The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

ENERGY-AWARE

HARDWARE AND SOFTWARE OPTIMIZATIONS

FOR EMBEDDED SYSTEMS

A Thesis in

Computer Science and Engineering

by

Hyun Suk Kim

© 2003 Hyun Suk Kim

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2003

The thesis of Hyun Suk Kim has been reviewed and approved* by the following:

Vijaykrishnan Narayanan
Assistant Professor of Computer Science and Engineering
Thesis Co-Adviser
Co-Chair of Committee

Mary Jane Irwin
Distinguished Professor of Computer Science and Engineering
Thesis Co-Adviser
Co-Chair of Committee

Mahmut Kandemir
Assistant Professor of Computer Science and Engineering

Richard Brooks
Senior Research Associate

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

* Signatures are on file in the Graduate School.

Abstract

Widespread use of portable embedded systems and rapidly shrinking feature sizes have brought about a large body of research on low power systems design. Embedded systems realize the convergence of computers, communications, and multimedia into portable products, therefore, system designers should have better knowledge of power consumption of the entire system.

This thesis proposes and studies a set of energy evaluation and optimization techniques in architectural and software level for two main components of embedded systems: memory elements and processing elements.

First, the thesis deals with evaluation and optimization of memory systems. Applications are increasingly becoming data-intensive as image processing capabilities are incorporated into portable systems. Since memory accesses consume a significant amount of power, we propose memory energy models for a representative set of on-chip cache architectures and evaluate their energy behavior. As a large volume of data needs to be stored off-chip, we also propose an off-chip memory energy estimation models to enable the designers to evaluate off-chip energy behavior of individual applications.

Second, the study focuses on the energy evaluation and optimization of processing elements. Variants of VLIW architectures are increasingly becoming popular for DSP processors due to their support for wide instruction level parallelism and reduced hardware complexity. An energy simulator for VLIW architectures is developed, which is built on top of a publicly available compilation toolset. Since the compilation toolset has a set of state-of-the-art compilation techniques, we evaluate their energy consumption using the energy simulator, varying architectural parameters. Next, we propose and evaluate a new leakage optimization technique using this toolset. This optimization technique is important as leakage energy is expected to increase exponentially in the next decade.

The tools and techniques proposed in this thesis are expected to be useful for designing energy-efficient embedded systems.

Table of Contents

List of Tables ix

List of Figures xi

Acknowledgments xv

Chapter 1. INTRODUCTION 1

1.1 Outline of the Thesis 4

1.1.1 Evaluation and Optimization of Memory Elements 6

1.1.1.1 Multiple Access Caches: Energy Implications 6

1.1.1.2 Evaluation of Energy Behavior of Iteration Space Tiling 7

1.1.1.3 Estimating Influence of Data Layout Optimizations on SDRAM Energy Consumption 8

1.1.2 Energy Evaluation and Optimization of Processing Elements 8

1.1.2.1 A Framework for Energy Estimation of VLIW Architecture 8

1.1.2.2 Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures 9

Chapter 2. MULTIPLE ACCESS CACHES: ENERGY IMPLICATIONS 11

2.1 Introduction 11

2.2 Cache Energy Models 12

2.3 Experiments 17

2.3.1 Evaluation of Different Cache Architectures 17

2.3.2 Influence of Compiler Optimizations 20

2.3.3 Influence of Technology Changes 26

2.4 Conclusion 27

Chapter 3. EVALUATION OF ENERGY BEHAVIOR OF ITERATION SPACE TILING 29

3.1 Introduction 29

3.2 Loop Tiling 32

3.3 Our Platform and Methodology 35

3.4 Experimental Evaluation 38

3.4.1 Tiling Strategy 38

3.4.2 Sensitivity to the Tile Size 40

3.4.3 Sensitivity to the Input Size 42

3.4.4 Sensitivity to the Cache Configuration 43

3.4.5 Cache Miss Rates vs. Energy 47

3.4.6 Interaction with Other Optimizations 49

3.4.7 Analysis of Datapath Energy 50

3.4.8 Sensitivity to Technology Changes 51

3.4.9 Other Codes 53

3.5 Related Work 56

3.6 Conclusion 57

Chapter 4. ESTIMATING INFLUENCE OF DATA LAYOUT OPTIMIZATIONS ON SDRAM ENERGY CONSUMPTION 59

4.1 Introduction 59

4.2 Preliminaries 61

4.2.1 SDRAM Basics 62

4.2.2 The Polyhedral Model 63

4.3 Page Break Estimation Framework 64

4.4 Blocked Data Layout 70

4.4.1 Overview 70

4.4.2 Page Break Estimation Framework 71

4.4.3 Discussion 75

4.5 Experiments 77

4.5.1 Setup 77

4.5.2 Benchmarks 79

4.5.3 Results 80

4.6 Conclusion 83

Chapter 5. A FRAMEWORK FOR ENERGY ESTIMATION OF VLIW ARCHITECTURES 85

5.1 Introduction 85

5.2 Modeling 86

5.3 Evaluation - Software Optimizations 90

5.3.1 High Level Compiler Optimization 90

5.3.2 Block Formation Algorithms 92

5.3.3 Predication 94

5.4 Evaluation - Hardware Optimizations 96

5.4.1 Multiple Register Banks 96

5.5 Conclusion 98

Chapter 6. ADAPTING INSTRUCTION LEVEL PARALLELISM FOR OPTIMIZING LEAKAGE IN VLIW ARCHITECTURES 102

6.1 Introduction 102

6.2 Related Work 105

6.3 IPC Adaptation 106

6.3.1 Loop Identification 109

6.3.2 IPC Assignment 109

6.3.3 Adaptive Scheduling 112

6.3.4 Hardware Requirements and ISA Support 113

6.4 Experiments and Results 117

6.4.1 Leakage Energy and Performance Impact of Adaptive IPC . . 118

6.4.2 IPC Comparison 120

6.4.3 Granularity Analysis 124

6.4.4 Impact of Leakage Control Mechanism 126

6.5 Conclusions and Ongoing Work 127

Chapter 7. CONCLUSIONS AND FUTURE WORK 129

References 133

List of Tables

2.1 Average energy improvements (%) for data accesses for all benchmarks with a 16KB cache size (a) and for instruction accesses with a two-way set-associative cache (b). 24

3.1 Energy consumption of tiling with different input sizes. `dp`, `ic`, `dc`, and `mem` denote the energies spent in datapath, instruction cache, data cache, and main memory, respectively. 44

3.2 Improvements in miss rate and energy consumption. 49

3.3 Datapath energy breakdown (in %s) in hardware components level. . . 52

3.4 Datapath energy breakdown (in %s) in pipeline stage level. 52

3.5 Benchmark nests used in the experiments. The number following the name corresponds to the number of the nest in the respective code. . . 55

5.1 Non-stall cycles, cache stall cycles, and total operations taken for unoptimized/optimized benchmarks. 92

5.2 Performance and energy numbers of unoptimized `tomcatv` and optimized `tomcatv` as GPR size grows. 100

5.3 Energy (%) of predicate true and false instructions when modulo scheduling is activated. 100

5.4 Maximum number of registers used dynamically. Each column represents number of maximum registers required for local register files and common register file as depicted in Figure 5.5. 101

6.1 Array-intensive benchmark codes used in our experiments. 117

6.2 Normalized EDP (energy-delay product) with different IPC selection schemes (averaged over all applications). All values are normalized with respect to the EDP of the original schedule. LCM means leakage control mode. 126

List of Figures

1.1 Architectural diagram of a typical embedded processor-based system [76]. 5

2.1 Multiple access caches. 14

2.2 Energy consumption of data accesses with varying cache sizes (X-axis represents cache size of 8, 16, 32, 64, and 128KB and Y-axis represents energy consumption of data accesses in Joules). The cache block size is 32B for all configurations and no compiler optimization is used. 19

2.3 Energy improvement (%) in data accesses for different 16KB cache configurations (X-axis represents benchmarks and Y-axis represents energy improvement in percentage). `O1-O0` represents the percentage energy improvement of `O1` optimization level over `O0`, `O2-O0` for `O2` over `O0`, `O3-O0` for `O3` over `O0` (from top to bottom: two-way set-associative cache, direct-mapped cache, CA cache, HR cache, and MRU cache). 23

2.4 Energy improvement (%) of instruction accesses with optimization levels `O1, O2, O3` (X-axis represents cache size of 8, 16, 32, 64, 128KB and Y-axis represents energy improvement in percentage). The cache block size is 32B for all configurations. 25

2.5 Energy consumption of the cache (E_cell) and the main memory (E_main) for Em = 4.95e-9 (a) and Em = 4.95e-10 (b) in `pegwit decoder` benchmark (X-axis represents cache size of 8, 16, 32, 64, and 128KB. For each cache size, five bars represent two-way set-associative, CA, direct-mapped, HR, and MRU caches in order. Y-axis represents energy consumption of data accesses in Joules). No compiler optimization is used. 27

3.1 (a) A matrix-multiply code. (b) Tiled version of (a). (c) Access pattern of the original code. (d) Access pattern of the tiled code. 33

3.2 Cache miss rates for the original (untiled) and tiled matrix-multiply nests. 34

3.3 Energy consumptions of different tiling strategies. 41

3.4 Energy sensitivity of tiling to the tile size (blocking factor). 43

3.5 Impact of cache size and associativity on data cache energy. 47

3.6 Impact of block buffering (`bb`) and sub-banking (`sb`) on data cache energy. 48

3.7 Interaction of loop tiling with loop interchange and unrolling. 50

3.8 Energy consumption with different E_m (J) values. 54

3.9 Normalized energy consumption of example nested loops with two different E_m (J) values. 55

4.1 Diagram of a typical SDRAM. The mode register is used to set the memory access mode to use. If the burst mode is selected, an entire page is buffered. Whether such a buffering is beneficial or not depends strongly on the (page-level) data reuse exhibited by the application. 63

4.2 (a) Row-major memory layout. (b) Block-based memory layout. Note that block layout fits very well in the access pattern of video applications (which is also block based). Such a layout reduces the number of page breaks, thereby reducing energy consumption. 72

4.3 Our simulation setup. Simulated results were compared with our estimations. 77

4.4 Estimated and simulated number of page breaks for phods 80

4.5 SDRAM energy breakdown for three versions of qsdpcm. A tile (block) size of 64×16 was assumed for the block-based code (the last bar for each energy component). 81

4.6 Comparison of the estimated (left bars) and the simulated (right bars) energy consumption for the three benchmarks, qsdpcm, phods, and edge_detect (from left to right). 83

5.1 Simulator block diagram. 87

5.2 Energy distribution of benchmarks without (benchmark_un) and with (benchmark_op) high-level optimizations. 91

5.3 Energy and performance comparison of BB, SB, and HB when modulo scheduling is activated. 94

5.4 Energy distribution when modulo scheduling is activated. Each bar represents relative energy cost for register files, ALUs, caches, and clock circuitry from bottom to top. 95

5.5 The register file architecture evaluated. 98

5.6 Relative energy consumption of register files (including interconnects between register files and functional units). Each bar represents GPR size of 32, 64 and 128, respectively. 99

6.1 IPC variation (`eflux`). 104

6.2 Example DFGs. 108

6.3 (a) Example CFG fragment. (b) Original and modified schedules. . . . 111

6.4 % saved slots ((A-B)/X). This indicates potential for leakage energy reduction using leakage control mechanisms. 121

6.5 % increase in execution cycles. 122

6.6 % leakage energy improvements in IALUs. 123

6.7 Normalized EDP (energy-delay product) with different IPC settings. All values are normalized with respect to the EDP of the original schedule. The I8_L2 configuration was used for all cases. 125

6.8 Normalized EDP (energy-delay product) with different leakage control mechanisms. The I4_L1 configuration was used for all cases. 127

Acknowledgments

At the very end of my stay at Penn State, I feel so lucky to have those who have helped me to be an independent researcher and more mature as a person.

Foremost, I am grateful to my thesis co-advisers, Dr. Vijaykrishnan, and Dr. Irwin, for the guidance they have shown me during my time at Penn State. I am also grateful to Dr. Kandemir, for inspiration on a wide variety of topics, and, particularly, for giving me the chance to work in Belgium. I thank my other committee member, Dr. Brooks, for his insightful commentary on my work.

My days as an MDLer were full of joys thanks to my fellow lab-mates: David, Jeyran, Ben, Byungtae, and others. They shared with me not only offices and academic interests, but also happiness, worries, and other numerous feelings I went through as a graduate student.

The six months I spent in Belgium gave me lots of confidence in my research. My gratitude goes to those who helped my stay there to be fun and rewarding one: Erik, Francky, Nicholas, Gregory, Ilya, Minas, Sashi, and many others.

My dearest Koreans in CSE department, Jinha, Junghye, Eunjung (un-ni), and others, whose names are not mentioned here, have been great friends. I am so thankful for their help, concerns, and the delectable Korean dishes they have shared with me. I also cannot forget to thank Suneuy un-ni, who is soon going to be a mom, for her help and guidance during my earliest years at Penn State.

There were times in depression from doubts of my decisions and capabilities. My friends, Miyoun, and Dr. Choi (un-ni) were great listeners and advisers, as well as racket ball partners and co-painters. Without them, I can not imagine myself at the greatest moment.

My friends in Korea, who I am so looking forward to getting together with, gave me lots of tips in life, and large doses of encouragement when I needed them most.

My sisters, brothers (in-law), and grandma were great supporters from the beginning till this moment. I hope they could stand their big sister (grand-daughter) around them again. My first little nephew, Hyunsoo, gave lots of happiness to his proud aunt. I am so wishful that he could live in a world without any wrongful wars or aggression toward other people or nations.

Finally, I owe my biggest gratitude to my parents, who were there to be my best friends and supporters, when I was in frustration and doubts. They have been and will be my favorite role models in my life.

Chapter 1

INTRODUCTION

With more than 95% of current microprocessors used for battery-powered embedded systems such as cellular phones, personal digital assistants (PDA), and laptop computers, power dissipation has become a key design metric, since the lifetime of batteries is a decisive pricing factor in the market. The power problem has been aggravated since portable electronic systems increasingly demand more complex embedded functionalities, which is enabled by die size increase and rapid shrinking of feature sizes. Excessive heat generated by high clock speed and high density transistors cause reliability problems as well. They require more complicated cooling and packaging. These reasons drive system designers to consider power as well as performance as one of the important criteria in embedded systems design.

The power (P) and energy (E) are defined and categorized as follows:

$$
\begin{aligned}
P &= P_{switch} + P_{sc} + P_{leakage} \\
&= C_L V_{dd}^2 f_{0\rightarrow 1} + t_{sc} V_{dd} I_{peak} f_{0\rightarrow 1} + k_{design} V_{dd} I_{leakage} N_{transistor} \\
E &= C_L V_{dd}^2 P_{0\rightarrow 1} + t_{sc} V_{dd} I_{peak} P_{0\rightarrow 1} + k_{design} V_{dd} I_{leakage} N_{transistor} / f_{clock},
\end{aligned}
$$

where $P_{0\to1} = f_{0\to1}/f_{clock}$, C_L is the capacitive load of the circuit, V_{dd} is the supply voltage, $f_{0\to1}$ is the switching frequency, t_{sc} is the short-circuit time, f_{clock} is the clock frequency, and I_{peak} and $I_{leakage}$ are the peak current during switching and the leakage current, respectively. And k_{design} and $N_{transistor}$ are design specific parameter, and total number of transistors.

Power is composed of dynamic and static elements. P_{switch} and P_{sc} are dynamic; P_{switch} is consumed only when signals transition from 0 to 1, and P_{sc} when either from 0 to 1, or from 1 to 0. Power dissipation may be reduced by reducing any of the parameters. Dynamic power can be reduced by suppressing unnecessary bit transitions. Supply voltage reduction has quadratic effect on dynamic power consumption. However, it might slow down the system since energy is power consumed over time. Thus, the total energy might not be reduced.

The leakage power, $P_{leakage}$, is due to transistor leakage and independent of the switching activity. Power leaks constantly as long as the system is on, while dynamic power consumption varies significantly depending on workload. As the supply and threshold voltage scales down with technology improvements, it will account for as much as half of the total power consumption in deep submicron technologies. One obvious way is to reduce the total number of devices, $N_{transistors}$. This effect can be achieved by turning off devices when not used. However, this technique itself can incur extra power consumption and the latency to recover to normal operation mode is not negligible. To mitigate the latency problem, when the power-gated module is to be used should be predicted well in advance.

There are many abstract levels to work toward power reduction. The levels are not completely disjoint of each other but rather work in conjunction with each other. The lowest process level and the highest algorithmic level are omitted due to their irrelevance to this thesis. They are from bottom to top:

- Circuit/Gate Level

  This abstract level deals with circuit level techniques. Supply voltage and frequency scaling is the most adopted technique at this level, since it yields considerable savings due to the quadratic dependence of power on supply voltage. The major problem of this technique is degradation of circuit speed. To overcome this problem, multiple-voltage and variable voltage techniques have been developed, where timing-critical modules are powered at high voltage level, while the rest of the modules is powered at low voltage level. Leakage energy problem is solved similarly with variable-threshold circuits. Circuits in critical path are maneuvered to low threshold voltage to speed up, whereas those in non-critical path is operated at high threshold voltage to reduce leakage.

