawc	0.02	0.90
djpeg + rawc + sha	0.02	0.96
sha + cjpeg + rawc	0.02	1.03
rawc + rawd + cjpeg + djpeg	0.02	0.86
patr + dijk + rawc + sha	0.02	0.34
dijk + sha + rawc + rawd	0.02	0.70

Table 4.1: Average increase in miss rate and energy delay product

is the number of intrinsic misses incurred by the corresponding task or ISR/TSR. If these misses are added up, we can obtain the total number of intrinsic misses incurred if these elements are run together in a multitasking system. We connected the same simulation of the multitasking system to a global cache as well as shown in 4.3. The number of misses resulting from this cache will include both the intrinsic and extrinsic misses. The extrinsic misses in this case will be the result of cache line evictions due to interference amongst tasks and other RTOS housekeeping functions. As expected, there was a decrease in the number of misses as compared to the first experiment since the second experiment ignored the case of extrinsic interference. Due to the lowered number of misses, the energy delay product for the second experiment was also found to be lower as compared to the first experiment.

Figure 4.4 shows the miss rate curve and the energy delay for two of eight multitasking programs tested. The difference in the miss rates and hence the energy delay product because of extrinsic interference, however, is found to be insignificantly low. This conclusion is further validated in table 4.1. The average difference in instruction cache miss rate is only 0.02% to 0.08% of all instructions executed. Consequently, as shown in table 4.1, the difference in energy delay product is also found to be negligible.

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.4: Difference between effects of standalone task/interrupt simulation added together and the actual multitasking system 4.4.a, 4.4.b, 4.4.c and 4.4.d

4.3.3 Theoretical explanation of empirical observations

The results obtained can be explained based on the observation that misses due to extrinsic interference happen once in a while as compared to missed due to intrinsic interference. They happen only when the thread of execution switches from one task to another or the other housekeeping functions that the RTOS may perform like servicing interrupts, traps, etc. As the execution proceeds after the switch, these transient misses incurred become negligible as compared to the rest of the misses incurred due to intrinsic interference or cache line conflicts.

To gauge the relative impact of extrinsic interference as compared to intrinsic inter-

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.5: Effect of extrinsic interference

ference, consider the example shown in figure 4.5. Here, the cache contents are shown to contain a loop consisting of four basic blocks - BB1, BB2, BB3 and BB4. However, during the execution of the loop, it is pre-empted by 'X' which could be a another task or an interrupt service routine. When the loop resumes execution after 'X' has finished executing, some of the instructions belonging to the loop would have been evicted. But, after just one iteration, the cache is filled up again with insructions of the loop and subsequent accessess to the instructions of the loops are hits.

Intrinsic interference in the example shown in figure 4.5 depends on the frequency of the loop shown in figure 4.5 which is significantly higher than the number of pre-emptions that might happen during the lifetime of the loop. It is therefore expected that the number of misses due to pre-emption or extrinsic interference will be much lesser as compared to the number of misses due to intrinsic interference. Considering that embedded applications spend a significant amount of time in such loops, it can be safely assumed that the frequency of these loops will be very high as compared to the misses due to the occasional pre-emptions of those loops as shown in this subsection. Consequently, the damage done by intrinsic interference is much higher as compared to extrinsic interference and therefore, intrinsic interferenceforms can be treated as the main driving force behind choosing an appropriate cache for the application using the cache tuning heuristics.

4.4 Proposed technique for cache tuning for multi-tasking systems

In the previous section, we discussed the relative impact of extrinsic interference and intrinsic interference on the total cache hit/miss rate. We showed that misses due to extrinsic interference to be significantly lesser than those due to intrinsic interference.

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

In this section, we develop ideas for a cache tuning framework for multitasking systems based on these observations.

As a task executes in a multitasking environment, the instructions of the loops would be flushed once in a while by intervening tasks or interrupts. However, the loops themselves will execute to completion regardless of the number of times they are pre-empted. The pre-emption might result in a few additional misses as shown in the preceeding section but it should not affect the optimal cache size for the application provided all the loops and their relative frequencies are taken into account in choosing the cache size.

To make this point clear, consider a two task system as an example. Task 1 (T1) has a loop of size 1kB which executes an aggregate of 1000 times. Lets assume that its execution is interspersed with another task 2 (T2) which does not have any loops in it. In this case, an ideal cache size for the application would be 1kB because it can hold the loop in T1 and mitigate any unnecessary cache misses. However, if there is another loop in T2 which has a size of 4kB and a significant frequency, then the optimal cache size for the system would be between 1kB and 4kB. The exact size of the optimal cache would depend on the frequency of the loop in T2. If the frequency is too low, then a cache size of 1kB mght be just right for the application. On the other hand, if it is too high, it would be wise to choode a cache size of 4kB. The choice of the optimal cache would be independent of how the execution of T1 and T2 is interspersed with each other because regardless of how many times T1 and T2 pre-empt each other, the loops in them would execute to completion.

Regardless of the order in which the tasks execute in a multitasking environment, the sole factor that determines the optimal cache size is the size and frequency of all the loops inherent in the tasks. In the following subsections, we present the broad methodology of our approach to profiling multitasking systems for cache tuning and the cache tuning framework itself. Our techniques rely on simulating the multitasking application to

obtain information about the loops using loop profiling techniques discussed earlier. For all purposes, we ignore the order in which tasks execute and just focus on obtaining relative frequencies of the loops in the multitasking application.

4.4.1 Loop profiling of user tasks

In a multitasking environment, the operating system is responsible for process, memory and I/O management. This is different from the execution of a standalone task wherein the task has absolute control over memory and I/O and can read and write to and from these entities as and when it wishes. Take as an example, the case of writing to memory. In the case of a standalone task, this operation proceeds without any checks on whether the task is permitted to write to the memory location or whether the memory resource is free in the first place. However, in a multitasking environment running under the control of an operating system, the tasks can write to a memory only through a system call by invoking the operating system. In a system call, the control is handed over to the operating system which follows a pre-defined set of checks and protocols before it allows the write to proceed.

Such system calls are often invoked from within loops in user tasks. Any technique for loop profiling of user level tasks must take into account any system call that the loop may invoke. The size of the system call(s) added to the size of the loop gives an accurate indication of the size of the cache required to hold the loop. For this reason, we profile user level tasks by taking into account any system calls that they might invoke. Take the example of figure 4.6, as task A executes, it makes system call SA1. This system call could very well be invoked from within a loop. Therefore, it is imperative to consider system calls as a part of loops in user tasks.

To identify the nature of loops user task, we simulate the multitasking application but profile individual tasks as if they were the only task running on top of a RTOS

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.6: Cache tuning methodology for multitasking systems

as shown in 4.6. Thereby, we ignore any pre-emptions that it might suffer from other tasks or other RTOS house-keeping functions like task management and interrupt/trap handling but ensure that the system calls are taken into account while calculating loop sizes for the task. This technique will give us an indication of the intrinsic misses that the multitasking application is likely to incur because of this task. If there are multiple tasks running on the system, this exercise can be repeated for every task executing on the system.

4.4.2 Loop profiling of the RTOS

Taking the tasks into consideration alone is not enough. Locality analysis for cache tuning must also take into account any loops that are executed by the operating system for house-keeping purposes. In a RTOS based multitasking system, the RTOS is the manager of processes, memory, I/O, etc. When executing multiple tasks, there are bound to be overheads in executing the RTOS code that is responsible for arbitrating multiple tasks competing for resources. For example, one of these tasks is to maintain timers to monitor how long a task has been running. For this, it has to respond to interrupts from the hardware timer periodically to update process times. Once a task exceeds the timeslice allocated to it, the RTOS is responsible for putting it to sleep and selecting a new process to run based on a scheduling algorithm. Task switching in this case would involve saving the context of the current task and loading the context of a new task.

Another one of such tasks performed by the RTOS would be to respond to page faults. In multitasking systems, because the memory is shared by several tasks, every task is given the impression that it is working with large, contiguous sections of memory. In reality, each process' memory may be dispersed across different areas of main memory, or to secondary storage. When a process requests access to its memory, virtual addresses are mapped to physical addresses through a page table. Sometimes it so happens that a

physical memory frame may not have been allocated for the virtual address. In such a case, there will be a page fault and the operating system is responsible for dealing with it. The handler for the page fault allocates a physical memory frame to the virtual page and create a page table entry and the memory access will continue.