- Architectural Level

  In this level of study, larger blocks such as caches and functional units are the main subject. In complex digital circuits, not all of the blocks perform meaningful operations every clock cycle. When a block is identified to be idle, it can be disabled to prevent useless but power consuming transitions. Circuit techniques such as clock gating provide ways to apply this technique. To tackle leakage power, idle blocks can be turned off by power-gating.

- Software Level

  An operation system can achieve major power reduction by performing energy-aware task scheduling and resource management. Processors these days adopt multiple power modes which are initiated by operating systems. These are collectively called dynamic power management (DPM). Another important system component is compilers. Compilers traditionally have been studied to generate efficient codes in terms of performance. Many of the performance optimization techniques also reduce power consumption. For example, spill code reduction result in both performance improvement and power reduction. There have been proposed power optimizing techniques that compromise performance, as well. Power-aware instruction scheduling technique can increase total number of cycles. However, the performance degradation has to be limited.

This thesis evaluates various hardware and software optimization techniques in terms of power to find out power dissipation sources, and proposes power optimization techniques targeting the main sources of power dissipation in architectural and software abstract levels.

1.1 Outline of the Thesis

Figure 1.1 shows a typical architectural model of an embedded processor core-based system, consisting of a processor core, on-chip memory, and off-chip memory [76]. It may also contain other modules such as co-processors and ASIC blocks. Since the entire system is operated by batteries, power problem should be tackled in every element of

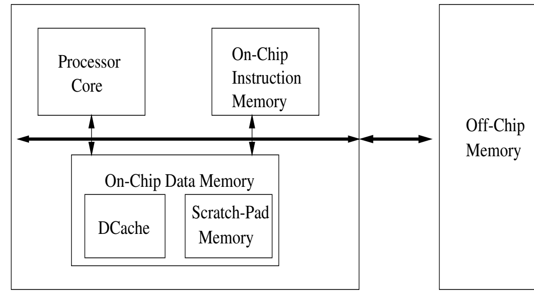


Fig. 1.1. Architectural diagram of a typical embedded processor-based system [76].

the system. The thesis covers two major components of an embedded system – memory and datapath (processor core).

Memories are typically in hierarchical structure: registers, on-chip memories such as caches and scratch-pad memories, and off-chip memories. As applications these days handle large amounts of data, memory elements play an important role in the design of embedded systems. It is estimated that up to ten times of energy is consumed by memory hierarchies [76]. In Chapter 2, multiple access cache architectures are evaluated in terms of energy, using our proposed energy estimation models [61]. In Chapter 3, a very popular compiler performance optimization technique targeting reduction of the main memory latency – loop tiling – is evaluated in terms of energy [52, 51]. Modern off-chip memories have a few interesting features to satisfy increasing bandwidth requirements, such as banking and burst access modes. Their energy and performance characteristics change significantly depending on the types of DRAM operations heavily used to support the application. In Chapter 4, an estimation model to predict the energy consumed in off-chip memory accesses for an application is proposed [60]. This estimation model can

be used to estimate main memory energy as well as performance of various compiler optimization techniques. As an example, a data layout scheme is evaluated to show its off-chip memory energy behavior.

Very Long Instruction Word (VLIW) architectures are getting more and more attention in DSP arena, since they can accommodate many instructions (or operations) per cycle, which is enabled by transferring complex hardware-propelled functionalities into software. In Chapter 5, an architectural energy estimation tool is developed, based on a publicly available VLIW compilation toolset [62]. This enables us to evaluate many VLIW compilation techniques in terms of both power and performance, varying architectural parameters such as the number of functional units and their operation latencies. As technology feature size decreases, leakage power increases dramatically. Leakage power is expected to take up as much as the dynamic power in submicron technologies. In Chapter 6, we propose a loop-level compilation optimization technique to reduce leakage power [63].

The following sections outline the topics in detail.

1.1.1 Evaluation and Optimization of Memory Elements

1.1.1.1 Multiple Access Caches: Energy Implications

On-chip caches have been extensively used to achieve fast memory access latency and to reduce energy by keeping the number of slower and energy-consuming main memory access as small as possible and their sizes are becoming larger as transistor budget increases. There has been much research on circuit design and architectural techniques [9, 49, 41, 64] to improve memory energy consumptions. Most of them were

designed originally to improve performance rather than energy consumption. Multiple access caches were also proposed to improve cache access time. In multiple access caches, the cache banks are accessed multiple times serially rather than once in parallel as in set-associative caches, thereby avoiding the selection logic delays. In this part of the thesis, we propose energy calculation models for these cache architectures and evaluate their energy implications.

1.1.1.2 Evaluation of Energy Behavior of Iteration Space Tiling

It is widely known now that, in addition to efficient hardware techniques, software also plays a major role on energy consumption behavior [85, 94]. This is only natural as the main contributor of overall energy consumption in a given system, namely dynamic energy, is mainly determined by the types and frequencies of switching activities which, in turn, are determined by the software. Consequently, the last couple of years have witnessed a host of studies in application [4, 85], operating system [72, 85], and compiler [85, 21, 53] domains that address this growing energy issue. Given a large body of research in optimizing compilers that target enhancing the performance of a given piece of code [99], we focus on iteration space tiling, a popular high-level (loop-oriented) transformation technique used mainly for optimizing data locality. We evaluated it, with the help of `SimplePower` [104], from the energy point of view, and investigated energy-sensitivity of tiling to tile size and input size varying cache configurations.

1.1.1.3 Estimating Influence of Data Layout Optimizations on SDRAM Energy Consumption

An important problem in extracting maximum benefits from an SDRAM-based architecture is to exploit data locality at the page granularity. Frequent switches between data pages (we call this switch of pages “page break”) can increase memory latency and have an impact on energy consumption. In this part of the thesis, we show that Presburger arithmetic and Ehrhart polynomials can be used for estimating the number of page breaks statically (i.e., at compile time). The result shows that the estimated number of page breaks in SDRAM is close to that obtained by simulation.

We extend the proposed estimation framework to estimate the number of page breaks for a block-based memory layout and investigate the influence of the layout on exploiting page-level locality in memory accesses, and its energy implication. The results obtained using video codes indicate that the proposed memory layouts generate very good energy results, and our estimation can be very useful.

1.1.2 Energy Evaluation and Optimization of Processing Elements

1.1.2.1 A Framework for Energy Estimation of VLIW Architecture

Lack of fast yet reasonably accurate power estimation tools served as a major obstacle for architectural and software level research. Recently, a number of cycle-accurate energy simulators have been developed for simple RISC, DSP, and superscalar architectures [104, 15]. `SimplePower` [104] is an example of such power estimation tools developed in our research group. It is transition-sensitive energy simulator for simple

five-stage pipelined architecture. `Wattch` [15] is a fast architectural simulator based on parameterizable power models for superscalar architectures. VLIW architectures are now becoming popular in embedded systems due to both its architectural simplicity and enlarged number of instruction level parallelism (ILP) enabled by advancements in compiler technology. In this part of the thesis, an energy simulator for Very Long Instruction Word (VLIW) architectures is presented, which is based on analytical models similar to `Wattch` [15].

1.1.2.2 Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures

Since dynamic power is approximately proportional to the square of the supply voltage, the most effective way to reduce power consumption is to lower the supply voltage. However, scaling the supply voltage will adversely affect the performance of the system. Since the propagation delay decreases with the reduction of the threshold voltage, the transistor threshold voltage should also be scaled in order to satisfy the performance requirements. Unfortunately, such scaling leads to an exponential increase in the subthreshold leakage power, which offsets the power savings achieved from reducing the supply voltage. Moreover, leakage energy show exponential dependency on temperature. Therefore, leakage energy reduction for functional units is crucial due to the temperature surge caused by heavy usage of the modules. There exists a great amount of techniques to reduce leakage energy dissipation from less aggressive techniques such as input vector control to very aggressive ones such as gating V_{dd}. More aggressive techniques show better energy savings, while it takes more cycles to apply them. These

techniques should be applied carefully so that the energy savings is not overshadowed by the overheads.

VLIW architectures are greatly dependent on their compilers. For example, mapping operations to the corresponding functional unit properly, considering dependencies (i.e. instruction scheduling), is solely performed by compilers as opposed to superscalar architectures, where the mapping is done by hardware. Therefore, a compilation algorithm is proposed to make more consecutive empty (unused) functional units while instruction scheduling is performed, so that more aggressive leakage energy reduction schemes can be adopted.

Chapter 2

MULTIPLE ACCESS CACHES: ENERGY IMPLICATIONS

2.1 Introduction

With the advent of mobile computing, low power system design has become an important issue. Several hardware-oriented and software-oriented techniques have been proposed to address this problem by minimizing the energy consumption of the various system components. Many of these efforts [92, 48, 64] have focused on the memory subsystem that has been found to be a major energy consumer of the entire system. For example, on-chip caches of DEC 21164 CPU consume 25% of the total chip power [48].

Multiple access caches have been proposed to address the high access latency associated with set-associative caches. In multiple access caches, the cache banks are accessed multiple times serially rather than once in parallel as in set-associative caches, thereby avoiding the selection logic delays. The energy consumption of one such multiple access cache, the MRU (Most Recently Used) way-prediction cache, was evaluated by Inoue et.al. [45]. In their work, the MRU algorithm was used to predict and probe a way first. If the prediction turns out to fail, all remaining ways are accessed at the same time in the next cycle. Many other multiple access cache techniques have been proposed

earlier for performance considerations, such as hash-rehash (HR) caches and column-associative (CA) caches [8, 55, 19]. To the best of our knowledge, no prior research has been done on their energy implication.

In this chapter, we compare three multiple access caches – HR caches, CA caches, and MRU caches – with commonly used set-associative caches and direct-mapped caches in terms of energy. Further, since system level power optimization depends on both the hardware components and the software executing on it, we also investigate the influence of compiler optimizations on the energy-efficiency of these cache architectures. In the next section, the energy models used for the different cache architectures are presented. Then, the experimental results are given and discussed in Section 2.3. Finally, we close with concluding remarks in Section 2.4.

2.2 Cache Energy Models

Associative caches have become commonplace in current processor architectures. While increasing associativity has been used by many researchers to improve performance, it also leads to longer cycle times and more energy consumption. The main reason for longer access times in associative caches is the additional multiplexing logic that selects the data from the correct way. Also, associative caches can affect performance by hindering speculative dispatch of data [105]. The focus of prior research has been to address these problems with associative caches by using multiple access caches [19]. While most of these efforts have investigated the delay aspects, this chapter focuses on the energy efficiency of these cache architectures.

In order to perform our energy evaluation, we build upon the memory system energy model developed in [89] for associative cache architectures. Their model is just for read accesses. We enhanced their energy model based on 0.8 μm technology to incorporate write accesses as well, as shown below:

$Energy = E_{bus} + E_{cell} + E_{pad} + E_{main}$

$E_{bus} = E_{add_bus} + E_{data_bus}$

$E_{cell} = \beta * (Word_line_size) * (Bit_line_size + 4.8) * (Nhit + 2 * Nmiss)$

$E_{pad} = E_{add_pad} + E_{data_pad}$

$E_{main} = Em * 8L * Nmiss * (1 + dirty_r)$

$E_{add_bus} = 0.5e{-}12 * Pr1 * V^2 * (Nhit + Nmiss) * Wadd$

$E_{data_bus} = 0.5e{-}12 * Pr2 * V^2 * (Nhit + Nmiss) * 32$

$E_{add_pad} = 20e{-}12 * Pr3 * V^2 * Nmiss * Wadd$

$E_{data_pad} = 20e{-}12 * Pr4 * V^2 * (1 + dirty_r) * Nmiss * 64,$

and

$Word_line_size = m * (8L + T + St)$

$Bit_line_size = C/(m * L)$

$\beta = 1.44e - 14, Em = 4.95e - 9,$

where C = cache size; L = cache line size; m = set-associativity; T = tag size in bits; St = number of status bits per block; $Nhit$ = number of hits; $Nmiss$ = number of misses; $Wadd$ = the width of a address bus; $dirty_r$ = percentage of blocks written back into memory on replacement.

In this formulation, $Pr1$, $Pr2$, $Pr3$ and $Pr4$ are the bit switch rate for add_bus, $data_bus$,

add_pad, and *data_pad*, respectively. $Pr1$ and $Pr2$ of the data cache are obtained from the simulator execution (explained in Section 2.3), and other bit switch rates were assumed to be 0.25 as in [89]. V denotes the voltage level and is 3.3 Volts. Em is the energy consumed by a main memory access [89].

We further developed energy-models for multiple access caches that try to compromise between low hit ratio and fast access time by accessing the most probable way first and then the second way, if the first access probe fails. If the first probe hits, its access time is the same as that of a direct-mapped cache. It also allows second probe on a miss to the cache unlike direct-mapped caches. In this chapter, we only consider multiple access cache structures for two-way associativity.

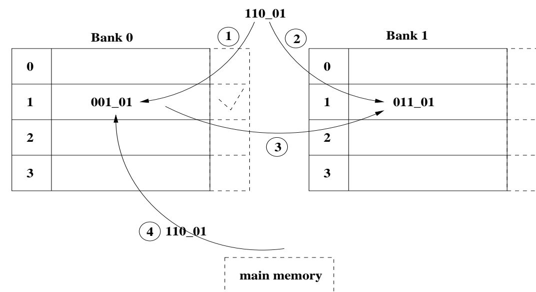


Fig. 2.1. Multiple access caches.

In HR caches [19], the first probe is chosen by a hash function. Hash function is a direct-mapped lookup, which works the same way as in direct-mapped caches. For example, in Figure 2.1, the memory address 11*001* falls into the cache line 1 of bank 0,

while the memory address 11*101* falls into the cache line 1 of bank 1. We define the bank to which the direct-cache mapping function of an address maps as its *home bank*.

In step 1 of Figure 2.1, the cache reference 110_01 falls on cache line 1 of Bank 0 (its home bank). Since the tag bits (i.e., the bits before _) at this location do not match with the tag bits of the reference, rehashing is performed. In case of the HR variant of the two-way set-associativity, rehash function results in probing the other bank. If the rehash succeeds, the two cache lines are swapped, so that the next reference can potentially hit on its first probe. If the rehash fails as in step 2 of Figure 2.1, the data from the first probe is moved to the cache line of the other bank as in step 3, and the data referenced is brought in from main memory to bank 0 as in step 4.

CA caches [8, 19] are almost similar to HR caches except that CA caches can reduce needless probes by keeping a rehash bit for each cache line. The rehash bit is set whenever the cache line is moved from its *home bank* to the other bank. In Figure 2.1, dashed cache line bits are to keep rehash bits for each cache line in the CA cache. If the rehash bit of the cache line probed first is set, the second probe will never hit. Thus, for the example shown in Figure 2.1, the CA cache avoids the second access to the cache and step 3, saving energy and time.

MRU [55, 19] caches, on the other hand, keep a history bit of the most recently used way for each cache line and access the appropriate way based on the history bit value. For example, way 0 is accessed if the corresponding history bit is 0. For a two-way set-associative cache, only a single bit is required.

To account for the operation of the multiple access caches, we modified the memory energy models as follows:

$$E_{cell1} = \beta * (Word_line_size) * (Bit_line_size + 4.8) * (Nfhit + 4 * Nshit + 2 * Nfmiss + 4 * Nsmiss)$$

$$E_{cell2} = \beta * (Word_line_size) * (Bit_line_size + 4.8) * (Nfhit + 2 * Nshit + 3 * Nsmiss)$$

$$Word_line_size = 8L + T + St,$$

where

$Nfhit$ = number of hits on the first probe

$Nshit$ = number of hits on the second probe

$Nfmiss$ = number of misses identified after first probe

$Nsmiss$ = number of misses identified after second probe.

Here, E_{cell1} is energy consumption of HR or CA cache cell accesses and E_{cell2} is that of MRU cache cell accesses. Other equations are the same. $Nfhit + Nshit + Nfmiss + Nsmiss$ is equal to total number of memory references. In case of HR and MRU caches, $Nfmiss$ is 0. This is because only the CA cache can identify a miss after the first probe ($N_{fmiss} \geq 0$) by using the rehash bit. As can be seen above, coefficient m of $Word_line_size$ is 1 as in direct-mapped caches, since only one way is accessed at a time in multiple access caches. A factor of 4 was multiplied to $Nshit$ and $Nsmiss$, and 2 to $Nfmiss$ in CA caches. If there is a hit in the second probe, two extra cache accesses are required to swap the two cache lines in addition to the two probes already done. This results in four cache accesses for second probe hits ($Nshits$) and hence the multiplicative factor 4 in front of the $Nshit$ parameter in E_{cell1}. A similar situation occurs when both probes turn out to fail ($Nsmiss$ cases). In CA caches, useless second way probes can be detected using the rehash bit and no swapping is needed, which accounts for the 2 multiplied to $Nfmiss$. MRU caches do not require swapping action, so only two accesses

are required accounting for the multiplicative factor of 2 for $Nshit$ and three accesses, accounting for the multiplicative factor of 3 for $Nsmiss$.