It is evident from the above that the RTOS needs to do a lot of house-keeping functions when multitasking applications run on it because all system resources are shared by multiple tasks. Therefore, for tuning the cache for multitasking systems, it is imperative to consider the loops in the house-keeping functions that the RTOS performs. These loops will be in addition to any user task level loops or loops in system calls. To identify these loops, we simulate the multitasking application and profile the RTOS components responsible for performing these functions independent to profiling the tasks. This approach is shown in figure 4.6 wherein IT1 to IT6 are profiled as being part of the RTOS housekeeping functions.

4.4.3 Overview of the tuning framework

The overall framework for tuning the cache for multitasking system is shown in figure 4.7. The framework is divided into three major components. In the first stage, we simulate the multitasking system to obtain the call graph and control flow graphs of the individual tasks, the associated system calls and other house-keeping functions that the RTOS may perform for the execution of the multitasking application. After obtaining the call graph and the control flow graphs, we profile them on a task by task basis to obtain the nature of the loops inherent in them. We also profile the RTOS for obtaining the loops in all the house-keeping functions that it may perform. After obtaining information about all loops, we use this information to estimate the hit rates for multiple cache sizes in one go. In the final stage, we feed the estimated hit rates to a tuning engine which performs cache tuning based upon a predefined set of heuristics. Because we perform

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.7: Cache tuning framework for multitasking systems

energy-centric cache tuning of the predictor based filter cache hierarchy in this work, this module also uses the cache energy per access obtained from the CACTI models. The final output of the framework is the optimal filter cache and optimal L1 cache size i.e. the cache sizes which results in the lowest energy delay product for the application.

4.5 Experimental results

In this section, we present our results on cache tuning for multitasking applications. We first discuss our experimental setup and then go on to compare design space exploration using our proposed techniques with design space exploration using exhaustive simulation.

4.5.1 Experimental Setup

We ran the benchmark programs from the Mibench [22] and the Mediabench [32] benchmark suite on the M5 simulator [9]. The M5 simulator simulated a single issue processor with the Alpha Dec instruction set. The operating system running on the simulator was Linux. We assumed a unified L2 cache of 512kB with a 32B block size and 4-way set associativity. The selection space for the design space exploration and the instruction cache parameters used are shown in table 4.2. The measurements of access energy for the different cache configurations were obtained using the CACTI tool [65].

4.5.2 Comparison of proposed techniques with exhaustive simulation

The proposed EDP estimation techniques closely follow the results obtained through exhaustive simulation of the instruction cache hierarchy. Table 4.3 shows the filter cache and L1 cache sizes chosen through exhaustive simulation (sim) and through the estimation heuristics (est). The exhaustive simulation approach is always able to identify the optimal configuration - the one with the least EDP - because it resorts to brute force search.

Cache parameter	Value or range
FC size range	256B to L1 Size
FC block size	16B
FC associativity	1
FC access time	1 cycle
L1 size range	1kB to 32kB
L1 block size	32B
L1 associativity	4-way
L1 access time	1 cycle
L2 size	512kB
L2 block size	32B
L2 associativity	4-way
L2 access time	6 cycle

Table 4.2: Instruction cache parameters for design space exploration

Multitasking benchmark	Config(est)	Config(sim)
dijk + patr + sha	(1kB,32kB)	(1kB,32kB)
rawd + cjpeg + sha	(512b,4kB)	(512B,4kB)
epic + unepic + rawc	(256B,1kB)	(512B,1kB)
djpeg + rawc + sha	(512B,4kB)	(1kB,4kB)
sha + cjpeg + rawc	(512B,4kB)	(512B,4kB)
rawc + rawd + cjpeg + djpeg	(1kB,4kB)	(1kB,4kB)
patr + dijk + rawc + sha	(1kB,32kB)	(1kB,32kb)
dijk + sha + rawc + rawd	(256B,4kB)	(512B,4kB)

Table 4.3: Filter cache and L1 cache sizes chosen by the estimation algorithm and the exhaustive simulation

However, our estimation heuristics are also able to identify the optimal configuration or a near-optimal one in all cases. The EDP of the configurations chosen by the two techniques is shown (in figure 4.8) next to the EDP of a base 32kB, 4-way set associative and 32B line size L1 cache which is the norm in many processors. As is evident from the figure, the EDP of the configuration obtained through the estimation techniques is optimal in almost all cases and near-optimal for the rest.

While the estimation technique has been shown to identify optimal or near-optimal cache configurations in almost all cases, the real value of the proposition comes through in the runtime of the algorithm and its scalability.

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

Figure 4.8: Savings in EDP achieved by estimation algorithm vs. exhaustive simulation

Multitasking benchmark	Time(est)	Time(sim)	Savings
dijk + patr + sha	6m 29s	35m 17s	82%
rawd + cjpeg + sha	4m 15s	11m 18s	62%
epic + unepic + rawc	2m 44s	23m 50s	89%
djpeg + rawc + sha	1m 56s	9m 30s	80%
sha + cjpeg + rawc	1m 56s	13m 13s	85%
rawc + rawd + cjpeg + djpeg	3m 35s	13m 17s	73%
patr + dijk + rawc + sha	11m 56s	39m 39s	70%
dijk + sha + rawc + rawd	5m 59s	18m 1s	67%

Table 4.4: Comparison of the average runtime for the estimation algorithm and exhaustion simulation techniques

Table 4.4 shows the time that was required to run the estimation algorithms as compared to an exhaustive simulation. As can be seen, a significant amount of time is saved if the design space exploration is done using the estimation heuristics. For some of the multitasking programs, the estimation techniques are able to achieve a speedup of upto an order of magnitude.

Another huge advantage of such an approach is its scalability. The time complexity of the exhaustive simulation techniques is $O(n)$ where n is the number of configurations

tested. On the other hand, the time complexity of the estimation heuristics is constant irrespective of the number of cache sizes that the estimation is required for. This constant time is the time taken to simulate the application, build its call graph and control flow graph and to estimate the EDP for different cache sizes through a one-time loop profiling.

4.6 Estimation of misses due to extrinsic interference

While the number of misses due to extrinsic interference or code mixing have been shown to be nominal in the preceeding section. It is useful to have a way to estimate the number of such misses. In this section, we describe a technique to estimate the number of misses due to code mixing. Our approach is based on estimating the worst-case and best-case misses due to code mixing and then taking the average of the best and the worst case as the estimate. In the subsequent part of this section, we first describe how to obtain the worst-case estimate, then we discuss what is the best-case estimate and finally conclude the section by show that the average-case estimate follows the actual number of misses incurred due to code mixing closely.

We start off the estimation process by estimating a worst-case estimate for the number of misses that could happen due to code mixing. We assume the following for our worst-case estimate. Firstly, we assume that when interrupts or context switches occur, they flush out useful cache lines that could have been potentially been reused at a later time. Secondly, the number of lines that are flushed out is equal to the number of cache lines that the interrupt service routine occupies in the cache if the size of the interrupt service routine is smaller than the cache size. Otherwise, we assume that the number of cache lines flushed out is the total number of cache lines in the cache. This information is shown in equation [Eq. 4.1](#). For the case of context switches, we assume that the cache context is completely flushed out as shown in [Eq. 4.2](#).

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

A value for the worst-case estimate of misses due to extrinsic interference can thus be obtained using equation [Eq. 4.3](#).

$$damage_{intr} = \max(intr_lines, cache_lines) \quad (\text{Eq. 4.1})$$

$$damage_{ctxsw} = cache_lines \quad (\text{Eq. 4.2})$$

$$\begin{aligned} misses_{ei-wc} &= \sum freq_{intr} \times damage_{intr} \\ &+ \sum freq_{ctxsw} \times damage_{ctxsw} \end{aligned} \quad (\text{Eq. 4.3})$$

This is a worst-case estimate because the actual number of misses for will be lesser than this. This could be because of two factors - the size of the cache and the timing of the interrupts. If the cache is large enough, the amount of flushing would be lesser than what has been assumed because a large cache would be able to accommodate the instructions of the interrupt service routine without flushing out other cache lines. Also, if interrupts happen in close succession then there is also a chance that the cache lines will be reused instead of being flushed out.