2.3 Experiments

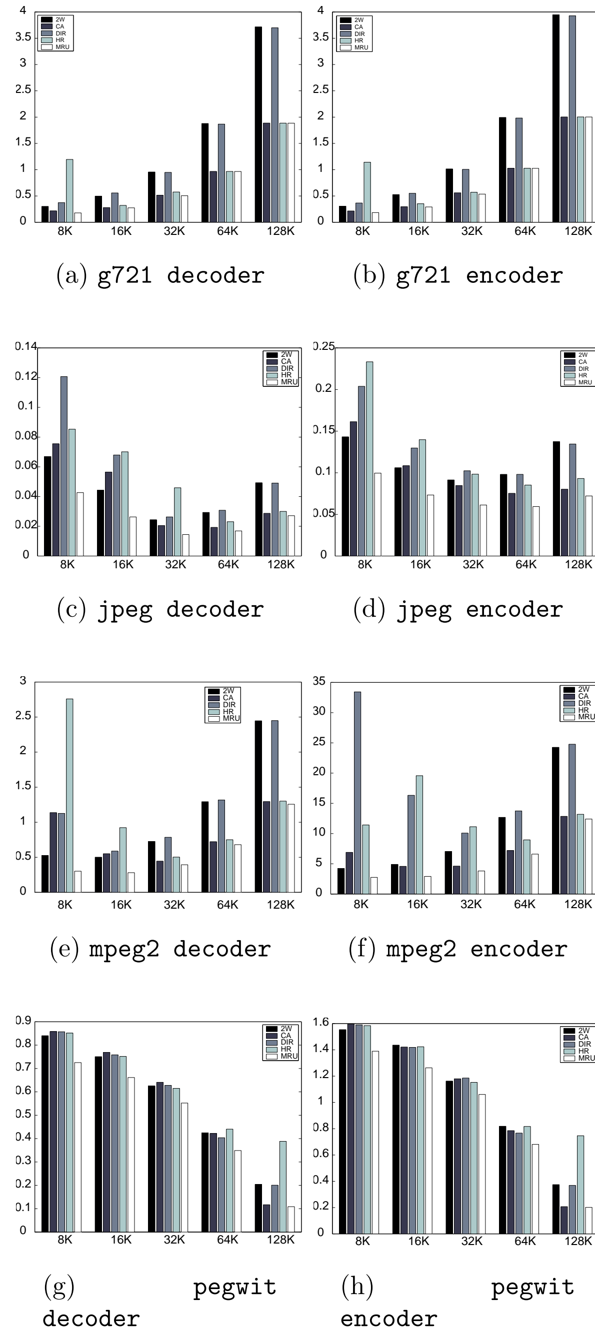
In order to evaluate the energy-efficiency of the various multiple access caches (HR, CA, and MRU), we enhanced the *cachesim5* simulator available in Sun's *Shade* suite [26] and interfaced it with the energy models described in the previous section. A split data and instruction cache, and a block size of 32 bytes were used for all the simulations. The Mediabench benchmark suite [69], a benchmark for multimedia applications, was used in this study.

First, we evaluate the energy consumed by the different cache architectures without using any compiler optimizations. This will provide an insight into energy saving that is inherent to the hardware organizations of these architectures. Then, we introduce the compiler optimizations and discuss the resulting energy improvements brought about by different levels of optimizations. Our study investigates the energy influence of these optimizations considering both instruction and data accesses. We also investigate the impact of emerging eDRAM technology [75] on the energy consumed in the memory system.

2.3.1 Evaluation of Different Cache Architectures

Figure 2.2 shows the energy consumption due to data accesses as cache size increases from 8KB to 128KB for selected benchmarks. It can be observed that MRU caches consume the least energy for all sizes of caches and for all the benchmarks. When

the cache size is small, most of the energy is consumed by memory accesses in all cache models, but as the cache size is increased, most of the energy is consumed by cache accesses. The percentage of memory system energy consumed by the cache cells in the two-way set-associative cache increases more rapidly than that of the CA cache with increasing cache sizes. This indicates that the energy saving of multiple access caches becomes more important when the number of cache misses decreases. As the impact of larger energy consumption due to cache misses declines with decreasing miss rates, the total memory system energy saving of the CA cache gets ahead of the two-way set-associative caches. For example, CA cache in `jpeg decoder` consumes more energy than two-way set-associative cache when the cache size is 8KB and 16KB, but consumes less energy when the cache size grows to 32KB. We observed a similar trend for all the Mediabench benchmarks with increasing cache size. However, the cache sizes beyond which the CA cache performs better is different from benchmark to benchmark. It can also be observed that the HR caches consume considerably more energy than the other multiple access caches for some benchmarks. This is due to both the high miss ratio and extra energy expended for useless second probes and swaps that is avoided by CA caches.



(a) `g721 decoder` (b) `g721 encoder`

(c) `jpeg decoder` (d) `jpeg encoder`

(e) `mpeg2 decoder` (f) `mpeg2 encoder`

(g) `pegwit decoder` (h) `pegwit encoder`

Fig. 2.2. Energy consumption of data accesses with varying cache sizes (X-axis represents cache size of 8, 16, 32, 64, and 128KB and Y-axis represents energy consumption of data accesses in Joules). The cache block size is 32B for all configurations and no compiler optimization is used.

2.3.2 Influence of Compiler Optimizations

Optimizations In this section, we evaluate the impact of classical compiler optimizations on the energy consumption of the cache architectures considered. To accomplish this, we decided to run the programs in our benchmark suite with different optimization flags (levels) as explained below:

- `O1`

  With this optimization flag, we try to reduce size and execution time of the resulting code. The optimizations enabled can be considered collectively as peephole optimizations, where, at a point in time, a short sequence of target instructions is examined and replaced by a faster and/or shorter sequence. They include redundant instruction elimination, algebraic simplifications, and use of machine idioms. On machines that have delay slots, the delayed branch optimizations are also performed at this level.

- `O2`

  With this flag, in addition to the optimizations listed above, nearly all supported compiler optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example (as they increase code size). As compared to `O1`, this option increases both compilation time and the performance of the generated code. The optimizations performed include induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, tail call elimination, and complex expression expansion.

- `O3`

  This optimization performs like `O2` but, also optimizes references or definitions for external

variables. Loop unrolling, inlining, and software pipelining are also performed at this level. Note that, in general, the `O3` level may result in increased code size.

Energy consumption of data accesses Figures 2.3 shows the energy consumption improvement for data accesses when the compiler optimizations explained above are applied to traditional caches and multiple access caches. A 16KB data cache was used in this experiment and `O1-O0` represents the percentage energy improvement of `O1` optimization level over `O0`, `O2-O0` for `O2` over `O0`, `O3-O0` for `O3` over `O0`. We note that, for the benchmarks such as `adpcm decoder/encoder`, `g721 decoder/encoder` and `mpeg2 decoder/encoder`, compiler optimizations reduce energy consumption by about 60 $\sim$ 90%. In these benchmarks, the number of data references reduces dramatically with compiler optimization, whereas the number of misses remains almost the same. For example, the number of two-way set-associative cache data accesses decreased from 6,116,881 to 527,120 when `O3` optimization level was applied.

For the benchmarks such as `mesa mm/tg` (`gsm decoder/encoder` with HR cache), energy consumption increased because the number of data references decreased very slightly, while the number of misses increased with compiler optimizations. For example, in `mesa mm`, the number of data references was reduced from 40,800,534 to 37,031,303, while the number of cache misses increased to 221,686 from 215,910 with `O1` optimization.

Further, we observe that for most of the benchmarks, aggressive optimizations such as `O2` and `O3` do not significantly reduce energy as compared to the `O1` optimization. From table 2.1.`(a)`, which summarizes the percentage improvement provided by our optimization levels on the five different cache architectures over all benchmarks, three

out of five cache architectures, the `O2` optimization resulted in the best performance from the energy perspective.

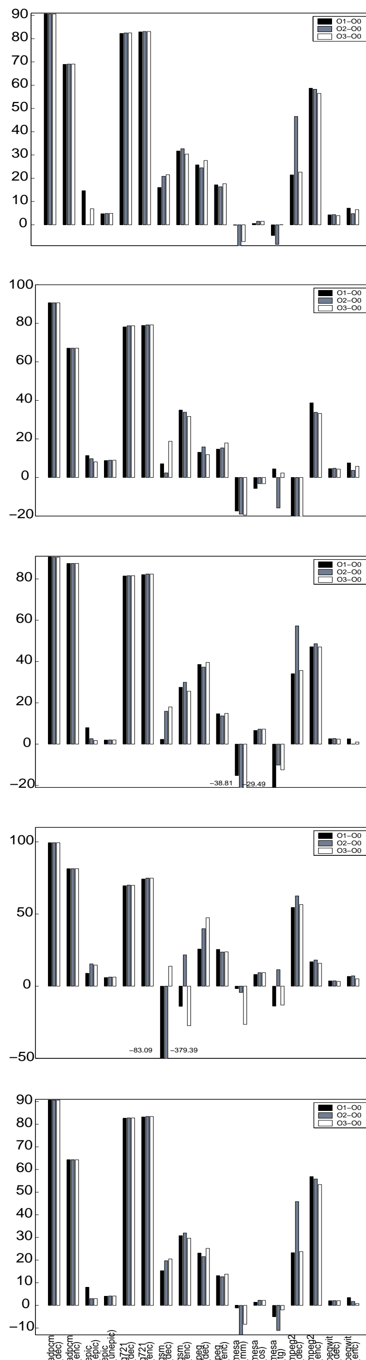


Fig. 2.3. Energy improvement (%) in data accesses for different 16KB cache configurations (X-axis represents benchmarks and Y-axis represents energy improvement in percentage). `O1-O0` represents the percentage energy improvement of `O1` optimization level over `O0`, `O2-O0` for `O2` over `O0`, `O3-O0` for `O3` over `O0` (from top to bottom: two-way set-associative cache, direct-mapped cache, CA cache, HR cache, and MRU cache).

	O1	O2	O3
2W	30.72	30.63	30.48
CA	28.86	30.03	29.14
DIR	10.18	21.71	12.38
HR	21.66	9.48	26.75
MRU	29.1	29.29	28.79

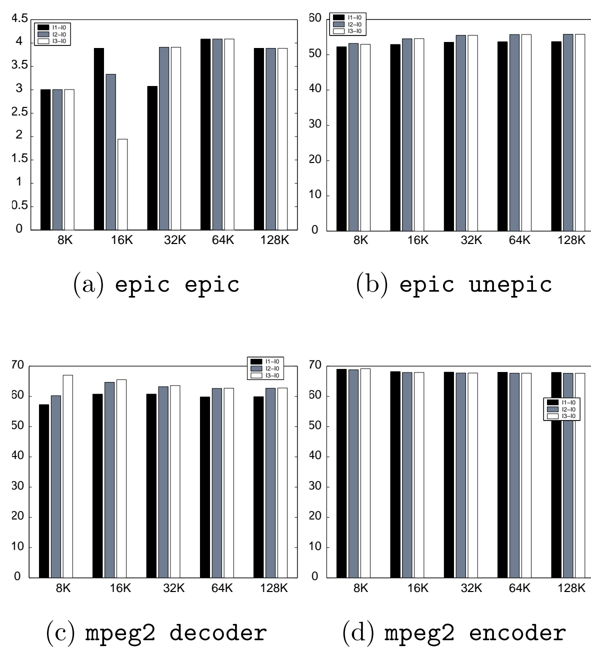
(a)

	O1	O2	O3
8K	42.67	42.70	44.46
16K	41.43	42.65	42.67
32K	40.08	40.66	40.94
64K	40.09	40.58	40.79
128K	40.09	40.66	40.86

(b)

Table 2.1. Average energy improvements (%) for data accesses for all benchmarks with a 16KB cache size (a) and for instruction accesses with a two-way set-associative cache (b).

Energy consumption of instruction accesses Figure 2.4 shows the energy consumption improvement due to instruction accesses for selected benchmarks when instruction cache sizes are varied. In all the experiments, a two-way set-associative cache was used. All benchmarks but `epic epic` and `mesa mm/os/tg` exhibit energy saving of up to 90% with compiler optimizations. Aggressive optimizations such as `O2` or `O3` do not make much difference in terms of energy saving. This saving is mostly caused by reduced instruction references. For example, the number of instruction references is reduced from 425,300,599 to 170,198,103 in `mpeg2 decoder` when `O1` optimization level is used. It must be noted that no benchmark shows any performance degradation (as opposed to those observed for data accesses). Table 2.1.`(b)`shows average instruction access energy improvement over all benchmarks for each of the cache sizes and optimization levels.



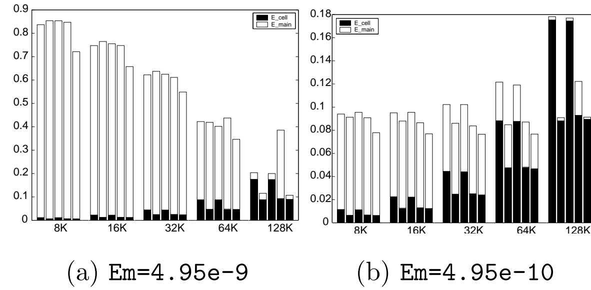
(a) `epic epic` (b) `epic unepic`

(c) `mpeg2 decoder` (d) `mpeg2 encoder`

Fig. 2.4. Energy improvement (%) of instruction accesses with optimization levels `O1, O2, O3` (X-axis represents cache size of 8, 16, 32, 64, 128KB and Y-axis represents energy improvement in percentage). The cache block size is 32B for all configurations.

2.3.3 Influence of Technology Changes

With improvements in process technology such as the capability to have an embedded DRAM on a chip, the impact of the various cache configuration on the memory system energy could vary. In order to study this impact, the energy consumption of a main memory access (Em in the model) was reduced to one-tenth of our initial value for external memory accesses. This assumption was made based on the energy reductions observed in embedded DRAM accesses in [75]. We observe that if the embedded DRAM is exploited, the merit of the two-way set-associative cache is diminished. Figure 2.5 shows the energy consumption of cell accesses and embedded DRAM accesses of each cache model with varying cache sizes in `pegwit decoder`. As the cache size increases, the energy consumption in the embedded DRAM becomes negligible and the cache energy consumption dominates. This shows that small multiple access caches such as CA caches between a processor and a large embedded DRAM could save cache energy.



(a) `Em=4.95e-9` (b) `Em=4.95e-10`

Fig. 2.5. Energy consumption of the cache (E_cell) and the main memory (E_main) for Em = 4.95e-9 (a) and Em = 4.95e-10 (b) in `pegwit decoder` benchmark (X-axis represents cache size of 8, 16, 32, 64, and 128KB. For each cache size, five bars represent two-way set-associative, CA, direct-mapped, HR, and MRU caches in order. Y-axis represents energy consumption of data accesses in Joules). No compiler optimization is used.

2.4 Conclusion

Due to their critical role in improving the performance of the memory system, caches have become an important system component. As a result, different cache architectures have emerged including multiple access caches whose main purpose is to reduce the access latencies of associative caches. In this chapter, we investigate three different multiple access caches from the energy perspective using the Mediabench benchmark suite. Our results indicate that a memory system with a MRU cache consumes the least energy among all cache configurations and multiple access caches can save energy when applications show inherently low cache miss ratio or cache size becomes bigger. Further, it is observed that the energy reductions obtained by using the multiple access caches can

become more important as the cost of main memory access is reduced by the emerging eDRAM technology.

On the software part, our observation is that compiler optimization can significantly reduce the memory system energy across all cache architectures, although direct-mapped caches and HR caches show less energy saving than other cache configurations. However, the most aggressive optimizations (obtained using `O3` optimization level in our experiments) do not necessarily lead to the most energy efficient code. Thus, software system designers need to experiment with different optimization levels before deciding on the most energy efficient solution. We also find that, while the optimizations do not always reduce the energy consumed by data accesses, the energy consumed by instruction accesses for the Mediabench benchmark suite is always reduced by optimizations.

Chapter 3

EVALUATION OF ENERGY BEHAVIOR OF ITERATION SPACE TILING

3.1 Introduction

Energy consumption has become a critical design concern in recent years, driven by the proliferation of battery-operated embedded devices. While it is true that careful hardware design [16, 36] is very effective in reducing the energy consumed by a given computing system, it is agreed that software can also play a major role [85, 34, 97, 59, 58]. In particular, the application code that runs on an embedded device is the primary factor that determines the dynamic switching activity, one of the contributors to dynamic power dissipation.