In the best case, it is assumed that every cache line will be fully reused before being flushed out due to any intervening interrupt or context switch. Therefore, there would be no misses due to code mixing. The only misses that are incurred would be a one-time penalty of bringing in the cache lines into the cache and misses due to intrinsic interference. These misses are already captured in the estimated number of misses obtained through standalone task profiling.

As can be seen from figure [4.9](#), our average case estimate for the number of misses due to code mixing follow the measured number of misses due to code mixing quite closely. Therefore, the proposed techniques can be relied upon to estimate the number of extrinsic misses in a multitasking application.

CHAPTER 4. CACHE TUNING FOR RTOS-DRIVEN MULTITASKING APPLICATIONS

4.9.a: Code mixing misses for dijkstra + patricia + sha

4.9.b: Code mixing misses for epic + unepic + rawcaudio

4.9.c: Code mixing misses for rawcaudio + cjpeg + sha

4.9.d: Code mixing misses for rawcaudio + djpeg + sha

Figure 4.9: Estimated and actual misses due to extrinsic interference shown in sub-figures 4.9.a, 4.9.b, 4.9.c and 4.9.d

4.7 Summary

In this chapter, we analyzed the control flow in embedded multitasking systems for the purpose of loop profiling and subsequent cache tuning. We showed that although the nature of control flow in multitasking system is pseudo-random, for cache tuning purposes the interspersed execution of user tasks can be ignored and they can be profiled in isolation to obtain information about the nature of the loops in them. This information about the loops could then be used to identify the optimal cache size for an application. We validated the proposed technique by applying it to tune the cache hierarchy for embedded multitasking applications composed of commonly used embedded benchmarks programs. We divided the system execution into the execution of user-tasks/system-calls and other housekeeping function that the RTOS may perform. We profiled these components in isolation, obtained information about all the loops that were executed and were able to identify optimal or near-optimal L1 and filter cache size for all multitasking applications considered. Our proposed framework was shown to achieve significant speed-up as compared to exhaustive cache simulations was well. The proposed techniques was shown to be highly scalable as well because it was be relied upon to estimate the energy delay product estimates for any range of cache size after a one time profiling.

Chapter 5

RTOS models for rapid cache tuning

5.1 Introduction

Rising complexity in modern day embedded applications is leading to pervasive use of RTOS-driven multitasking systems. In such systems, a significant amount of time is spent executing the RTOS code which has a direct impact upon the instruction cache performance of the system. These applications are complicated and even one-time simulation of these applications for the purpose of cache tuning can be time consuming. Additionally, for such applications, the system designer may sometimes want to know the tradeoffs associated with using a different RTOS. Even for the same RTOS, the designer may wish to evaluate the suitability of different options available. For example, from the perspective of instruction cache access energy, using shared memory for inter-process communication may be more efficient than using message queues. This could be because communication using shared memory may be achieved in lesser number of micro-instructions as compared to communication using message queues.

It would be prohibitively time consuming to simulate the multitasking application multiple times to evaluate multiple RTOSes and options within the same RTOS. For the purpose of instruction cache tuning, the RTOS need to be modeled so that instruction cache hit rates may be inferred at design time without multiple simulations. In this chapter, we propose a technique to model the RTOS for rapidly inferring instruction

cache hit rates. Not only does this help in choosing the ideal cache size, but also helps in choosing the best RTOS given a cache size and a multitasking system.

Given the complexity of the operating system, building one single model to relate all hit rates with all possible influences is infeasible. In our modeling techniques, we first divide the RTOS into components and identify the factors or parameters that will affect the instruction cache hit rates for those components. In the second step, we develop test cases to simulate these RTOS components. The test cases are chosen in such a way that multiple combinations of the parameters affecting the instruction hit rates are considered. For the identification of RTOS components and the parameters affecting the hit rates and generation of test cases, we follow techniques similar to [49][50]. In the third step, we use the profiling based hit rate estimation techniques described in chapter 3 to rapidly estimate the hit rate curves for multiple combinations of these parameters. Finally, we construct statistical models that relate these hit rate values to the parameter values. When referenced, these models can estimate the hit-rate for any combination of parameters thereby saving the designer crucial time during design space exploration.

The rest of the chapter is organized as follows. In section 5.2 we discuss how the RTOS can be divided into components for building RTOS models. The factors that affect the instruction cache hit rates for these components and how to capture there factors in the models are discussed next in section 5.3. We describe the broad methodology for modeling the RTOS components in 5.4. Strategies for selection of suitable testcases for simulating the RTOS components are presented in that section. We also discuss how the instruction cache hit rates for the RTOS components can be estimated for multiple cache sizes using techniques described in chapter 3 in that section. We also show how the estimated hit rates can be captured in a statistical model that relates the estimated hit rate to the parameters that influence the hit rate. We present the overview of the modeling framework in section 5.5. In section 5.6, we discuss show results pertaining

to the accuracy of our models as compared to measured hit rates and summarize the chapter in section 5.7.

5.2 Segmentation of the RTOS into components

During the execution of a multitasking application, the RTOS can be invoked in a multitude of ways. In this section we analyze the ways in which the RTOS can be invoked. Based on the analysis, we then segment the RTOS into components for building RTOS models that relate instruction cache hit rates for the component to the factors that influence the hit rates for these components.

5.2.1 System-call driven execution of the RTOS

In multitasking systems, system resources are shared by multiple tasks. Tasks compete with each other for the system resources and accesses to resources like processor time, memory, peripherals, etc need to be arbitrated. System calls are invoked by user tasks to partake in the arbitration process. For example, consider a task that wishes to print a message on the screen. In this case, the tasks would execute a *printf* statement which is a call to the standard C-library. The library would then forward the request to the RTOS. The RTOS is responsible for checking whether the task is privileged enough to print on the screen, whether there are any higher priority tasks already writing or waiting to write on the screen, etc. Thus, system-call-driven execution of the RTOS occurs when user tasks request to achieve transactions with the system resources in a shared-resource environment.

5.2.2 Exception-driven execution of the RTOS

Another way in which the RTOS may be invoked is through interrupts or traps. In a multitasking system, the task for servicing interrupts is delegated to the RTOS because

it usually involves manipulating the system resources at high levels of privilege and sometimes requires rescheduling the tasks. For example, an interrupt from a peripheral like an analog-to-digital converter may signal that data is available for processing. This may require the RTOS to wake up and activate a high priority task that was waiting for the data. Two of the most common forms of interrupts handled by RTOSes are timer interrupts and page faults. Timer interrupts are necessary to update process times and achieve scheduling so that the processor can be shared fairly amongst all tasks. Page faults arise because the main memory is shared by many different tasks and is not large enough to hold all the virtual pages referenced by the tasks. Usually after a context switch, the virtual pages referenced by a task may not be in the main memory and need to be brought into the memory. If the requested page is not found in the main memory, a page fault occurs and is serviced by the RTOS by bringing in the requested page into the main memory.

It must be noted here that unlike system calls, exception-processing by the RTOS is not invoked by user tasks. Exceptions occur based on runtime conditions and automatically trigger invocation of the RTOS. During exception processing, the execution of user tasks is suspended.

5.2.3 A multi-entry-multi-exit view of the RTOS

Building a single model that relates hit rates of the RTOS executions with the factors that influence is infeasible given the complexity of the RTOS. We simplify this process by breaking up the RTOS into components and then building a model for each component. As discussed, any instance of execution of an RTOS component can be classified into the following two categories - system-call-driven execution and exception-driven execution. From the moment a system call occurs or an exception occurs to the moment the control returns to the user task, RTOS code is executed and therefore the RTOS can be

seen as a multi-entry-multi-exit code block. For each entry to the RTOS, there will be a corresponding exit to the user task. Such entry-exit pairs form the basis of our division of the RTOS into components. The RTOS code executed between each entry and the corresponding exit is treated as a component. Based on our discussions above, there are two broad classes of entry-exit points - System Call Entry Exit Point (SCEEP) and Interrupt Entry-Exit Point (IEEP). The execution of functions like *open*, *read*, *write*, *close*, etc fall into the category of SCEEPs while execution of service routines for interrupts or traps falls into the category IEEPs. A similar description of the RTOS was used in [49][50] for macro-modeling energy expended in running the RTOS components.

5.3 Factors influencing hit rates for RTOS components

When RTOS components execute, there are many factors that influence the hit rate. Among these factors, the major ones can be broadly classified into two classes - static factors and dynamic factors. Static factors are the ones that are known at compile-time while dynamic factors are known only at runtime. In this section, we discuss the static and dynamic factors the influence the hit rates for RTOS components.

5.3.1 Static factors - application and cache

The first class of factors are static in nature. They are known at compile-time and remain unchanged throughout the execution of the application. One set of static factors is application-specific in nature. The parameters driving the system-calls are examples of application-specific static factors that influence the instruction cache hit rates while executing RTOS components. Take the case of the function *read(N)* as an example. Here, the nature of execution of this function is dependant on the value of N. For example reading 1kB may need more work to be done by the function as compared to reading 32B.

This would in turn influence the frequency of the code segments directly and consequently the instruction cache hit rate as well. The other set of static factors are cache-specific in nature for e.g. cache size, block size, associativity, etc. For the same application, a larger cache will always lead to a higher cache hit rate.

Figure 5.1: Multivariate nature of instruction cache hits in $\text{read}(x)$

Figure 5.1 shows the number of hits for the function $\text{read}(N)$. It clearly shows that number of hits is dependent on both the cache size C and the size of the parameter for the read function N . This figure captures how application-specific factors and target-specific factors interact to influence the cache hit rate.

5.3.2 Dynamic factors - scheduler

Some of the calls are often accompanied by calls to the scheduler and there can be multiple outcomes of calling the scheduler. This will in turn affect the number of instructions

executed as a part of invoking the component as well as the locality of cache references and hence the hit rate too. The outcome of running the scheduler is only known at runtime and depends on the prevailing runtime conditions. So, the effect of running the scheduler on the cache hit rate is classified as a dynamic factor.

To understand what are the possible scenarios as a result of running the scheduler consider the following example. If a high priority task A is waiting for a message from a lower priority task B, it will be necessary to invoke the execution of task A immediately after task B has finished writing the message (usually through a system call to write to a message queue). In such a case, the scheduler is called after the message has been written by B. After this, the tasks are rescheduled and task A is chosen to be run. In this case, the scheduling as well as the context switch code is run before the chosen new task starts execution. So the hit rate will be different than just running the system call to service the message writing request by task B. Sometimes though, a context switch may not be necessary because no high priority task may be waiting for a message from task B and the control flow returns to the original task. In such a case, only the scheduling part of the scheduler code is executed but not the context switch part.

5.4 Methodology for modeling RTOS components

In this section, we discuss the issues and the broad methodology for modeling RTOS components to obtain a mathematical relationship between factors that influence the instruction cache hit rates of these components and the hit rate itself. We first describe the factors that must be considered in obtaining hit rates for these components and also how to obtain hit rate itself. Then, we describe how the obtained hit rates can be packaged in a model (a function that relates the output to the variables it depends on) that relates the various parameters that affect the hit rate to the obtained hit rate values.

5.4.1 Obtaining hit rates for RTOS components

Having described the factors which affect the hit rates of RTOS components, we now describe strategies for obtaining the hit rates for the RTOS components. The first step in obtaining the hit rates is the identification of suitable testcases to invoke the components. These testcases must be chosen such that they invoke the RTOS in as many different ways as possible. This will increase the reliability of the final model when the hit rates are related to the factors that influence it.

For system calls or application-oriented factors that were discussed in the preceeding section, it is sufficient to choose varied values for the parameters to the system calls. For example, in a system call *calloc*(*x*,*y*), it is necessary to choose multiple and varied combinations of *x* and *y* to obtain instruction cache hit rates.

For the cache-specific factors like block size, cache size and associativity, the techniques described in chapter 3 for rapid estimation of instruction cache hit rates can be relied upon. This involves a one-time simulation of the RTOS components and saves considerable amount of time as compared to exhaustive cache simulation as shown in chapter 3.

Modeling RTOS components which involve calls to the scheduler is more involved because there can be two major outcomes of executing the RTOS components and models must be built to represent each outcome. The first outcome of calling the scheduler is rescheduling of task(s). Rescheduling is the process of calculating task priorities and selecting a task to run based on the calculated priorities. If no other task high priority task is ready to run, then the system just continues executing the current task. Since no new task is selected for execution, there is no context switch in this case. The second outcome is execution of the RTOS call followed by rescheduling and context switch. This outcome will occur if another higher priority task is ready to run.

5.2.a: Timer interrupt and schedule only

5.2.b: Timer interrupt, schedule and context switch

Figure 5.2: Testcase results used for modeling instruction cache performance for timer interrupts

Take the case of the timer interrupt as an example to demonstrate the effect of calling the scheduler on the number of instructions executed by the RTOS component and its hit rate. If there is only one task, there will be multiple timer interrupts and the occasional rescheduling every few timer interrupts when the timeslice for the current task expires. If the number of instructions executed is counted from the time the interrupt happens to the time that the control flow returns to the user task, most of the times, the number of instructions executed will be small which will be the instructions executed to service the interrupt. But once in a while, a large number of instructions will be executed. This will be the sum of the number of instructions executed to service the interrupt and to reschedule the task. On the other hand, in a multitasking system, if another higher priority task is ready to run after rescheduling then the total number of instructions will increase further. This increase will be due to added instructions executed to perform the context switch..

To further demonstrate our point, we ran a set of experiments on the Linux operating

system. For observing the case of schedule-only, we ran a task that does nothing but execute a `while(1);` infinite loop. In this case, there is always only one task ready to run. When its timeslice expires, the task's priority is recalculated by the scheduler and the task starts to execute again. There is no context switch in this case. As seen in figure 5.2.a there are two clusters - A1 for the timer interrupt service routine executions and B1 for the scheduling part which happens every few timer interrupts. The frequency of A1 is clearly higher than B1 but it executes less number of instructions.

For observing context switches, we executed two tasks running the same `while(1);` loop at the same priority level. In this case, the scheduler keeps switching from one task to another at the expiration of each timeslice interval in order to avoid task starvation. The distribution of the instructions executed per timer interrupt is shown in figure 5.2.b for this case. We observe a cluster (marked as A2) similar to A1 in the schedule-only case. This cluster represents the timer interrupts where there is no rescheduling/context switch. However, there is no cluster similar to B1 in this case. Instead, we have a new cluster B2 which is similar in frequency of occurrence to B1 but executes much larger number of instructions. The additional number of instructions executed in B2 as compared to B1 is due to the context switch overheads.

It is evident from the experiments, that multiple models need to be constructed for RTOS components that are accompanied by a call to the scheduler. Therefore, testcases for such components need to be crafted in a way that consider all eventualities of calling the component. These eventualities are schedule without context switch when the original task resumes execution or schedule with context switch when a new task is chosen to be executed. Designing these testcases should be straightforward and ideas to do so can be drawn from the timer interrupt example that was discussed in this section.

5.4.2 Multivariate models for the estimated hit rates

After the hit rate curves have been obtained the next task is to bundle them into a statistical model for future reference. While some of the RTOS calls are non-parametric i.e. there are no application-specific factors influencing the instruction cache hit rate, most of them depend on a number of application-specific factors to determine their behaviour. This was shown for the case of `read(x)` in section 5.3. In this example, for different values of application-specific factors, the number of cache hits obtained were different. Apart from application-specific factors, cache-specific factors also play a role in influencing the hit rates. This is evident from the same example where for the same value of x , different number of hits were observed for different cache sizes.

In view of this, it is imperative to incorporate all the variables which contribute to the overall hit rate in our statistical model. The discussed example is simplistic and in reality, there may be many application and cache specific factors influencing the hit rate. We have a few options on how to go about building models that relate the instruction cache hit rates to the factors that influence it.

The first option to achieve this is traditional linear or nonlinear regression. In this scheme, the dependent variable (which is the hit rate in this case) y_i as follows:

$$y_i = f(\beta, x_i') + \epsilon_i \quad (\text{Eq. 5.1})$$

Here, β is a vector that needs to be estimated using the available data. It is called the parameter vector not to be mistaken with the parameters passed to the RTOS components or the application-specific factors which influence the hit rates. x_i' is the vector of predictors which in our case are the cache-specific and application-specific factors influencing the hit rate and ϵ_i is the fitting error. Such a scheme was used in [49] for modeling operating system energy consumption. The problem in using such an approach is that

the function $f(\cdot)$ relating the response variable to the predictors has to be supplied a priori. Identifying a candidate function can be difficult as it is difficult to know how the predictors affect the response. So, in this work we use non-parametric additive regression scheme instead, as was used to model multiprocessor cache miss rates in [19].