Given a large body of research in optimizing compilers that target enhancing the performance of a given piece of code (e.g., see [99] and the references therein), we believe that the first step in developing energy-aware optimization techniques is to understand the influence of widely used program transformations on energy consumption. Such an understanding would serve two main purposes. First, it will allow compiler designers to see whether current *performance-oriented* optimization techniques are sufficient for minimizing the energy-consumption, and if not, what additional optimizations are needed. Second, it will give hardware designers an idea about the influence of widely

used compiler optimizations on energy-consumption, thereby enabling them to evaluate and compare different energy-efficient design alternatives with these optimizations.

While it is possible and certainly beneficial to evaluate each and every compiler optimization from energy perspective, in this chapter, we focus our attention on *iteration space* (*loop*) *tiling,* a popular high-level (loop-oriented) transformation technique used mainly for optimizing data locality [99, 101, 100, 70, 67, 102].[1] This optimization is important because it is very effective in improving data locality and it is used by many optimizing compilers from industry and academia. While behavior of tiling from performance perspective has been understood to a large extent and important parameters that affect its performance have been thoroughly studied and reported, its influence on system energy is yet to be understood. In particular, its influence on energy consumption of different system components (e.g., datapath, caches, main memory system, etc.) has to be explored in detail.

Having identified loop tiling as an important optimization, in this chapter, we evaluate it, with the help of our cycle-accurate simulator, *SimplePower* [104, 96], from the energy point of view considering a number of factors. The scope of our evaluation includes different tiling styles (strategies), modifying important parameters such as input size and tile size (blocking factor) and hardware features such as cache configuration. In addition, we also investigate how tiling performs in conjunction with two recently-proposed energy-conscious cache architectures, how current trends in memory technology will affect its effectiveness on different system components, and how it interacts with

[1] Loop tiling can also be used for optimizing loop-level parallelism [46, 79].

other loop-oriented optimizations as far as energy consumption is concerned. Specifically, in this chapter, we make the following contributions:

- We report the energy consumed for different styles of tiling using a matrix-multiply code as a running example.
- We investigate energy-sensitivity of tiling to tile size and input size.
- We investigate its energy performance on several cache configurations including a number of new cache architectures and different technology parameters.
- We evaluate the energy consumption and discuss the results when tiling is accompanied by other code optimizations.

Our results show that while tiling reduces the energy spent in main memory system, it may increase the energy consumed in the datapath and on-chip caches. We also observed a great variation on energy performance of tiling when tile size is modified; this shows that determining optimal tile sizes is an important problem for compiler writers for power-aware embedded systems. Also, tailoring tile size to the input size generates better energy results than working with a fixed tile size for all inputs.

The remainder of this chapter is organized as follows. In Section 3.2, we review loop tiling briefly and report some performance numbers showing its usefulness in optimizing data locality. In Section 3.3, we introduce our experimental platform and experimental methodology. In Section 3.4, we report energy results for different styles of tiling using a matrix-multiply code as a running example and evaluate the energy sensitivity of tiling with respect to software and hardware parameters and technological

trends. In Section 3.5, we discuss related work and conclude the chapter with a summary in Section 3.6.

3.2 Loop Tiling

Loop tiling (also called *blocking*) is an essential technique used to improve data locality. While it can be used to exploit data reuse for any level in a given memory hierarchy, we focus here on tiling for optimizing cache performance. The idea is that data structures that are too big to fit in the data cache are divided into smaller pieces (called *blocks* or *tiles*) that fit in the cache. In other words, instead of operating on individual elements of arrays, tiling performs computation on blocks. Consider the matrix-multiply code given in Figure 3.1(a). As long as the arrays a, b, and c accessed by this code fit in cache memory, the performance of this loop nest can be expected to be good. The problem occurs when the total size of these arrays is larger than the cache capacity. Note that for computing an element $c[i][j]$, we need the ith row of array a and the jth column of array b (as shown in Figure 3.1(c)) and for each row of array c, we need to traverse all elements of array b. Unless we have a very large cache that is able to hold the entire b array, the elements of this array will be moved back and forth between main memory and cache. Now let us assume that we tiled this loop (and selected the tile size) as shown in Figure 3.1(b) such that a block of c is calculated by taking the product of a block-row of a with a block-column of b. In this case, if we can hold these three blocks in cache at the same time, for the entire computation of the block of c, we can reuse the elements in the column-block of b, thereby reducing the memory traffic and improving the cache locality substantially (Figure 3.1(d)).

```
for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  for(k=0;k<n;k++)
    c[i][j]+=a[i][k]*b[k][j];
```
(a)

```
for(it=0;it<n;it+=t)
 for(jt=0;jt<n;jt+=t)
  for(k=0;k<n;k++)
   for(i=it;i<min(it+t,n);i++)
    for(j=jt;j<min(jt+t,n);j++)
      c[i][j]+=a[i][k]*b[k][j];
```
(b)

array c　　array a　　array b

(c)

array c　　array a　　array b

(d)

Fig. 3.1. (a) A matrix-multiply code. (b) Tiled version of (a). (c) Access pattern of the original code. (d) Access pattern of the tiled code.

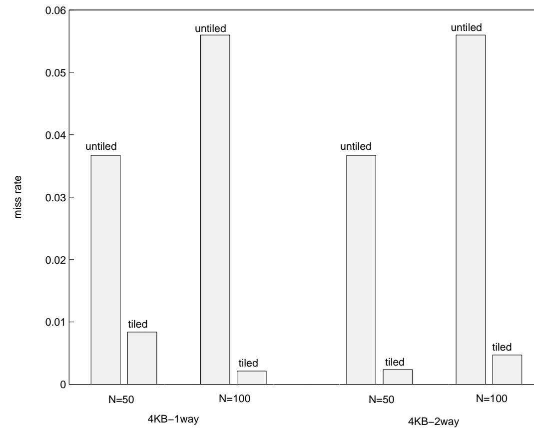


Fig. 3.2. Cache miss rates for the original (untiled) and tiled matrix-multiply nests.

One can easily see that the bigger the blocks are, the greater the reduction in memory traffic but only up to a point. Note that here the three blocks (one from each array) must fit into cache at the same time, therefore there is an upper bound on how big the blocks can be. In addition, the blocks should not conflict with themselves or with one another, that is, different elements of the blocks must not map to the same cache lines, otherwise they will need to be read multiple times (depending on the extent of this conflict) from memory during the course of execution. These conflicts can be reduced or eliminated by appropriate padding of the leading dimension of the arrays [82], careful alignment (layout) of the array in memory with respect to one another [99], copying individual blocks to consecutive memory locations [93], further limiting of the block (tile) size [67], or by using a combination of these. Also, as we make the blocks larger, we will need more TLB (Translation Look-aside Buffer) entries to map their data. If the blocks become too large, the TLB can thrash. All these considerations led to a

large body of research in selecting optimal tile sizes for a given computation and input size [27, 67, 35, 99].

Figure 3.2 shows how tiling improves performance of our matrix-multiply nest. The figure gives the data cache miss rates for the untiled version (Figure 3.1(a)) and the tiled version (Figure 3.1(b)) for two different input sizes (N=50 – 50×50 integer matrices – and N=100 – 100×100 integer matrices –) and two different cache configurations (4KB, 1-way and 4KB, 2-way). The results show significant decreases in miss rates when tiling is used.

3.3 Our Platform and Methodology

The experiments in this chapter were carried out using the *SimplePower* energy estimation framework [96, 104]. This framework includes a transition-sensitive, cycle-accurate datapath energy model that interfaces with analytical and transition-sensitive energy models for the memory and bus sub-systems, respectively. The datapath is based on the ISA of the integer subset of the SimpleScalar architecture [17] and the modeling approach used in this tool has been validated to be accurate (average error rate of 8.98%) using actual current measurements of a commercial DSP architecture [25]. The memory system of *SimplePower* can be configured for different cache sizes, block sizes, associativities, write and replacement policies, number of cache sub-banks, and cache block buffers. *SimplePower* uses the on-chip cache energy model proposed in [48] using 0.8μ technology parameters [98] and the off-chip main memory energy per access cost

based on the Cypress SRAM CY7C1326-133 chip.[2] The analytical on-chip cache models have been found to be within 2% error of measurements from actual designs [48]. The overall energy consumed by the system is calculated as

$$\mathtt{E}_{\mathtt{system}} = \mathtt{E}_{\mathtt{datapath}} + \mathtt{E}_{\mathtt{Icache}} + \mathtt{E}_{\mathtt{Dcache}} + \mathtt{E}_{\mathtt{main\ memory}} + \mathtt{E}_{\mathtt{buses}},$$

where $\mathtt{E}_{\mathtt{datapath}}$ is the energy consumed in the five-stage pipeline processor, $\mathtt{E}_{\mathtt{Icache}}$ and $\mathtt{E}_{\mathtt{Dcache}}$ are the energies consumed in the instruction cache (Icache) and data cache (Dcache), respectively, and are evaluated analytically by summing the energy consumed in the word lines, bit lines, and address input lines. $\mathtt{E}_{\mathtt{main\ memory}}$ is the energy expended on main memory accesses and is evaluated using a *per memory access energy* of $\mathtt{E}_{\mathtt{m}} = 4.95 \times 10^{-9}$J, which is representative of current technology. It must be noted that based on the memory technology (e.g., SRAM, eDRAM, DRAM, etc.), the number of memory modules used, the low power modes supported by the memory chip, and the memory bank configuration, the $\mathtt{E}_{\mathtt{m}}$ value could change. Finally, $\mathtt{E}_{\mathtt{buses}}$ is the energy dissipated when driving interconnect lines external to the cache toward the datapath side or the main memory side. In our design, the datapath and instruction and data caches are assumed to be in a single package and the main memory in a different package. We input our C codes into this framework to obtain the energy results.

Our experimental methodology is as follows. We first evaluate different tiling strategies for our matrix-multiply code using two different input sizes to see the energy

[2] It must be mentioned that SRAMs and embedded DRAMs (eDRAM) are popular choices as main memory in many embedded devices.

benefits of these strategies on different system components. All the tiled codes in this chapter are obtained using an extended version of the source-to-source optimization framework discussed in [50]. Each tiled version is named by using the indices of the loops that have been tiled. For example, `ij` denotes a version where only the i and j loops have been tiled. Unless otherwise stated, in both the *tile loops* (i.e., the loops that iterate over tiles) and the *element loops* (i.e., the loops that iterate over elements in a given tile), the original order of loops is preserved except that the untiled loop(s) is (are) placed right after the tile loops and all the element loops are placed into the innermost positions. Note that the matrix-multiply code is fully permutable [67] and all styles of tilings are legal from the data dependences perspective. We also believe that the matrix-multiply code is an interesting case study because (as noted by Lam et al. [67]) locality is carried in three different loops by three different array variables. However, similar data reuse patterns and energy behaviors can be observed in many codes from the signal and video processing domains.

We then investigate the energy sensitivity of the tiled codes to the tile size. For a given input size, we experiment with five different tile sizes; this allows us to capture the points beyond which reduction in energy is replaced by an increase in energy. After experimenting with different input sizes, we modify cache parameters and evaluate the impact of tiling on two recently-proposed cache architectures (block buffering and cache sub-banking). Then, we compare the variations (due to tiling) in energy with the variations in miss rates. After that, we study the interaction of tiling with two other widely-used loop-based optimizations (loop permutation and loop unrolling) and present a detailed breakdown of datapath energy. We then measure the influence of tiling on

the energy consumption considering the current trends in memory technology. Finally, we measure the energy performance of tiled versions of seven other nested loops.

In all the experiments, our default data cache is 4 KB, one-way associative (direct-mapped) with a line size of 32 bytes.[3] The instruction cache that we simulated has the same configuration. All the reported energy values in this chapter are in Joules (J).

3.4 Experimental Evaluation

3.4.1 Tiling Strategy

In our first set of experiments, we measure the energy consumed by the matrix-multiply code, tiled using different strategies. The *last two graphs* in Figure 3.3 show the total energy consumption of eight different versions of the matrix-multiply code (one original and seven tiled) for two different input sizes: N=50 and N=100. We observe that (for both the input sizes) tiling reduces the overall energy consumption of this code.

In order to further understand the energy behavior of these codes, we break down the energy consumption into different system components: datapath, data cache, instruction cache, and main memory. As depicted in the *first* two graphs in Figure 3.3, tiling a larger number of loops in general increases the datapath energy consumption. The reason for this is that loop tiling converts the input code into a more complex code which involves complicated loop bounds, a larger number of nests, and macro/function calls (for computing loop upper bounds). All these cause more branch instructions in the resulting code and more comparison operations that, in turn, increase the switching activity (and

[3] Embedded processors typically have much smaller caches than those found in traditional processors [43].

energy consumption) in the datapath. For example, when the input size is 50 (100), tiling only the i loop increases the datapath energy consumption by approximately 10% (7%).

A similar negative impact of tiling (to a lesser extent) is also observed in the instruction cache energy consumption. As can be seen from Figure 3.3, when we tile, we have a higher energy consumption in the instruction cache. This is due to the increased number of instructions accessed from the cache due to the more complex loop structure and access pattern. However, in this case, the energy consumption is not that sensitive to the tiling strategy, mainly because the small size of the matrix-multiply code does not put much pressure on the instruction cache. We also observe that the number of data references increase as a result of tiling. This causes an increase in the data cache energy. We speculate that this behavior is due to the influence of back-end compiler optimizations when operating on the tiled code.

When we consider the main memory energy, however, the picture totally changes. For instance, when we tile all three loops, the main memory energy becomes 35.6% (18.7%) of the energy consumed by the untiled code when the input size is 50 (100). This is due to reduced number of accesses to the main memory as a result of better data locality. In the overall energy consumption,[4] the main memory energy dominates and the tiled versions result in significant energy savings. To sum up, *we can conclude that (for this matrix-multiply code) loop tiling increases the energy consumption in datapath, instruction cache, and data cache, but significantly reduces the energy in main*

[4]It should be noted that, for each code version in Figure 3.3, the total energy is slightly higher than the sum of datapath, cache, and main memory energies as it also includes the energy expended on buses.

memory. Therefore, if one only intends to prolong the battery life, one can apply tiling aggressively. If, on the other hand, the objective is to limit the energy dissipated within each package (e.g., the datapath+caches package), one should be more careful as tiling tends to increase both datapath and cache energies. It should also be mentioned that, in all the versions experimented with here, tiling improved the performance (by reducing execution cycles). Hence, it is easy to see that, since there is an increase in the energy spent (and decrease in execution time) within the package that contains datapath and caches, tiling causes an increase in average power dissipation for that package.

3.4.2 Sensitivity to the Tile Size

We now investigate the sensitivity of the energy behavior of tiling to the tile size. While the tile size sensitivity issue has largely been addressed in performance-oriented studies [67, 27, 35, 70, 83], the studies that look at the problem from energy perspective are few [90]. The results are given in Figure 3.4 for the input sizes 50 and 100. For each tiling strategy, we experiment with five different tile sizes (for N=50, the tile sizes are 2, 5, 10, 15, and 25, and for N=100, the tile sizes are 5, 10, 20, 25, and 50, both from left to right in the graphs for a given tiling strategy). We make several observations from these figures. First, increasing the tile size reduces the datapath energy and instruction cache energy. This is because a large tile size (blocking factor) means smaller loop (code) overhead. However, as in the previous set of experiments, the overall energy behavior is largely determined by the energy spent in main memory. Since the number of accesses to the data cache is almost the same for all tile sizes (in a given tiling strategy), there is little change in data cache energy as we vary the tile size.

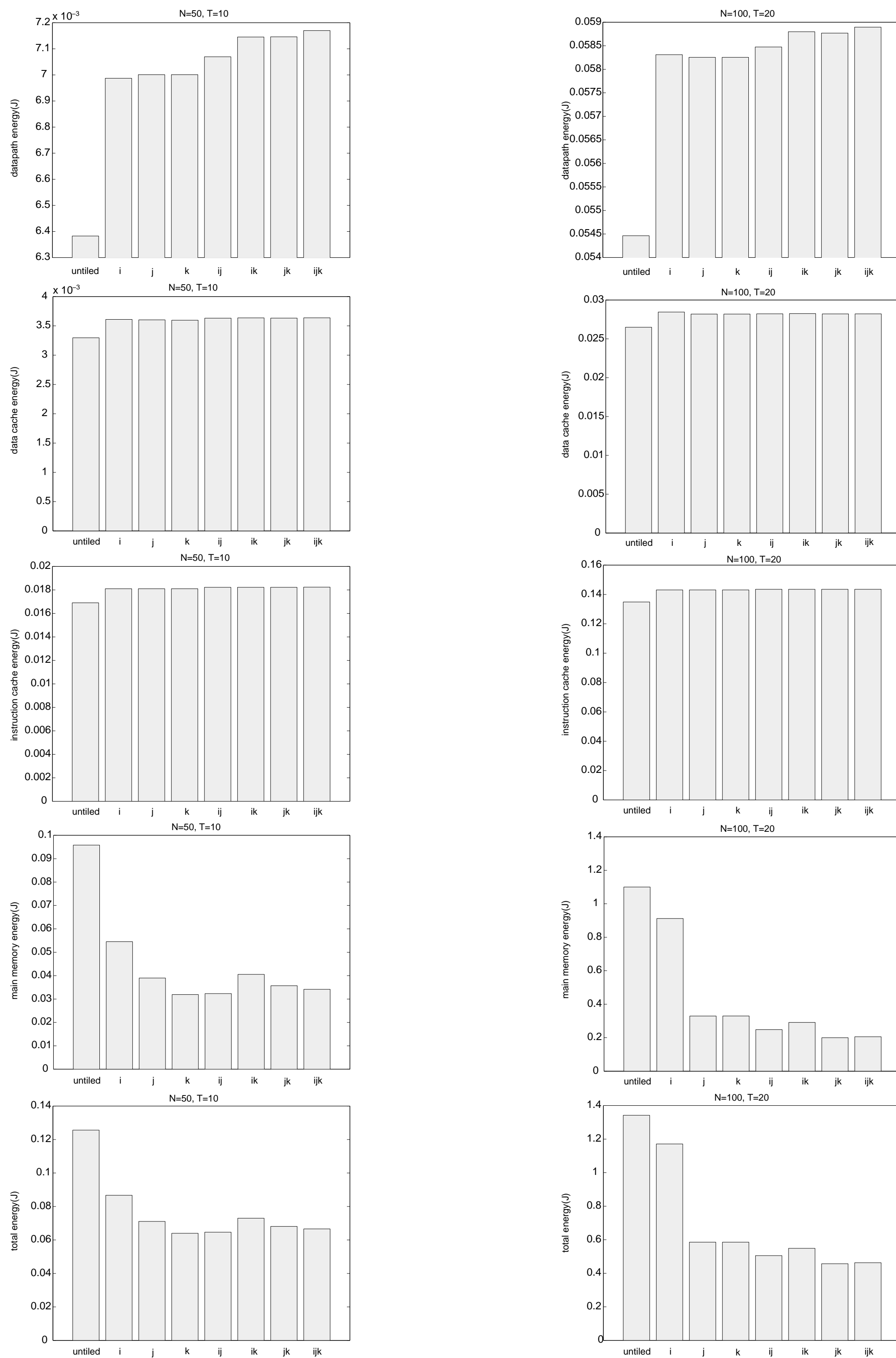


Fig. 3.3. Energy consumptions of different tiling strategies.

It is also important to see that the energy performance of a given tiling strategy depends to a large extent on the tile size. For instance, when N is 50 and both j and k loops are tiled, the overall energy consumption can be as small as 0.064J or as large as 0.128J depending on the tile size chosen. It can be observed that, for each version of tiled code, there is a most suitable tile size beyond which the energy consumption starts to increase. Moreover, *the most suitable tile size (from energy point of view) depends on the tiling style used.* For instance, when using the `ij` version (with N=50), a tile size of 10 generated the best energy results, whereas with the `ik` version the most energy efficient tile size was 5. This is because the low-level code generated for the `ij` version is quite different from that of the `ik` version as far as the memory access patterns are concerned; therefore, they work best with different tile sizes. Further, *for a given version, the best tile size from energy point of view was different from the best tile size from the performance (execution cycles) point of view.* For example, as far as the execution cycles are concerned, the best tile size for the `ik` version was 10 (instead of 5). These results motivate further research in determining optimal tile size from energy and energy–delay perspectives.

3.4.3 Sensitivity to the Input Size

In this subsection, we vary the input size (N) and observe the variation in energy consumption of the untiled code and a specific tiled code in which all three loops in the nest are tiled (i.e., the `ijk` version). Specifically, we would like to observe the benefits (if any) of tailoring the tile size to the input size. The input sizes used are 50, 100, 200, 300, and 400. For each input size, we experimented with five different tile sizes (5, 10, 15, 20,

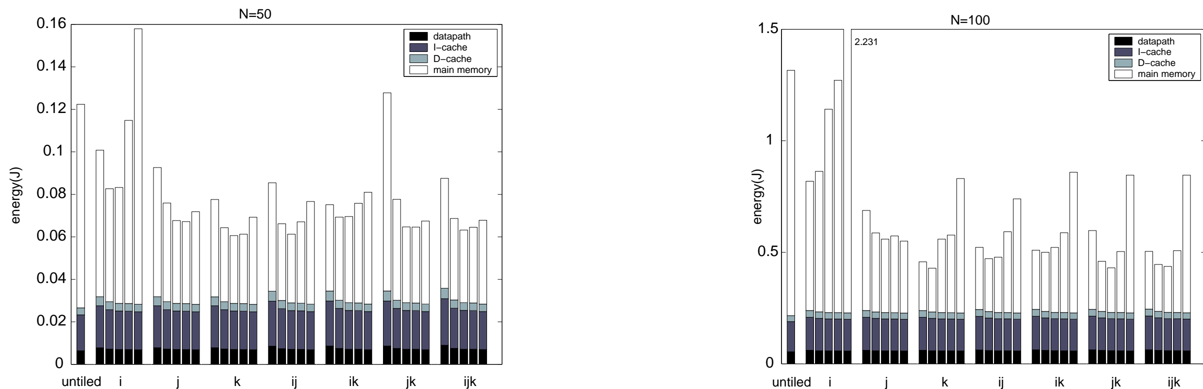


Fig. 3.4. Energy sensitivity of tiling to the tile size (blocking factor).

and 25). The results, given in Table 3.1, show that *for each input size there exists a best tile size from energy point of view.* To be specific, the best possible tile sizes (among the ones we experimented) for input sizes of 50, 100, 200, 300, and 400 are 10, 20, 20, 15, and 25, respectively. Consequently, it may not be a very good idea from energy point of view to fix the tile size at specific values. While one can develop optimal tile size detection algorithms (for energy) similar to the algorithms proposed for detecting best tile sizes for performance (e.g., [27, 67, 100, 35]), this issue is beyond the scope of this chapter. We also observe from Table 3.1 that the relative contributions of different system components to the overall energy are relatively stable over different input sizes.

3.4.4 Sensitivity to the Cache Configuration

In this subsection, we evaluate the data cache energy consumption when the underlying cache configuration is modified. We experiment with different cache sizes and

N		tile size					
		untiled	5	10	15	20	25
50	`dp`	0.006	0.008	0.007	0.007	0.007	0.007
	`ic`	0.017	0.019	0.018	0.018	0.018	0.018
	`dc`	0.003	0.004	0.004	0.004	0.004	0.004
	`mem`	0.096	0.038	0.034	0.036	0.035	0.039
100	`dp`	0.054	0.063	0.060	0.060	0.059	0.059
	`ic`	0.135	0.151	0.146	0.145	0.144	0.143
	`dc`	0.026	0.031	0.029	0.029	0.028	0.028
	`mem`	1.100	0.259	0.210	0.217	0.206	0.277
200	`dp`	0.385	0.484	0.467	0.464	0.459	0.460
	`ic`	1.078	1.209	1.166	1.156	1.148	1.145
	`dc`	0.213	0.248	0.233	0.229	0.226	0.225
	`mem`	11.003	3.080	2.635	2.683	2.586	3.287
300	`dp`	1.388	1.704	1.648	1.642	1.627	1.632
	`ic`	3.638	4.080	3.937	3.895	3.875	3.864
	`dc`	0.722	0.836	0.785	0.770	0.763	0.760
	`mem`	42.859	9.448;	8.033	7.618	7.951	9.365
400	`dp`	3.166	3.962	3.849	3.826	3.791	3.795
	`ic`	8.621	9.671	9.331	9.238	9.185	9.158
	`dc`	1.707	1.982	1.859	1.826	1.808	1.799
	`mem`	96.087	22.324	18.385	18.666	17.984	15.041

Table 3.1. Energy consumption of tiling with different input sizes. `dp`, `ic`, `dc`, and `mem` denote the energies spent in datapath, instruction cache, data cache, and main memory, respectively.

associativities as well as two energy-efficient cache architectures, namely, block buffering [92, 49, 41] and sub-banking [92, 49, 41].

In the block buffering scheme, the previously accessed cache line is buffered for subsequent accesses. If the data within the same cache line is accessed on the next data request, only the buffer needs to be accessed. This avoids the unnecessary and more energy consuming access to the entire cache data array. Thus, increasing temporal locality of the cache line through compiler techniques such as loop tiling can save more energy. In the cache sub-banking optimization, the data array of the cache is divided into several sub-banks and only the sub-bank where the desired data is located is accessed. This optimization reduces the per access energy consumption and is *not* influenced by locality optimization techniques. We also evaluate cache configurations that combine both these optimizations. In such a configuration with block buffering and sub-banking, each sub-bank has an individual buffer. Here, the scope for exploiting locality is limited as compared to applying only block buffering as the number of words stored in a buffer is reduced. However, it provides the additional benefits of sub-banking for each cache access.

We first focus on traditional cache model and present in Figure 3.5 the energy consumed *only* in data cache for different cache sizes and associativities. We experiment with two different codes (with N=200), the untiled version and a blocked version where all three loops (i, j, and k) are tiled with a tile size of twenty. Our first observation is that the data cache energy is not too sensitive to the associativity but, on the other hand, is very sensitive to the cache size. This is because for a given code, the number of read accesses to the data cache is constant and, the cache energy per data access is

higher for a larger cache. Increasing associativity also increases per access cost for cache (due to increased bit line and word line capacitances), but its effect is found to be less significant as compared to the increase in bit line capacitance due to increased cache sizes. As a result, embedded system designers need to determine minimum data cache size for the set of applications in question if they want to minimize data cache energy. Another observation is that for all cache sizes and associativities going from the untiled code to tiled code increases the data cache energy.

We next concentrate on cache line size and vary it between 8 bytes and 64 bytes for N=200 and T=50. The energy consumption of the `ijk` version for line sizes of 8, 16, 32, and 64 bytes were 0.226J, 0.226J, 0.226J, and 0.227J, respectively, indicating that (for this code) the energy consumption in data cache is relatively independent from the line size. It should also be mentioned that while increases in cache size and degree of associativity might lead to increases in data cache energy, they generally reduce the overall memory system energy by reducing the number of accesses to the main memory.

Finally, we focus on block buffering and sub-banking, and in Figure 3.6 give the data cache energy consumption for different combinations of block buffering (denoted `bb`) and sub-banking (denoted `sb`) for both the untiled and tiled (the `ijk` version) codes. *The results reveal that for the best energy reduction block buffering and sub-banking should be used together.* When used alone, neither sub-banking nor block buffering is much effective. The results also show that increasing the number of block buffers does not bring any benefit (as there is only one reference with temporal locality in the innermost loop). It should be noted that the energy increase caused by tiling on data cache can (to some extent) be compensated using a configuration such as `bb+sb`.

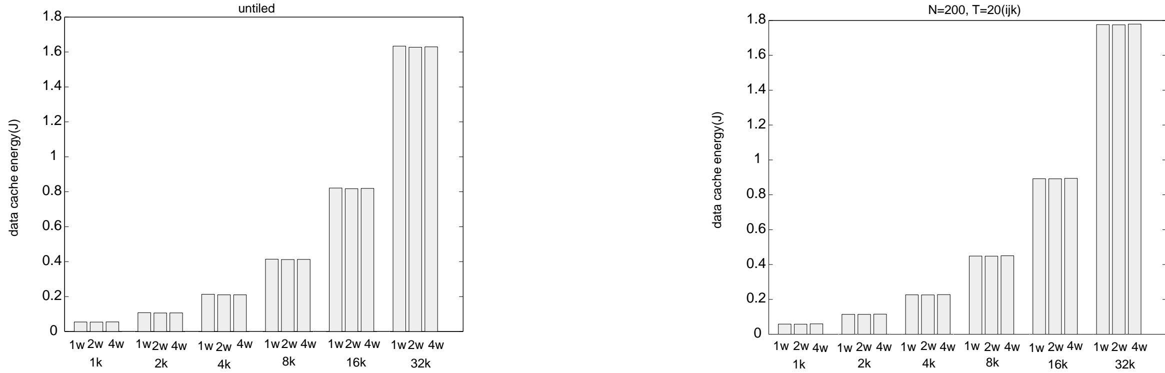


Fig. 3.5. Impact of cache size and associativity on data cache energy.

3.4.5 Cache Miss Rates vs. Energy

We now investigate the relative variations in miss rate and energy performance due to tiling. The following three measures are used to capture the correlation between the miss rates and energy consumption of the unoptimized (original) and optimized (tiled) codes.

$$\texttt{Improvement}_{\texttt{m}} = \frac{\text{Miss rate of the original code}}{\text{Miss rate of the optimized code}},$$

$$\texttt{Improvement}_{\texttt{e}} = \frac{\text{Memory energy consumption of the original code}}{\text{Memory energy consumption of the optimized code}},$$

$$\texttt{Improvement}_{\texttt{t}} = \frac{\text{Total energy consumption of the original code}}{\text{Total energy consumption of the optimized code}}.$$

In the following discussion, we consider four different cache configurations: 1K, 1-way; 2K, 4-way; 4K, 2-way; and 8K, 8-way. Given a cache configuration, Table 3.2 shows how these *three* measures vary when we move from the original version to an optimized version.

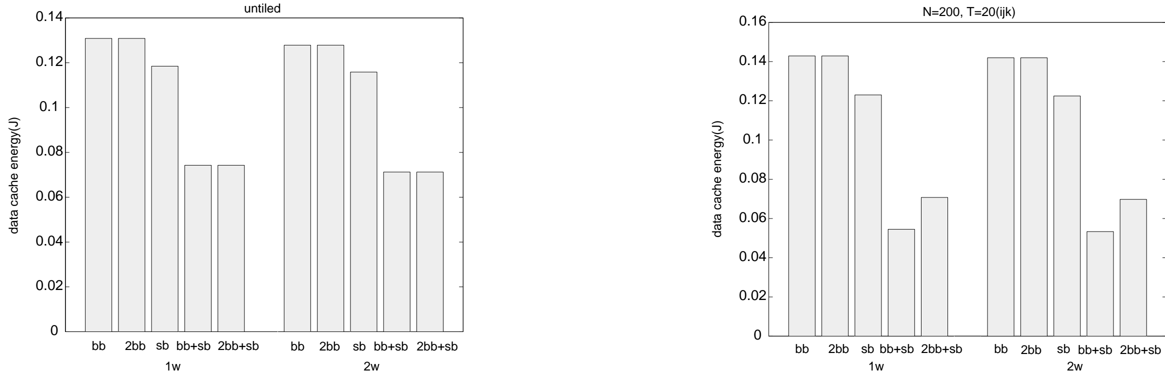


Fig. 3.6. Impact of block buffering (`bb`) and sub-banking (`sb`) on data cache energy.

We see that in spite of very large reductions in miss rates as a result of tiling, the reduction in energy consumption is not as high. Nevertheless, it still follows the miss rate. We also made the same observation in other codes we used. We have found that $\texttt{Improvement}_{\texttt{e}}$ is smaller than $\texttt{Improvement}_{\texttt{m}}$ by a factor of 2 - 15. Including the datapath energy makes the situation worse for tiling (from the energy point of view), as this optimization in general increases the datapath energy consumption. Therefore, *compiler writers for energy aware systems can expect an overall energy reduction as a result of tiling, but not as much as the reduction in the miss rate.* We believe that some optimizing compilers (e.g., [87]) that estimate the number of data accesses and cache misses statically at compile time can also be used to estimate an approximate value for the energy variation. This variation is mainly dependent on the energy cost formulation parameterized by the number of hits, number of misses, and cache parameters.

	1K, 1-way	2K, 4-way	4K, 2-way	8K, 8-way
$\texttt{Improvement}_m$	6.21	63.31	20.63	19.50
$\texttt{Improvement}_e$	2.13	18.77	5.75	2.88
$\texttt{Improvement}_t$	1.96	9.27	3.08	1.47

Table 3.2. Improvements in miss rate and energy consumption.