In non-parametric additive modeling, the response variable is expressed as a sum of functions of the explanatory or predictor variables. For example, if y depends on x and z , then an additive model defining the relationship can be written as

$$y_i = f_1(x_i) + f_2(z_i) + \epsilon_i \quad (\text{Eq. 5.2})$$

In such a model, we need to estimate the partial regression functions f_j rather than the values for β as was the case with traditional linear or nonlinear regression schemes. This is usually achieved by a technique called backfitting [67][19]. The major steps in backfitting adapted from [67] are described below. In the following steps, $\bar{y}$ is the mean of the dependent variable y and m is the number of predictor variables in the model.

- (i) Set $\hat{\alpha} = \bar{y}$ and $\hat{\mathbf{f}}_j = \mathbf{0}$ for $j = 1, \dots, m$.
- (ii) Repeat steps (iii) to (v) until estimates $\hat{\mathbf{f}}_j$ stop changing.
- (iii) For $j = 1, \dots, m$. repeat steps (iv) and (v).
- (iv) Calculate the partial residuals: $\mathbf{e}_p^j = \mathbf{y} - \hat{\alpha} - \sum_{k \neq j} \hat{\mathbf{f}}_k$
- (v) Set $\mathbf{f}_j$ equal to the result of smoothing e_p^j with respect to x_j .

Clearly, additive non-parametric regression is a neat solution for our problem because of two reasons. Firstly, it does not require us to think of a candidate function for fitting the data. Secondly, it offers a highly scalable solution to problems where there are multiple predictors as is often the case with hit rates for RTOS components.

5.5 Overview of the RTOS modeling framework

In this section, we discuss the proposed framework for modeling RTOS components for the purpose of instruction cache hit rates estimation. The framework is shown in figure 5.3 and has three major parts.

In the first part, the RTOS is segmented into components and the application-specific factors that affect the hit rates are identified. For example, for the case of $write(x)$, x would be an application-specific factor. Then, testcases are developed to simulate the components for multiple values of the application-specific factors. At this stage, care is taken to consider whether the component calls the scheduler. If the components calls the scheduler then test cases are also developed to consider the different eventualities of calling the scheduler like rescheduling only or rescheduling followed by context switch. At the end of this exercise, a testcase database is built up where there are a set of testcases for each RTOS component.

The second part of this framework deals with simulating these testcases. This is followed by estimation of instruction hit rates for the testcase for multiple cache configurations. The estimation is achieved by using techniques described in chapter 3. At the end of this exercise, we obtain hit rates corresponding to a combination of application-specific and cache-specific factors. For example, in the case of $calloc(x,y)$, we obtain hit rate h corresponding to multiple combinations of x , y , the cache size C , the block size B and associativity A .

The final stage in the framework deals with obtaining a model that relates h in the previous example to x , y , C , B and A . For this purpose, we use additive regression schemes described in the preceeding subsections. To perform additive regression, we use a statistical software called R [67] to obtain the smooth functions of the predictor variables (application-specific and cache-specific parameters like x , y , C , B and A) that

Figure 5.3: Overview of the modeling framework

determine the hit rate for a given RTOS component to express the hit rate as a function of these parameters.

5.6 Experimental results

In this section, we comment on the accuracy of the proposed modeling methodology by comparing it with actual measured values of the instruction cache hit rates. We tested our models for the embedded Linux operating system. Instead of modeling all the components of the operating system, we chose a handful few that were a good representation of the functionalities provided by most modern day embedded operating systems. The

proposed techniques are, however, generic enough to be extended to all components. We wrote testcases for the filesystem, memory manager, inter-procedural communication and process management subsystems. The components tested are listed in table 5.1. For those components whose instruction cache hit rate is influenced by application-specific factors, table 5.1 also shows the range of parameter values tested in bytes. For the sake of simplicity, we have demonstrated only one RTOS component calling the scheduler which is the timer interrupt. Building models for similar components that involve calls to the scheduler is straightforward and ideas developed in this chapter for the timer interrupt can be relied upon for this purpose.

We compared our models with the hit rates obtained for a 2-way set associative cache memories using the DineroIV cache simulator. The cache block size was fixed at 32 bytes and the cache size was varied from 128 bytes to 64 kilobytes.

As can be seen in table 5.1, the proposed techniques are quite accurate as the average difference in hit rates between the model based estimation and the actual hit rates was well within 0.05 for most of the cases tested. Modeling with the measured hit rates obtained using cache hierarchy simulation instead of estimation will lead to more accurate results. However, in such a case obtaining hit rate curves for such a large number of functionalities and parameter combinations would be prohibitively complex and therefore not recommended.

5.7 Summary

In this chapter, we proposed techniques for segmenting the RTOS into components and then building models to predict the instruction cache hit rates for those components. Our approach relies on simulating the components by varying the values of the factors that affect the hit rate of the component. Then, we build statistical models to relate the obtained hit rates values with those factors.

CHAPTER 5. RTOS MODELS FOR RAPID CACHE TUNING

Filesystem		
<i>RTOS functionality</i>	<i>Parameter range</i>	Δhr
Open()	-	0.03
Close()	-	0.01
Read(x)	$x \in \{32,64,\dots,1024\}$	0.00
Write(x)	$x \in \{32,64,\dots,1024\}$	0.01
Memory Management		
<i>RTOS functionality</i>	<i>Parameter range</i>	Δhr
Malloc(x)	$x \in \{32,64,\dots,1024\}$	0.03
Calloc (x,y)	$x \in \{16,32,64\}, y \in \{16,32,64\}$	0.02
Free(x)	$x \in \{32,64,\dots,1024\}$	0.01
Inter-Procedural Communication		
<i>RTOS functionality</i>	<i>Parameter range</i>	Δhr
Msgget()	-	0.05
Msgsnd(x)	$x \in \{32,64,\dots,1024\}$	0.02
Msgrev(x)	$x \in \{32,64,\dots,1024\}$	0.02
Pipe()	-	0.00
PipeRd(x)	$x \in \{32,64,\dots,1024\}$	0.02
PipeWr(x)	$x \in \{32,64,\dots,1024\}$	0.02
Process Management		
<i>RTOS functionality</i>	<i>Parameter range</i>	Δhr
Fork()	-	0.04
TimerISR	-	0.03
TimerISR+Sched	-	0.04
TimerISR+Sched+	-	0.05
ContextSw	-	

Table 5.1: Mean difference in hit rates between the model based and exhaustive simulation based approach

We first discussed how to break up the RTOS into components and also presented the factors that affect the instruction cache hit rates of the RTOS components. Then,

we discussed how these factors should be taken into consideration while obtaining the hit rates for these components. After the hit rates were obtained, we presented non-parametric additive regression techniques to obtain statistical models that relate the obtained hit rates with the factors that influence the hit rate. The suitability of the proposed techniques for modeling were demonstrated select RTOS components that are a good representation of the features offered by most modern day operating systems. The instruction cache hit rates obtained using the models were shown to closely match those obtained using actual simulation of the cache hierarchy.

The techniques proposed in this paper are highly scalable and can be extended to any RTOS components. They can also be relied upon to model instruction cache performance of other operating systems. The hit rate estimation stage can estimate the hit rates for any combination of cache parameters. It is suitably supported by the additive regression modeling module which can support any number of parameters which affect the instruction cache hit rate of the RTOS component, thereby offering a highly scalable solution.

This work can find application in a variety of areas in design space exploration like cache tuning for multitasking systems, comparison of the cache performance of different operating systems and energy modeling of system software.

Chapter 6

A generic framework for cache tuning

6.1 Introduction

In the previous chapters, we developed techniques for cache tuning for single task based and multitask based embedded applications. We also proposed techniques to model RTOS components to observe their impact on the instruction cache hit rates. However, the techniques proposed were geared towards a particular RTOS or target platform. With a large number of real-time operating systems and commercial microcontrollers to choose from, it is necessary for any system-level estimation framework to be extensible to all possible permutations of microcontrollers, operating systems and applications. In this chapter, we present a framework using which it is possible to tune the cache for a multitasking application for any target platform and RTOS combination using a one-time target and RTOS independent profiling.

The rest of the chapter is summarized as follows. We present techniques to estimate the nature of loops in an application for the purpose of cache tuning when it is compiled for any arbitrary target in section 6.2. Our techniques rely target-independent profiling techniques for this purpose and the nature of loops can be inferred for any target platform after a one-time simulation of the application using this technique. Having established a

generic technique which can estimate nature of the loops for any given target, we extend the proposed techniques to include the effect of choosing any arbitrary RTOS in section 6.3. Herein, we present techniques to tune the cache for multitasking applications for any RTOS and target platform pair they may run on after a one-time target independent profiling phase. Finally, we summarize the chapter in section 6.4.

6.2 Target-independent profiling for target-specific loop characteristic estimation

In this section, we describe a technique to estimate nature of loops and consequently the instruction cache hit rates for any target platform without having to run the application on it. This proposed technique is a step towards making the hit-rate estimation and cache tuning framework target-independent.

While it is possible for the underlying micro-instructions to change for program code segments when compiled to different targets, the high-level control flow graph of functions remains the same. This forms the basis of our proposition for rapidly estimating loop characteristics for different target platforms. If the code frequencies can be obtained for the high-level call graph and control flows graph by simulating it on any baseline platform, the application can be compiled to any target and the code size can be inferred from the target instruction set information as well. The code size information combined with the frequency information obtained from the baseline simulation can be used for estimation of cache hit rates and subsequent cache tuning using the techniques described in chapter 3. In the rest of this section, we describe the steps in this target-independent profiling process followed by target-specific loop characteristic estimation. We conclude the section by giving an overview of the framework for target-independent profiling.

6.2.1 High-level representation of the application

The first step in target-independent profiling for target-specific loop characteristic estimation and subsequent cache tuning is the representation of the application using high-level call graph and control flow graph. This information is usually obtained at the programming language level (which is ANSI-C in our case) after passing the application through the compiler front end. The compiler represents the applications in this fashion to run its own front-end optimizations and we leverage on this representation to devise a technique for target-independent profiling for target code size estimation. During this step, the application is divided into high-level basic blocks which each basic block containing snippets of C-code. The control dependencies between these basic blocks is captured in the control flow graphs of the individual functions in the application. The control flow dependency between individual functions is captured in the application's call graph.

6.2.2 Obtaining code frequencies using baseline simulation

A static control flow graph and call graph does not convey much information about the frequency of the code segments in the application. This information is important to us because we use the size and frequency of the loops to estimate the hit rates for multiple cache configurations and perform cache tuning, subsequently. Therefore, after obtaining the static control flow graphs and call graph, the next step in the target-independent profiling is to compile the application to any baseline processor and execute the code to collect information about execution frequencies of different parts of the code. This is necessary to gauge the impact of loops whose boundary is known only at runtime on the instruction cache hit rate. It is a common practice to simulate the application on a target to obtain the execution frequency at the C-code level. Many open-source tools support this with the most commonly used tool being *gcov* and *gprof*. While selecting the baseline processor though, care must be taken to select a target which is able to

meet system performance constraints. Otherwise, the simulation of the application will not be credible and the code frequencies obtained will be unreliable. At the end of the simulation of the application on a baseline processor, a C-code level weighted call graph and weighted control flow graphs of the functions in the application is obtained.

6.2.3 Estimation of code size for a target instruction set

The final step in the framework deals with estimating the size of the frequent code segments to estimate their impact on the cache hit rates. Target instruction sets affect the size of basic blocks in a loop and therefore have a direct impact on the size of the loops themselves. For example, a floating-point instruction may either be executed as a single micro-instruction if a floating point unit is present in the target or it may require several micro-instructions to complete the operation if implemented in software. In this step, the high level control flow graphs and call graphs are passed through the compiler back-end to translate the high level basic blocks in assembly code for the target. Then the instruction set information for the target processor can be used to estimate the code size for the basic blocks. This exercise gives us a target-specific weighted control flow graph which is the input to the cache tuning techniques described in chapter 3.

6.2.4 Overview of target-independent profiling

Figure 6.1 shows the overview of the proposed target-independent profiling framework for target-specific code size estimation and obtaining code execution frequencies. We use an *if-else* code construct as an example to demonstrate the proposed techniques. In the first step, a high-level control flow graph is obtained after passing the code through the compiler front-end. In the second step, the code is compiled to a baseline processor (the micro-instructions for the baseline processor end with `._BL` in figure 6.1) and then executed to obtain the high-level code frequencies. This information when plugged into

Figure 6.1: Overview of target-independent profiling

a target-independent control flow graph, gives us a target-independent weighted control flow graph. In the final step, the code is compiled to any target platform T (microcode instructions for target T end with $_{T}$ in figure 6.1) and then the sizes of the basic blocks can be estimated by taking into account the target instruction set information. This step gives us a target-specific weighted control flow graph. The size and frequency information (obtained using baseline simulation) can then be combined to estimate the hit rate for multiple cache configurations using techniques described in chapter 3 and subsequently, the estimated hit rates can be used for cache tuning.

6.3 A generic tuning framework for multitasking applications

In the preceeding section, we presented a technique to estimate the characteristics of loops in an application for any given target instruction set after a one-time target-independent profiling. While the proposed techniques are good for cache tuning for single-task-based systems, they need to be extended for RTOS driven multitasking systems. This is because cache tuning techniques that are truly generic should take into consideration there are many RTOSes available and the system designer may very well choose to experiment with a few before choosing one as is the case with the target platform. In this section, we present a profiling and tuning framework for RTOS-driven multitasking systems wherein it is possible to consider any possible combination of RTOS and target before performing the cache tuning for that combination after a one-time target independent profiling.

We draw on the ideas discussed in chapter 5 to include variability of the RTOS in our proposed generic framework. In chapter 5, we discussed techniques to model RTOS components to estimate the effect of running them on the instruction cache. For our generic framework, we assume that these models are available for a wide choice of target microcontrollers. Indeed, constructing these models for multiple target platforms should

not be difficult if the techniques described in the preceeding section are used. The thing that should be noted here is that in addition to capturing the impact of the loops in the RTOS components using RTOS models, it will be necessary to build another set of models for estimating the size of system-calls for a given RTOS-target pair. This is necessary because sometimes system calls are invoked from loops in user tasks and their size needs to be added to the size of the loop in the user task to accurately gauge the impact of user task level loops on the instruction cache hit rate and the cache tuning process. Building models for sizes of system calls is fairly straightforward and the techniques described in chapter 5 can be relied upon for this purpose as well.

Figure 6.2 shows the proposed generic framework for tuning the instruction cache for multitasking embedded applications. The framework relies on a target and RTOS independent analysis followed by baseline simulation for identifying the frequency of execution of user code and invocation of RTOS components. After the requisite frequencies are obtained, a target-dependent estimation of the nature of the loops can be done using the techniques described in this chapter. For including the impact of the loops in the RTOS components in the tuning heuristics, RTOS based models described in chapter 5 can be relied upon. The information about the size and frequency of the loops can then be fed into the cache tuning framework for identification of optimal cache sizes.

6.4 Summary

In this chapter, we presented techniques to tune the cache for any arbitrary combination of RTOS and target platform that a multitasking application may run on. The proposed techniques rely on a one-time simulation of the application on any baseline target to obtain information about user code execution frequencies and RTOS component execution frequencies. Then, the application can be compiled to any target platform and information about the instruction set of the target platform can be used to comment on

Figure 6.2: Generic Multitasking Cache Tuning Framework

CHAPTER 6. A GENERIC FRAMEWORK FOR CACHE TUNING

the size of the loops in the application. To include the effect of RTOS invocations on the instruction cache tuning process, pre-built RTOS models that capture the effect of running the RTOS component on that particular platform are relied upon.

For a given RTOS and target platform combination, the proposed techniques can identify the optimal cache configuration without needing to execute the application on the chosen target or simulating the chosen RTOS. Given the myriads of choices available to the system designer, the proposed techniques also offers a feasible way to compare multiple RTOS and target platform combinations from the perspective of instruction cache tuning for a fixed instruction cache configuration.