3.4.6 Interaction with Other Optimizations

In order to see how loop tiling gets affected by other loop optimizations, we perform another set of experiments where we measure the energy consumption of tiling with linear loop optimization (loop interchange [99] to be specific) and loop unrolling [99]. Loop interchange modifies the original order of loops to obtain better cache performance. In our matrix-multiply code, this optimization converts the original loop order i, j, k (from outermost to innermost) to i, k, j, thereby obtaining spatial locality for arrays b and c, and temporal locality for array a, all in the innermost loop. We see from Figure 3.7 that tiling (in general) reduces the overall energy consumption of even this optimized version of the matrix-multiply nest. Note however that it increases the datapath energy consumption. Comparing these graphs with those in Figure 3.3, we observe that interchanged tiled version performs better than the pure tiled version, which suggests that *tiling should be applied in general after linear loop transformations for the best energy results.*

The interaction of tiling with loop unrolling is more complex. Loop unrolling reduces the iteration count by doing more work on a single loop iteration. We see from Figure 3.7 that untiled loop unrolling may not be a good idea as its energy consumption

is very high. Applying tiling brings the energy consumption down. Therefore, *in circumstances where loop unrolling must be applied* (*e.g., to promote register reuse and/or to improve instruction level parallelism*), *we suggest to apply tiling as well to keep the energy consumption under control.*

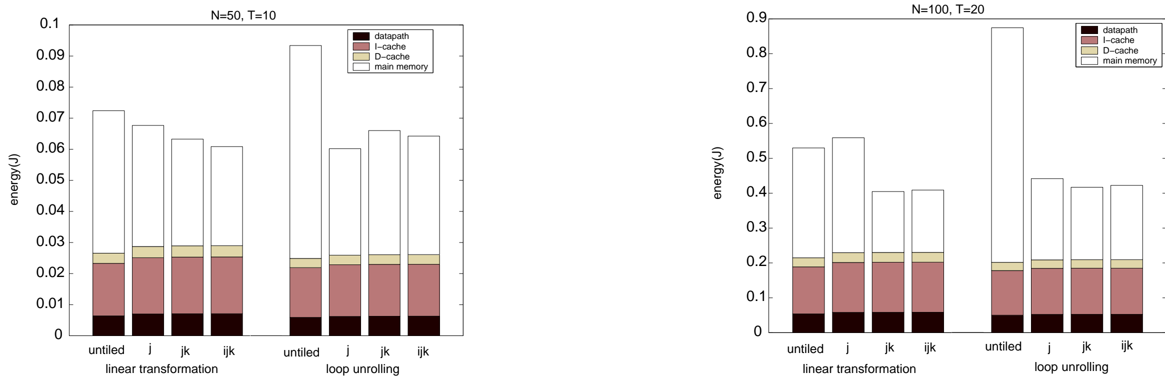


Fig. 3.7. Interaction of loop tiling with loop interchange and unrolling.

3.4.7 Analysis of Datapath Energy

We next zoom-in on the datapath energy, and investigate the impact of tiling on different components of the datapath as well as on different stages of the pipeline. Table 3.3 shows the breakdown of the datapath energy for the matrix-multiply code into different hardware components. For comparison purposes, we also give the breakdown for two other optimizations, loop unrolling (denoted `u`) and linear loop transformation (denoted `l`), as well as different combinations of these two optimizations and tiling (denoted `t`), using an input parameter of N=100. Each entry in this table gives the

percentage of the datapath energy expended in the specific component. *We see that (across different versions) the percentages remain relatively stable. However, we note that all the optimizations increase the (percentage of) energy consumption in functional units due to more complex loop nest structures that require more computation in the ALU. The most significant increase occurs with tiling, and it is more than 26%.* These results also tell us that most of the datapath energy is consumed in register files and pipeline registers, and therefore the hardware designers should focus more on these units. Table 3.4, on the other hand, gives the energy consumption breakdown across five pipeline stages (the fetch stage `IF`, the instruction decode stage `ID`, the execution stage `EXE`, the memory access stage `MEM`, and the write-back stage `WB`). The entries under the `MEM` and `IF` stages here do not involve the energy consumed in data and instruction cache memory, respectively. We observe that most of the energy is spent in the `ID`, `EXE`, and `WB` stages. Also, the compiler optimizations in general increase the energy consumption in the `EXE` stage, since that is where the ALU sits; this increase is between 1% and 8% and also depends on the program being run.

3.4.8 Sensitivity to Technology Changes

The main memory has been a major performance bottleneck and has attracted a lot of attention [91, 75, 3]. Changes in process technology have made possible to embed a DRAM within the same chip as the processor core. Initial results using embedded DRAM (eDRAM) show an order of magnitude reduction in the energy expended in main memory [91]. Also, there have been significant changes in the DRAM interfaces [30] that can potentially reduce the energy consumption. For example, unlike conventional DRAM

Version	Register File	Pipeline Registers	Functional Units	Data-path Muxes
unoptimized	35.99	36.33	15.76	8.36
l	36.09	34.87	17.34	8.11
u	36.19	36.17	15.98	8.31
t	34.60	33.56	19.93	7.80
l+u	35.87	34.12	18.19	7.93
l+t	35.27	33.74	19.25	8.17
t+u	35.31	35.07	17.89	8.06
t+l+u	35.41	34.15	18.38	7.96

Table 3.3. Datapath energy breakdown (in %s) in hardware components level.

Version	IF	ID	EXE	MEM	WB
unoptimized	3.33	22.94	33.17	8.70	31.87
l	3.10	23.88	34.20	8.32	30.50
u	3.18	23.93	33.47	8.63	30.78
t	3.25	24.04	35.91	7.95	28.85
l+u	2.97	24.83	34.81	8.13	29.27
l+t	2.95	23.23	35.73	8.07	30.02
t+u	3.15	23.61	34.78	8.34	30.12
t+l+u	2.95	24.63	35.02	8.14	29.26

Table 3.4. Datapath energy breakdown (in %s) in pipeline stage level.

memory sub-systems that have multiple memory modules that are active for servicing data requests, the direct RDRAM memory sub-system delivers a full bandwidth with only one RDRAM module active. Also, based on the particular low power modes that are supported by the memory chips and based on how effectively they are utilized, the average per access energy cost for main memory can be reduced by up to two orders of magnitude [31].

In order to study the influence of changes in $\texttt{E}_\texttt{m}$ due to these technology trends, we experiment with different $\texttt{E}_\texttt{m}$ values that range from 4.95×10^{-9} (our default value) to 2.475×10^{-11}. We observe from Figure 3.8 that from $\texttt{E}_\texttt{m} = 4.95 \times 10^{-9}$ on, the main memory energy starts to lose its dominance and instruction cache and datapath energies constitute the largest percentage. While this is true for both tiled and untiled codes, the situation in tiled codes is more dramatic as can be seen from the figure. For instance, when $\texttt{E}_\texttt{m} = 2.475 \times 10^{-10}$, N=100, and T=10, the datapath energy is nearly 5.7 times larger than the main memory energy (which includes the energy spent in both data and instruction accesses), and the Icache energy is 13.7 times larger than the main memory energy. With the untiled code, however, these values are 0.98 and 2.43, respectively. *The challenge for future compiler writers for power aware systems then is to use tiling judiciously so that the energy expended in datapath and on-chip caches can be kept under control.*

3.4.9 Other Codes

In order to increase our confidence in our observations on the matrix-multiply code, we also performed tiling experiments using several other loop nests that manipulate

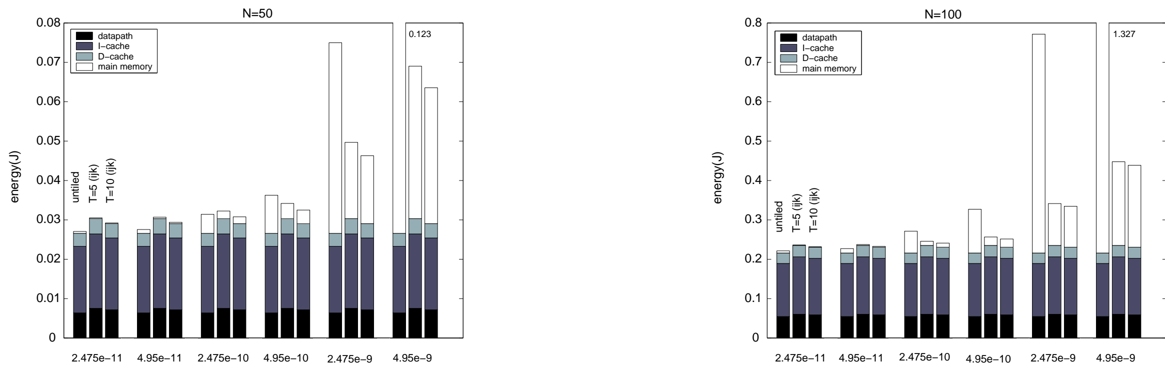


Fig. 3.8. Energy consumption with different E_m (J) values.

multi-dimensional arrays. The salient characteristics of these nests are summarized in Table 3.5. The first four loop nests in this table are from the Spec92/Nasa7 benchmark suite; `syr2k.1` is from Blas; and `htribk.2` and `qzhes.4` are from the Eispack library. For each nest, we used several tile sizes, input sizes, and per memory access costs, and found that the energy behavior of these nests are similar to that of the matrix-multiply. However, due to lack of space, we report here only the energy break-down of the untiled and two tiled codes (in a *normalized* form) using two representative E_m values (4.95×10^{-9} and 2.475×10^{-11}). In Figure 3.9, for each code, the three bars correspond to the untiled, tiled (`ijk`,T=5), and tiled (`ijk`,T=10) versions, respectively, from left to right. Note that while the main memory energy dominates when E_m is 4.95×10^{-9}, the instruction cache and datapath energies dominate when E_m is 2.475×10^{-11}.

nest	arrays	data size	tile sizes
`btrix.4`	two 4-D	21.0 MB	10 and 20
`vpenta.3`	one 3-D and five 2-D	16.6 MB	20 and 40
`cholesky.2`	one 3-D	10 MB	10 and 20
`emit.4`	one 2-D and one 1-D	2.6 MB	50 and 100
`htribk.2`	three 2-D	72 KB	12 and 24
`syr2k.1`	three 2-D	84 KB	8 and 16
`qzhes.4`	one 2-D	160 KB	15 and 30

Table 3.5. Benchmark nests used in the experiments. The number following the name corresponds to the number of the nest in the respective code.

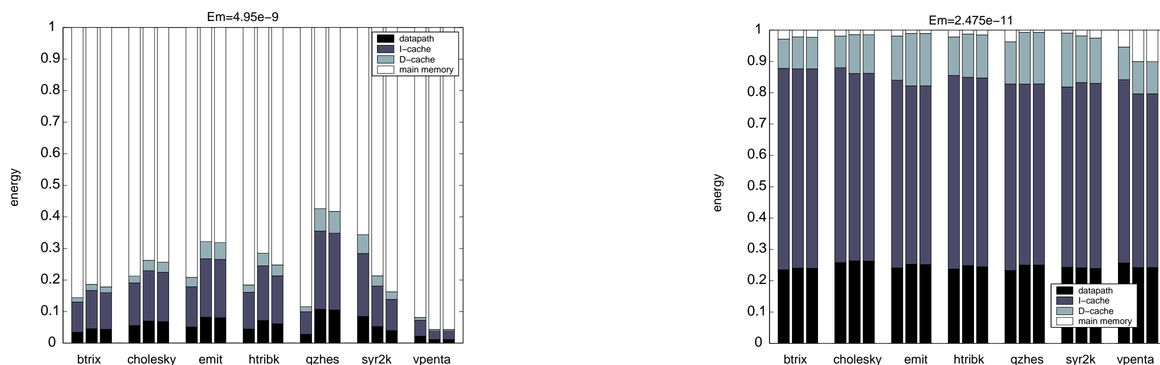


Fig. 3.9. Normalized energy consumption of example nested loops with two different E_m (J) values.

3.5 Related Work

Compiler researchers have attacked the locality problem from different perspectives. The works presented in Wolf and Lam [100], Li [70], Coleman and McKinley [27], Kodukula et al. [65], Lam et al. [67], and Xue and Huang [102], among others, have suggested tiling as a means of improving cache locality. In [100] and [70], the importance of linear locality optimizations before tiling is emphasized. While all of these studies have focused on the performance aspect of tiling, in this chapter, we investigate its energy behavior and show that the energy behavior of tiling may depend on a number of factors including tile size, input size, cache configuration, and per memory access cost.

Memory optimizations for embedded systems have been partly addressed by Panda et al. [77] and Shiue and Chakrabarti [90]. Panda et al. [77] used cache size and processor cycle count as performance metrics and proposed a method for off-chip data placement. Shiue and Chakrabarti [90] presented a memory exploration strategy based on three metrics, namely, processor, cycles, cache size, and energy consumption. They have found that increasing tile size and associativity reduces the number of cycles but does not necessarily reduce the energy consumption. In comparison, we focus on the entire system (including datapath and instruction cache) and study the impact of a set of parameters on the energy behavior of tiling using several tiling strategies. We also show that datapath energy consumption due to tiling might be more problematic in the future, considering the current trends in memory technology. The IMEC group [20] was among the first to work on applying loop transformations to minimize power dissipation in data dominated embedded applications.

In this chapter, we utilize the framework that was proposed in [96, 104]. This framework has been used to investigate the energy influence of a set of high-level compiler optimizations that include tiling, linear loop transformations, loop unrolling, loop fusion, and distribution [53]. However, the work in [53] accounts for only the energy consumed in data accesses and does not investigate tiling in detail. In contrast, our current work looks at tiling in more detail, investigating different tiling strategies, influence of varying tile sizes, and the impact of input sizes. Also, in this chapter, we account for the energy consumed by the entire system including the instruction accesses. While the work in [52] studies different tiling strategies, this chapter focuses on impact of different tiling parameters, the performance versus energy consumption impact, and the interaction of tiling with other high-level optimizations.

The studies of energy consumption in caches are also growing. Bunda et al. [16] and Furber et al. [37] presented studies of instruction set design and its effects on cache performance and power consumption. Su and Despain [92] and Burgress et al. [36] also presented low-power cache design examples. These studies are orthogonal to our work and the interaction between hardware low power cache design techniques and compiler-directed optimizations for low energy merits further investigation.

3.6 Conclusion

When loop nest based computations process large amounts of data that do not fit in cache, tiling is an effective optimization for improving performance. While previous work on tiling has focused exclusively on its impact on performance (execution cycles), it is critical to consider its impact on energy as embedded and mobile devices are becoming

the tools for mainstream computation and start to take the advantage of high-level and low-level compiler optimizations.

In this chapter, we study the energy behavior of tiling considering both the entire system and individual components such as datapath, caches, and main memory. Our results show that the energy performance of tiling is very sensitive to input size and tile size. In particular, selecting a suitable tile size for a given computation involves tradeoff between energy and performance. We find that tailoring tile size to the input size generally results in lower energy consumption than working with a fixed tile size. Since the best tile sizes from the performance point of view are not necessarily the best tile sizes from the energy point of view, we suggest experimenting with different tile sizes to select the most suitable one for a given code, input size, and technology parameters. Also, given the current trends in memory technology, we expect that the energy increase in datapath due to tiling will demand challenging tradeoff between prolonging battery life and limiting energy dissipated within a package.

Chapter 4

ESTIMATING INFLUENCE OF DATA LAYOUT OPTIMIZATIONS ON SDRAM ENERGY CONSUMPTION

4.1 Introduction

The growing speed gap between processors and off-chip memories and the increasing memory bandwidth demand of video processing applications make efficient memory accesses critical. The power consumption of memory accesses by such applications has also become a very crucial factor as more and more portable devices support video in addition to other forms of data. Newer DRAM families such as SDRAMs and RDRAMs now provide various modes to support the application's bandwidth requirement (e.g., in the burst mode, a burst of data can be accessed in one transaction). The multiple banked memory architectures make it possible for two consecutive accesses to different banks be overlapped seamlessly by careful selection of burst size. However, if two successive accesses are to two different rows (pages) of the same bank, they are in large part serialized, thereby increasing memory latency. We refer to this change of pages as the "page break." Although the cycle penalty can be hidden by judicious placement of memory accesses (using, for example, software prefetching) or bank interleaved accesses, the energy consumption caused by precharge and activation of memory cannot always be avoided. As an example, for the Micron's four banked 8MB SDRAM with 32-bit wide

data output lines, a page activation costs about six times more energy than the data transfer of one word [1].

In the past, several techniques have been proposed to reduce the number of page breaks in DRAMs. For example, in [56], new CDFG (control-data flow graph) operations are defined to enable finer memory access scheduling during high-level synthesis. Various optimization techniques have also been proposed. Specifically, their array-to-bank assignment algorithm builds array interference graphs and partitions them so that arrays which are accessed in parallel are assigned to different banks. They reorder array references in the code such that same array references are moved close to one another in a basic block (this enables the scheduler to identify the possibility of invoking the page or burst mode accesses). Various loop transformations are also applied to take advantage of SDRAM access modes.

In this part of the thesis, our focus is on an SDRAM-based architecture that executes video processing codes. First, we propose an estimation framework to count the number of page breaks. While it might be possible to use detailed simulation for small kernels [57], we believe that a fast estimation technique (which can also be embedded within an optimizing compiler) can be of great value. We show that Presburger arithmetic and Ehrhart polynomials can be used for estimating the number of page breaks statically (i.e., at compile time). The experimental results indicate that the proposed framework can estimate page breaks very well.

Our second objective is to study the impact of memory layout on energy savings. Specifically, we propose to use block-based layouts for data arrays instead of more traditional row-major or column-major layouts. The idea here is to make the memory

layouts more compatible with the access patterns of video applications and to minimize the number of page breaks. It should be observed that page breaks can be costly from both energy and performance perspectives. As compared to previous hardware-based studies in this direction [57], our approach is purely software-based and does not require any modification to the memory controller hardware. While block layouts have been used by prior research in the area of cache locality [24], our work employs them in the context of SDRAMs.

We also extend the estimation framework for block-based layouts. This is important, since it can be defined using different dimension sizes (e.g., 50×50 blocks versus 75×60 blocks). To determine the best sizes, we need to estimate the number of page breaks incurred by a given size.