Chapter 7

Conclusion and Recommendations

7.1 Conclusion

In this research work, techniques were proposed to identify energy-efficient cache hierarchies in a fast and efficient manner.

Existing profile-driven cache tuning schemes were examined closely and their shortcomings were reported. Existing techniques assumed a full cache line utilization while calculating loop sizes which was shown to be untrue through experiments. They ignored the cascading effect of function calls when functions were called from within a loop while calculating loop sizes. Both these shortcomings misappropriated the size of the cache required to hold the loop and led to a wrong choice of cache for the application. Existing techniques were also unable to identify any irreducible loops in the application due to a poor choice of loop profiling algorithms which resulted in some loops being left out of the cache tuning process. We modified the existing techniques to adopt a call graph and control flow graph based representation of the application for capturing the effect of cascaded function calls in calculating loop sizes. Irreducible loops were identified in our techniques by using the dominator-join algorithm which identifies both reducible and irreducible loops. In calculating the loop sizes, we also took into account the amount of cache line utilization for the application by normalizing the loop size with the line utilization factor. This led to a fast and highly accurate framework for instruction cache

CHAPTER 7. CONCLUSION AND RECOMMENDATIONS

hit rate estimation and subsequent cache tuning. The estimated hit rates closely followed measured hit rates for a large range of cache sizes. We applied them to tune a predictor-based filter cache hierarchy and showed that the proposed techniques were able to identify optimal or near optimal filter and L1 cache sizes for all tested benchmarks and were able to achieve speed-up of 63 % to 83% compared to exploration using exhaustive cache simulations for the tested benchmark programs.

Extending the cache tuning schemes for single-task based applications to RTOS-driven multitasking applications posed significant challenges because user tasks and the RTOS shared the cache memory in such applications and jointly impacted the energy/delay of the cache. Analysis of invocations of the RTOS in such applications showed that system calls serviced by the RTOS were likely to be invoked from user-task loops. Therefore, it was proposed that such invocations should be profiled along with user-tasks for accurate loop size estimation. It was also shown that the rest of RTOS invocations could be classified as exception-driven invocations and be treated as any other single-task based application for profiling purposes. Next, we analyzed the effect of task switching on the cache tuning process and showed it to be negligibly small through experiments and theoretical explanations. Based on this observation, we proposed that the instruction cache for multitasking applications could be tuned by profiling the tasks and the RTOS in isolation to derive information about the loops inherent in them and then using this information to tune the cache. For the multitasking applications as well, the proposed techniques were applied to tune the predictor-based filter cache hierarchy for instructions. They were able to identify optimal or near-optimal filter and L1 cache sizes for all applications tested while being able to achieve speed-up of 62% to 89% compared to exhaustive cache simulation techniques for the applications tested.

With the RTOS code being frequently invoked by user-tasks, it became evident that the simulation time for cache tuning can be further reduced by having RTOS models

CHAPTER 7. CONCLUSION AND RECOMMENDATIONS

that capture the impact of RTOS components on cache tuning. Such models for multiple combinations of RTOS and targets could also form the basis of a generalized cache tuning framework wherein it would be possible to select the optimal cache configuration for an application given an RTOS and a target platform. It was observed here that the RTOS was too complicated for a single model to represent the RTOS hit rates. Therefore, the RTOS was partitioned into components where each component was defined as a pair of entry point going into and exit point coming out of the RTOS code. The fact that many of these components called the scheduler at the end of their execution required multiple models to be built for each scheduling outcome. Since the RTOS components exhibited different behaviour for different software and hardware parameters like system-call and cache parameters, we proposed using additive multivariate regression schemes to relate the hit rates to the parameters that affect them. When compared with actual hit rate values measured from cache simulators, the models had an absolute error ranging 0.01% to 0.05% and were able to estimate hit rates for varied combinations of parameter values.

Building generic tuning techniques that enable tuning the cache for any RTOS and target combination entailed extraction of information about the loops in the application for that RTOS and target. While the size of the loop may vary from one instruction set to another, the relative frequency of loops at the programming language level remains the same. Based on this observation, we engineered techniques to collect frequency of the loops and invocation of RTOS components through simulation on a baseline processor. For the size of the loops, it was proposed that the target instruction set could be used as an input to estimate the loop size. To incorporate the impact of the RTOS on the cache tuning process, RTOS models developed for rapid cache tuning were relied upon. Thus, the techniques have been proposed to perform cache tuning for any RTOS and target platform that is chosen to run an application after a one-time simulation and profiling of the application.

7.2 Recommendations for future work

While we were able to propose solutions for many outstanding issues in application-specific cache tuning, there is still scope for future work in this area. We discuss these avenues for future research work in this section.

- Identification of suitable datasets for simulations:** In this research work, we used standard benchmark applications and datasets that came along with the benchmarks for simulating the benchmarks. We assumed that the datasets were good enough to simulate the application and obtain the weighted call graphs to meaningfully tune the instruction cache. However, changing the dataset often changes the way the application executes and therefore it is necessary to work towards universal datasets which are representative of the nature of data the program is bound to encounter. Also, investigation into how the datasets influence the energy delay product measure would be useful as it will provide for additional information to tune the cache hierarchy in a reliable manner.
- Profile driven schemes for data cache selection:** In this work, we tuned the instruction cache by profiling control flow graphs and then identifying a suitable cache configuration based on the nature of loops in the application. It would be interesting to extend this technique to select a suitable data cache based on analysis of data flow in the application. Data flow graphs are analogous to control flow graphs and for all applications that are compiled, data flow graphs are used internally by the compilers for data flow analysis and register allocation. Therefore, locality of reference of data elements or data-reuse clusters in the application may be obtained from weighted data flow graphs in a manner similar to how it was done for the control flow graphs in this work. Unlike the instruction cache, however, the data cache is both written to and read from. The effect of data replacement algorithms

CHAPTER 7. CONCLUSION AND RECOMMENDATIONS

would therefore have to be taken into consideration in the data flow driven analysis. This is a significant undertaking and would be necessary to validate the locality exploration based data cluster identification.

- **Identification and development of suitable multitasking benchmarks:**

Typical embedded systems today are characterized by growing software complexity that necessitates the adoption of RTOS based multitasking environments. For our experiments on multitasking cache tuning, we assumed equi-priority processes scheduled dynamically by the RTOS. The processes used in the experiments were benchmark programs from standard benchmark suites. However, a set of multitasking benchmarks that closely matches behaviour normally encountered in embedded multitasking applications would have provided an accurate insight into the possible energy savings in real-world applications using cache tuning. Therefore, suitable multitasking benchmarks need to be engineered to facilitate the detailed evaluations of the proposed schemes for multitasking environments.

- **Dynamic schemes for directing cache accesses in multitasking systems:**

Since most real-time systems need to be highly responsive, one of the primary requirements for an RTOS is for it to be lightweight so that user tasks can run on the processor for most of the time. This is especially true for exception processing code segments in an RTOS. It is required of the RTOS to finish processing the exception as soon as possible so that the user tasks can resume execution. As a result, RTOS code is often kept short and simple with few loops in them. This characteristic in the RTOS can be exploited in developing dynamic schemes for managing the predictor in predictor-based filter cache hierarchies to direct cache accesses. The cache accesses can be directed to a small cache or line buffer when the RTOS is executing which can result in further savings in energy expended in cache accesses.

- **Dynamic reconfiguration of the cache for reduced energy consumption:**

In this work, we assumed a one-time configuration of the cache prior to executing the applications. It is worth noting that cache requirements of an application can vary during the lifetime of an application. Sometimes, the application may be executing a large loop thereby needing a large cache. At other times, it may suffice to use a smaller cache. Dynamic reconfiguration of the cache would exploit these variations in cache requirements and further reduce the energy spent in accessing the cache. This is a complicated arrangement and requires a substantial amount of further research work. Firstly, accurate algorithms have to be developed to segment the application according to the cache requirements of the loops in it. Secondly, the application has to be instrumented with appropriate code to carry out the tuning process. Finally, efficient hardware schemes for dynamic reconfiguration of the cache have to be researched in order to facilitate fast tuning of the cache without affecting performance of the application.