The rest of this chapter is organized as follows. In the next section, we provide background information on operations of SDRAM and on the Polyhedral model. In Section 4.3 and Section 4.4, we discuss our approach in detail. In Section 4.5, we introduce our simulation environment and present experimental data. Finally, we conclude this chapter in Section 4.6.

4.2 Preliminaries

In this section, we first explain the basic operations of an SDRAM, and then give background information on the Presburger formulas (which is the cornerstone of our implementation).

4.2.1 SDRAM Basics

In our work, we focus on SDRAM, since it is currently the most widely used DRAM type in embedded video applications. The diagram of a typical SDRAM is depicted in Figure 4.1. Basic operations in an SDRAM consist of three steps : precharge, activation, and read/write. A page activation command selects a bank and row address, and transfers that row's cell data, which is stored in the array, to the sense amplifiers. The data stays in the sense amplifiers until a new precharge command is issued to the same bank, during which a read or a write operation can take place. The memory can be precharged right after the activation, which is called the closed page policy, or before the next activation (referred to as the open page policy), depending on the SDRAM controller policy. We assume open page policy in our work.

In general, in an SDRAM, the memory array is divided into multiple banks (which share address and data buses) — in Figure 4.1, we have four banks. This allows one bank to be precharged while the other is being accessed. This hides precharge latency and effectively allows bandwidth to be increased. SDRAMs incorporate an on-chip burst counter, which can be used to increment column addresses for very fast burst access. The burst length and burst type (sequential or interleaved) can be selected by programming the mode register. It should be noted that one of the most important problems in making effective use of an SDRAM-based architecture is that of ensuring data reuse. This is because, if an access pattern frequently changes the pages that it touches, it incurs extra latency in memory accesses and increases energy consumption. Instead, a preferable access pattern would exploit data locality at the page granularity. As will

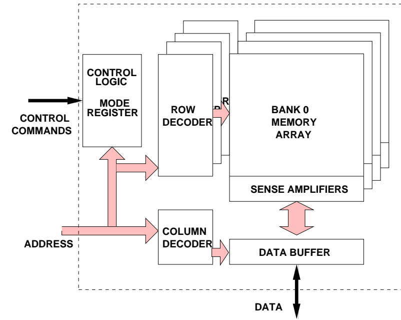


Fig. 4.1. Diagram of a typical SDRAM. The mode register is used to set the memory access mode to use. If the burst mode is selected, an entire page is buffered. Whether such a buffering is beneficial or not depends strongly on the (page-level) data reuse exhibited by the application.

be discussed in Section 4.4, exploiting data locality is particularly challenging in video applications, since these applications access data in rectilinear blocks, each of which may span multiple page boundaries.

4.2.2 The Polyhedral Model

Presburger formulas [66] are those formulas that can be constructed by combining affine constraints on integer variables with the logical operations $\vee$ (or), $\wedge$ (and) and ! (not), and the quantifiers $\forall$ and $\exists$. It should be noticed that the affine constraints can be either equality constraints or inequality constraints. For example, $\{[k] | \quad \exists \alpha$ such that $k = 2\alpha \wedge (0 < k < n)\}$ is a Presburger formula that represents all positive even numbers less than n. There are a number of algorithms for testing the satisfiability of arbitrary Presburger formulas, and the Omega Library [2] is a set of C++ classes for manipulating Presburger formulas. It is used in many research projects in the area of compilation for high-performance computers (e.g., data dependence analysis, program transformations,

detecting redundant synchronization, and code generation). In this work, we employ Omega Calculator, a user-friendly interface to the Omega Library to formulate the conditions for page breaks. We also make use of Ehrhart polynomials, which permit us to count the number of integer points contained in a parameterized polyhedron. [1] In our context, Ehrhart polynomials count the number of page breaks.

4.3 Page Break Estimation Framework

Video applications usually work in blocks; that is, the computation is staged into portions and each portion operates on a block of data. In a quadtree-structured motion estimation program, for example, 16 pixels in each of 4×4 block are averaged in the sub-4 sampling phase for each of the 16×16 macro-blocks [88]. Consequently, the array data are accessed block-by-block. Consider a typical access scenario illustrated in Figure 4.2(a). Here, a given two dimensional array is accessed block-by-block. There can be two types of page breaks (transitions) in such an access. First, in processing a block, execution may move from one page to another. This is termed as the "intra-block page break." In addition, it is also possible to incur a page break when moving from one block to another. This type of page break is called the "inter-block page break."

In our work, the number of intra-block and inter-block page breaks is represented using Presburger formulas [66]. At the high level, our approach builds Presburger formulas that represent page breaks (as will be explained shortly), and simplifies them using the Omega Calculator [2]. After that, the simplified union of the polytope is transformed

[1] An "affine half-space" of R^n is the set of vectors x that satisfy the linear inequality $\alpha.x \leq \beta$ for some vector α and real number β. The intersection of a set of half-spaces is called a "polyhedron." A bounded polyhedron is called a "polytope.'[99]

to a disjoint union of polytope using PolyLib [71]. Then, Ehrhart polynomials count the number of integer points in the disjoint union of polytope, which gives the number page breaks. In the following discussion, we give the details of our Presburger formulas for identifying the page breaks. Before going into details of our formulation, however, we first present the mathematical representation that we adopt.

We consider the case of references to arrays with affine subscript functions in nested loops, which are common in video applications. Consider such an access to a p-dimensional array in an d-deep loop nest. Let l denote the iteration vector (consisting of loop indices starting from the outermost loop). All values that can be taken on by l are collectively represented by an iteration space I. Each reference to a p-dimensional array can be represented as $LI + o$, where the $p \times d$ matrix L is called the access matrix and the p-element vector o is called the offset vector [100]. To illustrate the concept, consider a reference to array Y, such as $Y[i+1][j-1]$ in a nest with two loops: i (outer) and j (inner). In this case, L is a two-by-two identity matrix and o is $[1 \quad -1]^T$. The pair (L, o) is also referred to as the index function.

Each array 'block' reference R_u in the code is represented using a quadruple $R_u = (Y, F_u, S, B)$. In this quadruple, Y is the array (referenced) divided into blocks B, F_u is the index function (an (L, o) pair), and S is the statement that contains the reference. Array Y is assumed to be of size $[0, d_0 - 1][0, d_1 - 1]$. In this work, we restrict our discussions to two dimensional arrays. In the following, we make several (Presburger Formula) definitions. In these definitions, $\models$ means "means that".

Valid Iteration Point: The predicate $l \in I$ indicates the fact that iteration point $l = [l_0, \ldots, l_{d-1}]$ belongs to the iteration space, assuming d nested loops.

$$l \in I \models \bigwedge_{i=0}^{d-1} (0 \leq l_i < n_i).$$

Lexicographical Ordering of Accesses: The predicate $(R_u, l) \prec (R_v, m)$ describes the fact that the memory access made by reference R_u at iteration l precedes the memory access made by R_v at iteration m, or (if $l = m$) R_u textually precedes R_v in the loop.

$$(R_u, l) \prec (R_v, m) \models l \in I \quad \wedge \quad m \in I \quad \wedge$$
$$((\exists i : 0 \leq i \leq d-1 \quad \wedge \quad l_i < m_i \quad \wedge \quad \bigwedge_{j=0}^{i-1} l_j = m_j) \vee$$
$$(\bigwedge_{j=0}^{d-1} l_j = m_j \quad \wedge \quad u < v)).$$

Data Layouts in Memory: For a given array element, we need to determine its location in memory. If an array, Y, is stored in row-major order, an array element referred by the index $F_u(l)$, where l is a valid iteration point can be found at:

$$m = L_x(F_u(l), \mu_x) \models m \geq 0 \quad \wedge$$
$$m = \mu_x + (d_1 i_0 + i_1).$$

For an array stored in column-major order, its array element can be found at:

$$m = L_x(F_u(l), \mu_x) \models m \geq 0 \quad \wedge$$
$$m = \mu_x + (i_0 + d_0 i_1).$$

Here, μ_x is the starting address (base address) for the array.

Mapping Memory Locations to Memory Banks: We assume a linear mapping of memory locations to the SDRAM and that the total size of the SDRAM, $S_{total} = N_{row} * N_{bank} * P$, where N_{row}, N_{bank} and P represent the number of rows per (SDRAM) bank, the number of banks, and the size of a page in bytes, respectively. In addition, we assume that the address space of the SDRAM is $[0, S_{total} - 1]$. Based on these, a memory address m is assigned to row r of bank b, where $b*N_{row}*P+r*P \leq m < b*N_{row}*P+(r+1)*P$. This can be expressed as:

$$Map(m, b, r) \models 0 \leq b < N_{bank} \quad \wedge \quad 0 \leq r < N_{row} \quad \wedge$$
$$b * N_{row} * P + r * P \leq m < b * N_{row} * P + (r + 1) * P.$$

Page Break Model for Conventional Data Layouts: We define a fetch order between two elements of the block. Note that the lexicographical order defined earlier is for blocks. For each block reference, R_u, a fetch order for elements inside the block can

be defined as:

$$\begin{aligned}(r_i, r_j) \models i, j \in B_I \quad &\wedge \\ i \prec j \quad &\wedge \\ \neg(\exists k : k \in B_I \quad \wedge \quad i \prec k \prec j).\end{aligned}$$

B_I represents the iteration space for the block fetch, [0, 0] to $[b_0 - 1, b_1 - 1]$. This order is independent of loop nests but rather dependent on data layout order. For example, if the block of data is stored in row-major order, we might as well access each element row-wise.

An intra-page break occurs between two consecutive fetch elements, r_i and r_j, in a block reference, R_u, if they are mapped into two different rows and can be modeled as:

$$\begin{aligned}(r_i, r_j) \in IntraPageBreak(L) \models i, j \in B_I \quad &\wedge \\ r_i, r_j \in R_u \quad &\wedge \\ \exists b, r : Map(L_x(F_u(l) + i, \mu_x), b, r) \quad &\wedge \\ \exists r' : Map(L_x(F_u(l) + j, \mu_x), b, r') \quad &\wedge \\ (r \neq r').\end{aligned}$$

An inter-page break exists between two blocks, R_u and R_v, if the last element of R_u is fetched from a different row from that of first element of R_v and can be modeled

as:

$$
\begin{aligned}
(r_i, r_j) \in InterPageBreak(L) \models i, j \in B_I \quad \wedge \\
r_i \in R_u \quad \wedge \quad r_j \in R_v \quad \wedge \\
\neg(\exists i' : r_i \prec r_{i'}) \quad \wedge \\
\neg(\exists j' : r_{j'} \prec r_j) \quad \wedge \\
\neg(\exists w : R_u \prec R_w \prec R_v) \quad \wedge \\
\exists b, r : Map(L_x(F_u(l_u) + B, \mu_x), b, r) \quad \wedge \\
\exists r' : Map(L_x(F_v(l_v), \mu_x), b, r') \quad \wedge \\
(r \neq r').
\end{aligned}
$$

Counting Page Breaks: The total number of page breaks can be obtained by adding the number of intra-page breaks and inter-page breaks. We use the Omega Calculator to simplify the formulas above. After simplification, we are left with formulas defining a union of polytope. The number of integer points in this union is the number of breaks. PolyLib is used to operate on such unions. We first convert the union into a disjoint union of polytope, and then use Ehrhart polynomials to count the number of integer points in each polytope. It might happen that the location of the first block is dependent on the result of the computation such as motion vectors. It should be observed that this can be incorporated into our formulas as Presburger formula allow symbolic representations [66, 2]. The output of the final Ehrhart polynomials is represented in the form of the motion vectors as free variables. Since the motion vectors have specific

ranges allowed in the program, the number of page breaks are calculated by averaging the polynomial over that range, assuming random distribution of motion vectors.

4.4 Blocked Data Layout

4.4.1 Overview

As mentioned earlier, video applications work in blocks. However, such an access pattern might be problematic from the SDRAM perspective as illustrated in Figure 4.2(a). Every block fetch incurs almost three intra-block page breaks, since the data is stored row-wise. The strategy proposed in this thesis is based on the concept of "block-based memory layout" (also called the "tile-based layout"). The idea is that instead of storing the array in row-major (or column-major) order in memory — as is the method adopted by current languages/compilers, one can store the array in a block-by-block fashion. Specifically, a given array is divided into "blocks" ("tiles") of $H \times V$, and the elements in each block are stored in consecutive memory locations. It should be noted that the elements that map to the same block can be stored (within that block) in row-major or column-major order (or even in more complex storage forms [24]), and also, the relative storage order of blocks with respect to each other can be row-major or column-major. This alternative storage strategy is depicted in Figure 4.2(b). It should be noted that here we have $H \times V = P$, where P is the page size. One good characteristic of this storage form is that it fits very well to the access pattern in video codes (which is also block based). For example, if we consider the $a \times b$ block in the upper-left portion of Figure 4.2(b), we see that all data elements accessed are within the same page, meaning

that the locality at the page-level is exploited fully. It is easy to see in Figure 4.2(b) that the blocks marked by (1), (2), and (3) span, respectively, 2, 2, and 4 pages. Therefore, assuming the existence of a local memory (that can keep at least one block of data), we need to incur only 1, 1, and 3 page breaks, respectively. That is, we can access each page one-by-one and transfer the required array elements (from that page) to the local memory before moving to the next page.

It should be noted, however, that determining the most suitable values for H and V is critical. This is because if these values are not selected carefully, we may incur a large number of page breaks. A good selection of these values can minimize the number of the blocks marked (1), (2), and (3) in Figure 4.2(b). The main difficulty here is that, in general, it is not possible to determine the H and V values by considering only one nest in the application. Instead, the entire program should be taken into account. Based on this observation, we propose an estimation strategy that counts the number of page breaks considering the access pattern exhibited by the entire application. We restrict our focus to a single array at a time; that is, we do not consider page transitions between different arrays, since there exist techniques that reduce the number of such transitions by proper assignment of arrays to memory banks [56].

4.4.2 Page Break Estimation Framework

To estimate page breaks for blocked layout, the two following formulas substitute the ones defined for conventional layouts. The rest of the formulas hold for blocked layout, as well.

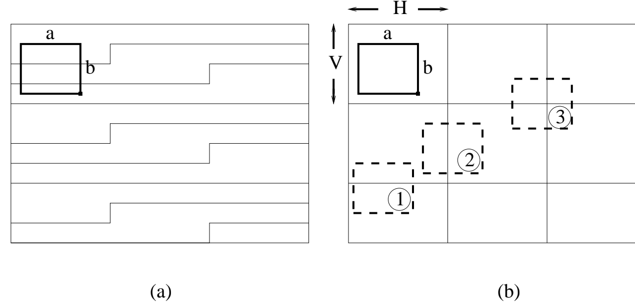


Fig. 4.2. (a) Row-major memory layout. (b) Block-based memory layout. Note that block layout fits very well in the access pattern of video applications (which is also block based). Such a layout reduces the number of page breaks, thereby reducing energy consumption.

Data Layouts in Memory: When block memory layout is adopted, this is not trivial. Let us assume that the array is divided into $H \times V$ blocks. Consequently, we have a total of $C_0 \times C_1$ blocks, where $C_0 = \lceil d_0/V \rceil$ and $C_1 = \lceil d_1/H \rceil$. Then, an array element referred by the index function $F_u(l)$, where l is a valid iteration point, can be found at location $m = \mu_x + P * C_1 * n_0 + P * n_1 + H * o_0 + o_1$ in the linear memory. This can be expressed as:

$$m = L_x(F_u(l), \mu_x) \models \exists n_0, n_1, o_0, o_1 :$$

$$0 \leq n_0 < C_0 \quad \wedge \quad 0 \leq n_1 < C_1 \quad \wedge$$

$$0 \leq o_0 < S_0 \quad \wedge \quad 0 \leq o_1 < S_1 \quad \wedge$$

$$F_u(l) = [V, H] * [n_0, n_1]^T + [o_0, o_1] \quad \wedge$$

$$m = \mu_x + P * C_1 * n_0 + P * n_1 + H * o_0 + o_1.$$

In this expression, $[n_0, n_1]$ is the block coordinates for $F_u(l)$, and $[o_0, o_1]$ is the offset coordinates within a block. In other words, each array element is identified using a block coordinate and an offset coordinate.

Page Break Model for Blocked Data Layouts: Let B_{size} represent the size of blocks ($= [b_0, b_1]$) for the reference R_u. Our model consists of three formulas to count two types of intra-page breaks and one type of inter-page breaks.