- **Identification of optimal cache replacement policy:** The focus of our work has been to identify optimal cache sizes assuming an LRU replacement scheme for set-associative cache memory since it is the most commonly used scheme. However, there are other line replacement policies like MRU, pseudo-LRU, etc which can boost cache performance for certain classes of applications. We have not looked into identifying the optimal cache line replacement policies once the optimal cache configuration is identified. Though the cache configuration is the over-riding factor in cache performance, the line replacement policy is also important for the performance of the cache and identification of the optimal line replacement scheme can be incorporated as a part of the tuning process. Therefore, it will be useful to look into this as a future enhancement to our tuning methodology.

References

- [1] A. Agarwal, J. Hennessy, and Horowitz M. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), 1989.
- [2] A.V. Aho and J.D. Ullman. *The theory of parsing, translation and compiling, vol. II: Compiling*. Prentice-Hall, 1972.
- [3] D.H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *32nd Annual International Symposium on Microarchitecture*, pages 248–259, 1999.
- [4] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
- [5] Altera. *Nios Embedded Processor 32-Bit Programmer's Reference Manual*, 2003. http://www.altera.com/literature/manual/mnl_nios_programmers32.pdf.
- [6] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory : A design alternative for cache on-chip memory in embedded systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, pages 73–78. ACM, 2002.
- [7] 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*, page 378, Washington, DC, USA, 1999. IEEE Computer Society.
- [8] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

REFERENCES

- [9] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.
- [10] Steve Carr, Steve Carr, Ken Kennedy, and Ken Kennedy. Compiler blockability of numerical algorithms. In *In Proceedings of Supercomputing '92*, pages 114–124, 1992.
- [11] 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, New York, NY, USA, 1987. ACM.
- [12] C. K. Chow. Determination of cache's capacity and its matching storage hierarchy. *IEEE Transactions on Computers*, 25(2):677–688, 1976.
- [13] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI*, pages 279–290, 1995.
- [14] Krisztián 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, Washington, DC, USA, 2002. IEEE Computer Society.
- [15] Free Software Foundation. *GCC Compiler Documentation*, 2008. <http://gcc.gnu.org/onlinedocs/>.
- [16] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput.*, 5(5):587–616, 1988.
- [17] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *In Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324. ACM Press, 1997.
- [18] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11):1317–1327, 2002.

REFERENCES

- [19] Ilya Gluhovsky and Brian O’Krafka. Comprehensive multiprocessor cache miss rate generation using multivariate models. *ACM Trans. Comput. Syst.*, 23(2):111–145, 2005.
- [20] 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.
- [21] Ann Gordon-Ross and Frank Vahid. A self-tuning configurable cache. In *DAC ’07: Proceedings of the 44th annual conference on Design automation*, pages 234–237, New York, NY, USA, 2007. ACM.
- [22] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th IEEE International Workshop on Workload Characteristics*, pages 3–14, December 2001.
- [23] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. Sh3: high code density, low power. *Micro, IEEE*, 15(6):11–19, 1995.
- [24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.
- [25] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.
- [26] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 273–275, 1999.
- [27] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *25 Years ISCA: Retrospectives and Reprints*, pages 388–397, 1998.
- [28] 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, New York, NY, USA, 2001. ACM.

REFERENCES

- [29] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, pages 184–193, 1997.
- [30] 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.
- [31] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.
- [32] 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.
- [33] 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, New York, NY, USA, 1999. ACM.
- [34] Afzal Malik, Bill Moyer, and Dan Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 241–243, New York, NY, USA, 2000. ACM.
- [35] R. L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
- [36] Kathryn McKinley and Steve Carr. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18:424–453, 1996.
- [37] S. Min, J. Kim, C. Kim, H. Shin, and D. Jeong. V-p cache: A storage efficient virtual cache organization, 1993.

REFERENCES

- [38] J. Montanaro, R.T. Witek, K. Anne, A.J. Black, E.M. Cooper, D.W. Dobberpuhl, P.M. Donahue, J. Eno, W. Hoepfner, D. Kruckemyer, T.H. Lee, P.C.M. Lin, L. Mad-den, D. Murray, M.H. Pearce, S. Santhanam, K.J. Snyder, R. Stehpany, and S.C. Thierauf. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *IEEE journal of solid-state circuits*, 31(11):1703–1714, 1997.
- [39] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.
- [40] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. *Hardware/Software Co-Design, International Workshop on*, 0:67, 2002.
- [41] 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, New York, NY, USA, 1995. ACM.
- [42] 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, New York, NY, USA, 2000. ACM.
- [43] Thomas Roberts Puzak. *Analysis of cache replacement-algorithms*. PhD thesis, University of Massachusetts, 1985.
- [44] Gabriel Rivera and Chau wen Tseng. Data transformations for eliminating conflict misses. In *In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, 1998.
- [45] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

REFERENCES

- [46] 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, New York, NY, USA, 1995. ACM Press.
- [47] R. Sugumar and S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical report, University of Michigan, 1991.
- [48] Rabin Andrew Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs*. PhD thesis, University of Michigan, 1993.
- [49] T. K. Tan, A. Raghunathan, and N. Jha. Embedded operating system energy analysis and macro-modeling. In *in Proc. Int. Conf. Computer Design*, pages 515–522, 2002.
- [50] T. K. Tan, A. Raghunathan, and N. K. Jha. Energy macromodeling of embedded operating systems. *Trans. on Embedded Computing Sys.*, 4(1):231–254, 2005.
- [51] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures, 2001.
- [52] W. Tang, R. Gupta, and A. Nicolau. Power savings in embedded processors through decode filter cache, 2002.
- [53] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [54] R. Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, 1973.
- [55] Tensilica. *Xtensa Processor Generator*, 2005. www.tensilica.com.
- [56] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, 1989.

REFERENCES

- [57] Er V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Ru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *In ICS*, pages 145–154, 1999.
- [58] Luis Villa, Michael Zhang, and Krste Asanovic. Dynamic zero compression for cache energy reduction. In *International Symposium on Microarchitecture*, pages 214–220, 2000.
- [59] Kugan Vivekanandarajah. *Customizable instruction cache hierarchy for embedded systems*. PhD dissertation, Nanyang Technological University, 2005.
- [60] Kugan Vivekanandarajah and Thambipillai Srikanthan. Custom instruction filter cache synthesis for low-power embedded systems. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 151–157, Washington, DC, USA, 2005. IEEE Computer Society.
- [61] Kugan Vivekanandarajah, Thambipillai Srikanthan, Saurav Bhattacharyya, and Prasanna Venkatesh Kannan. Incorporating pattern prediction technique for energy efficient filter cache design. In *IWSOC*, pages 44–47, 2003.
- [62] Kugan Vivekanandarajah, Thambipillai Srikanthan, and Christopher T. Clarke. Profile directed instruction cache tuning for embedded systems. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 277, Washington, DC, USA, 2006. IEEE Computer Society.
- [63] Kugan Vivekanandarajah, Thambipillai Srikanthan, Christopher T. Clarke, and Saurav Bhattacharyya. Static pattern predictor (spp) based low power instruction cache design. In *Embedded Systems and Applications*, pages 210–215, 2003.
- [64] 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, New York, NY, USA, 1990. ACM Press.
- [65] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

REFERENCES

- [66] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded risc architecture. *SIGMICRO Newsl.*, 23(1-2):81–91, 1992.
- [67] Simon N. Wood. *Generalized Additive Models - An Introduction with R*. Chapman & Hall/CRC.
- [68] 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, New York, NY, USA, 1999. ACM Press.
- [69] Xilinx. *MicroBlaze Processor Reference Guide*, v4.0 edition, August 2004. www.xilinx.com/ipcenter/processor_central/microblaze/doc/hwref.pdf.
- [70] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
- [71] C. Zhang and F. Vahid. Cache configuration exploration on prototyping platforms. In *Proceedings of International Workshop on Rapid Systems Prototyping*, 2003.
- [72] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A self-tuning cache architecture for embedded systems. In *Design, Automation and Test Conference in Europe (DATE), 2004*, pages 142–147, 2004.
- [73] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 136–146, New York, NY, USA, 2003. ACM.
- [74] Chuanjun Zhang, Frank Vahid, and Walid A. Najjar. A highly configurable cache for low energy embedded systems. *ACM Transactions in Embedded Computing Systems*, 4(2):363–387, 2005.