Type 1 Intra-Page Breaks: Type 1 intra-page breaks can occur two different ways ; case 1 (marked 1 in Figure 4.2 (b)), when the lower left corner of the block, $F_u(i) + [b_0, 0]$, reside in a different page from the upper left corner, $F_u(i)$, and the upper right corner of the block, $F_u(i) + [0, b_1]$, in the same page as the upper left corner, and case 2 (marked 2 in Figure 4.2 (b)), when the lower left corner of the block reside in the same page as the upper left corner, and the upper right corner of the block in a different page from the upper left corner. We can write this as follows:

$$\begin{aligned}
((R_u, i) \in IntraPageBreakType1(L)) \models i \in I \quad & \wedge \\
\exists b, r : Map(L_x(F_u(i), \mu_x), b, r) \quad & \wedge \\
\exists r' : ((Map(L_x(F_u(i) + [b_0, 0], \mu_x), b, r') \quad & \wedge \\
(Map(L_x(F_u(i) + [0, b_1], \mu_x), b, r))) \quad & \vee \\
(Map(L_x(F_u(i) + [0, b_1], \mu_x), b, r') \quad & \wedge \\
(Map(L_x(F_u(i) + [b_0, 0], \mu_x), b, r)))) \quad & \wedge \\
& (r \neq r').
\end{aligned}$$

Type 2 Intra-Page Breaks: Type 2 intra-page breaks occur when a block falls on the four-edged corner as in the block marked (3) in Figure 4.2 (b). We can identify these types of page breaks by examining if both lower left corner and the upper right corner reside in two different pages from the upper left corner:

$$
\begin{aligned}
((R_u, i) \in IntraPageBreakType2(L)) \models i \in I \quad \wedge \\
\exists b, r : Map(L_x(F_u(i), \mu_x), b, r) \quad \wedge \\
\exists r', r'' : (Map(L_x(F_u(i) + [b_0, 0], \mu_x), b, r') \quad \wedge \\
Map(L_x(F_u(i) + [0, b_1], \mu_x), b, r'')) \quad \wedge \\
(r \neq r' \neq r'').
\end{aligned}
$$

Inter-Page Breaks: Inter-page breaks occur when an access made by reference R_u touches a different page from the page accessed by the last element of the preceding reference, R_v. In mathematical terms:

$$
\begin{aligned}
((R_u, i) \in InterPageBreak(L)) \models i \in I \quad \wedge \\
\exists b, r : Map(L_x(F_u(i)), b, r) \quad \wedge \\
\exists j, r' : j \in I \quad \wedge \quad (R_v, j) \prec (R_u, i) \quad \wedge \\
Map(L_y(F_u(j) + [bv_0 + bv_1]), b, r') \quad \wedge \quad (r \neq r') \quad \wedge \\
\neg(\exists k, w : (R_v, j) \prec (R_w, k) \prec (R_u, i)).
\end{aligned}
$$

Counting Page Breaks: Type 1 intra-page breaks or inter-page breaks incur only one page break. In contrast, for a block with type 2 intra-page breaks, three additional

pages need to be accessed to fetch the complete block. So, we multiply the number of type 2 intra-page blocks by three. Therefore, the total number of page breaks can be obtained by adding the number of type 1 intra-page breaks and inter-page breaks, and three times the number of type 2 intra-page breaks. We use the Omega Calculator to simplify the formulas above, and Ehrhart polynomials to count the number of page breaks, as can be found in Section 4.3.

4.4.3 Discussion

In this subsection, we discuss two important issues regarding our approach. First, it is important to study how the block-based memory layout should be represented. It should be observed that the address translation from the conventional (row-major) layout to the block-based layout involves divisions and modulo operations:

- relative address of $A[i][j]$ in row-major layout:

  $i * N + j$

- relative address of $A[i][j]$ in block-based layout:

  $((j/H) + (i/V) * n) * P + (i\%V) * H + (j\%H)$.

Since H and V are assumed to be powers of two in this study, all multiplications (except perhaps the one with n) and divisions can be implemented using shift-left and shift-right operations. The two modulo operations can be implemented easily as well, since the results are the shifted values. However, when H and V cannot be expressed as powers of two, divisions and modulo operations cannot be easily removed. As suggested

by Anderson et al [10], simple code optimization techniques such as loop invariant removal and induction variable recognition/elimination can move some of the division and modulo operators out of inner loops, thereby reducing their negative impact. In fact, as demonstrated in [10], modulo and division operations can be converted into linear additions (inserting conditional statements when necessary to handle boundary cases). In [39], the modulo and division overhead can be almost completely removed by their ADOPT technique, which combines an algebraic transformation exploration approach to a technique for reducing the piece-wise linear indexing to linear pointer arithmetic. So we believe that implementing block-based layout in the code does not incur much overhead.

We now explain how our approach can be embedded within a compilation framework. As pointed out earlier, our approach gives a count of the number of page breaks for a given H, V and a, b parameters. In order to embed our approach within an optimizing compiler, two tasks need to be performed. First, assuming that we do not change the a, b values (though it is also possible to do so), the compiler has to generate a set of suitable H, V pairs and use our implementation to count the number of page breaks. In determining such H, V pairs, the compiler can follow different methods. For example, one simple strategy is to try only pairs $H = a_i$ and $V = b_i$, where a_i, b_i represents the (block) access pattern in nest i. In this way, we try only m alternatives, where m being the number of nests in the application, and select the one that generates the minimum number of page breaks (considering all the nests). Designing and implementing more sophisticated strategies for determining suitable H, V pairs to try is in our future agenda. The second task that needs to be performed is to move the estimation process

to the compiler. This can be achieved using the Omega Library instead of the Omega Calculator.

4.5 Experiments

4.5.1 Setup

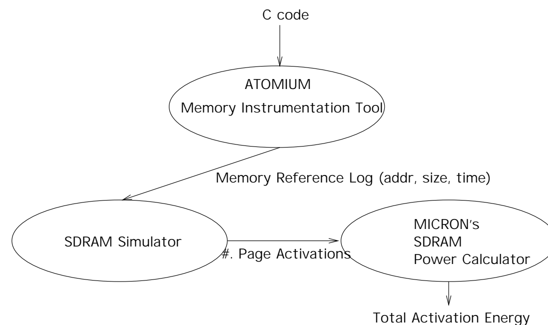


Fig. 4.3. Our simulation setup. Simulated results were compared with our estimations.

Figure 4.3 shows our experimental setup. To evaluate the accuracy of our estimations, we compared them to actual simulation results. The simulation results have been obtained by using the Atomium tool [21]; that is, we fed our benchmarks to the Atomium tool to generate burst access logs to SDRAMs [21]. Each line of the log is composed of memory address, size (of request), and time-stamp. This log is run through the SDRAM simulator that produces the number of memory pages activated by the program [29]. Then the Micron's Power Calculator is employed to obtain the energy consumption [1]. Page activation energy (E_ACT) and standby energy (E_STAT) are two major energy

components in current SDRAMs. In this study, we focus on these components, since the energy for data transfer remains unchanged regardless of the data layout used. E_ACT includes the energy to precharge and activate a memory page and can be calculated using:

$$E_ACT = (IDD0-IDD3) * Trc * VDD * clock_period * number_of_activations,$$

where IDD0 and IDD3 are average activation current and standby current, respectively. Trc is the minimum time required between two successive row activations to the same bank. By reducing the number of activations (number_of_activations), we can reduce E_ACT. On the other hand, E_STAT can be calculated as follows:

$$E_STAT = IDD3 * VDD * clock_period * total_cycles.$$

Note that minimizing the number of page activations reduces E_STAT as it reduces the total number of cycles (total_cycles). The simulated energy numbers have been obtained by using the framework in Figure 4.3. On the other hand, the estimated energy numbers have been obtained by first estimating the number of page breaks as explained in this chapter, and then by feeding this estimation to the two energy equations given above. For the experiments, we assume four SDRAM banks (each is 2MB) with a 32-bit wide bus and 1KB pages as the Micron's model provides. We also assume the existence of a local memory (for both row-major and tile-based memory layouts). Since our model works for data dependent blocks as well, we executed our simulations with a selection of different images.

4.5.2 Benchmarks

We use three benchmarks to illustrate three representative behaviors: qsdpcm (a quadtree-structured motion estimation application), phods (a parallel hierarchical motion estimation application), and edge_detect (an edge detection code) [68]. For each application, we used several sets of the form (block_1, block_2,..., block_n, `tile_shape`), where block_i is the access pattern (i.e., the (a,b) block) used in the i_{th} nest of the code, and `tile_shape` is the H, V pair (i.e., the block dimensions in the tile-based layout). In qsdpcm, the four sets are considered: set_1 = (16×4, 16×2, 18×15, 18×1, 16×2, `64×16`), set_2 = (16×4, 16×2, 18×15, 18×1, 16×2, `32×32`), set_3 = (16×4, 4×8, 18×15, 18×1, 4×8, `64×16`), and set_4 = (16×4, 4×8, 18×15, 18×1, 4×8, `32×32`). In set_3 and set_4, we replaced 16×2 blocks by 4×8 blocks. In phods, for each macro-block, x-axis and y-axis are searched independently to find the best motion vectors for each axis. This process is repeated for three passes of different granularities (4, 2, and 1) of step sizes. For the initial step size of 4, there are no motion vectors involved. However, for the remaining step sizes, the previously calculated motion vectors are used to find a better match. Accordingly, three sets are chosen: set_1 = (16×4, 16×2, 16×1, 4×16, 2×16, 1×16, `64×16`), set_2 = (16×4, 16×2, 16×1, 4×16, 2×16, 1×16, `32×32`), and set_3 = (16×4, 16×2, 16×1, 4×16, 2×16, 1×16, `16×64`). We also include a set with row-major data layout: row-major = (16×4, 16×2, 16×1, 4×16, 2×16, 1×16, row-major layout). Finally, in the edge detection application, 3×3 sized filters are multiplied with the input image one after another. Therefore, we choose three sets: set_1 = (3×3, `16×16`), set_2

= (3×3, 32×8), and set_3 = (3×3, 64×4). In this code, every element takes four bytes, so we have rather smaller tile dimensions compared to the previous two benchmarks.

4.5.3 Results

Figure 4.4 shows the estimated and the simulated number of page breaks for phods. Our estimation shows very good estimation result for row-major layout. We plan to conduct more experiments for conventional layouts.

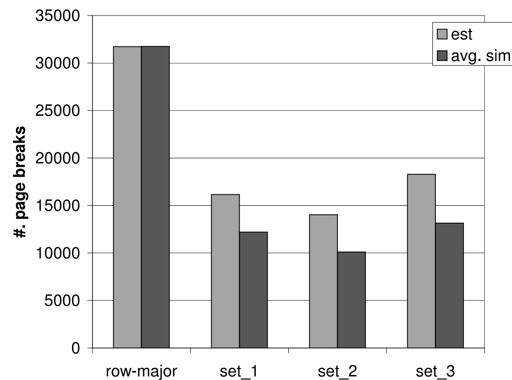


Fig. 4.4. Estimated and simulated number of page breaks for phods .

With an original image size of 176×144 bytes, the improvement with the block-based layout is not pronounced, since most of the rows of a block are stored in the same page. PDAs such as PalmPilot these days have 160×160 resolution with 4 bytes per pixel, which equals to 640×160 bytes. In this case, less than two rows of the image can be mapped to the same SDRAM page, assuming 1KB page size. We also expect

the resolution will increase in the near future as RAMs become cheaper and smaller. Figure 4.5 shows the SDRAM energy consumption for three versions of the qsdpcm application. For each energy component, the bars in this graph represent 800×640 and 176×144 with row-major data layout (original version), and the block-based (tile-based) data layout (64×16) in that order. As pointed out earlier, the performance and energy consumption of the original layout is largely dependent on the image size. In fact, when the image size is small (176×144), there is little difference with the tile-based layout. However, the tile-based memory layout shows no difference at all with varying image sizes. The fact that the results for block-based data layout are not affected by the image size (i.e., uniformly good performance for different array sizes) is another benefit of this layout, since the same result is expected without any code change for various image sizes.

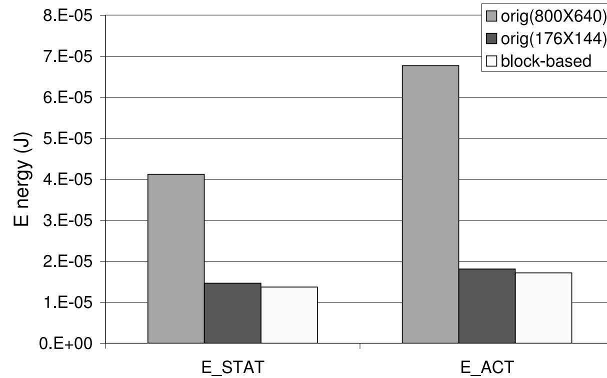


Fig. 4.5. SDRAM energy breakdown for three versions of qsdpcm. A tile (block) size of 64×16 was assumed for the block-based code (the last bar for each energy component).

In Figure 4.6, the estimated (left bars) and the simulated (right bars) energy consumptions are given for each of our benchmarks. In the first benchmark, qsdpcm, we can safely conclude that the set_1 is our choice. If we had arbitrarily chosen set_2, we would have spent more than half of overhead energy compared to the set_1. The figure also shows that our estimated numbers follow the simulated results very closely, considering that dynamic references of blocks that are dependent on motion vectors constitute approximately half the amount of total block references in the driver. In our second benchmark, phods, the estimations show around 70-80% accuracy compared with the simulated results. Most of the page breaks here are incurred by dynamic block accesses. Accesses by step size of 4 are the only static accesses (i.e., no motion vectors involved) but they are perfectly aligned with the tiles, incurring no intra-page blocks. And, the number of their inter-block page breaks is negligible. The result obtained by our formulas is the averaged result over the motion vector range of [-4, 4] for 16×2, and 2×16 blocks, and [-6, 6] for 16×1, and 1×16 blocks. Even though we observe a larger discrepancy between the estimated and the simulated results than the previous example, the results show that our estimation gives 32×32 as the best tile size, which is the same results returned by the simulation. In the last benchmark, edge_detect, all references are static, involving no dynamic vectors. Note that for static block references, we can obtain exact number of page breaks using our formulas.

Our formulas provide estimation results in constant time regardless of data set sizes and the images the simulations would run with. On the contrary, simulations would not be plausible when the data set size is huge, generating prohibitively large memory

reference logs, and taking long period of time. Finding proper sample data points would also be difficult.

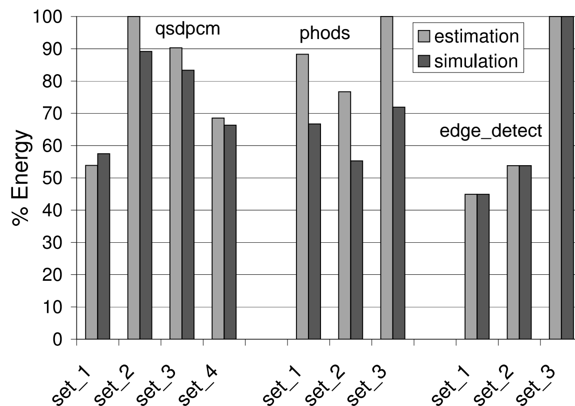


Fig. 4.6. Comparison of the estimated (left bars) and the simulated (right bars) energy consumption for the three benchmarks, qsdpcm, phods, and edge_detect (from left to right).

4.6 Conclusion

This chapter presents a mathematical formulation, which allows us to count the number of page breaks for both conventional and block-based data layout, and studies the impact of block-based memory layout in SDRAM energy consumption. The experimental results obtained using three benchmark codes indicate that our estimation strategy

is accurate and block-based memory layouts reduce the number of page breaks significantly. Our ongoing work includes embedding our tool within an optimizing compiler and performing experiments with larger dimensional arrays.

Chapter 5

A FRAMEWORK FOR ENERGY ESTIMATION OF VLIW ARCHITECTURES

5.1 Introduction

Energy consumption has become an important issue with the widespread use of battery operated mobile devices. Very Long Instruction Word (VLIW) architectures are becoming popular and being adopted in many DSP and embedded architectures [5]. These architectures are inherently more energy efficient than superscalar architectures due to their simplicity. Instead of relying on complex hardware such as dynamic dispatchers, VLIW architectures depend on powerful compilation technology. Various compiler optimizations have been designed to improve the performance of these VLIW architectures [44, 73]. However, not much effort has been performed at optimizing the energy consumption of such architectures. In this chapter, we present the design and use of a framework to enable more research on optimizing energy consumption in VLIW architectures.

Recently, a number of cycle-accurate energy simulators have been developed for simple RISC, DSP and superscalar architectures [104, 15]. An instruction level energy estimation methodology has also been proposed for exploring different architectural

topologies for VLIW architectures [86]. In contrast, the VLIW energy estimation framework proposed here provides flexibility in studying both software and hardware optimizations. As our energy estimation framework for VLIW architectures is built on top of the `Trimaran` compilation and simulation framework, it has access to various high level and low level compiler optimizations and can easily permit implementation of new compiler optimizations. In this work, our emphasis is on the design of this energy simulator and its use in studying both software and hardware optimizations.

The remainder of this chapter is organized as follows. Section 5.2 summarizes the `Trimaran` tool-set and explains our energy simulator. Section 5.3 presents example usages of our simulator. In this section, a high level compiler optimization technique (loop tiling) and block formation algorithms are evaluated, illustrating the tradeoff between energy and performance. Energy impact of predication is also discussed in the same section. An example of hardware optimization, multiple-banked register files, is evaluated in terms of energy in Section 5.4. Finally, our conclusions are presented in Section 5.5.

5.2 Modeling

`Trimaran` is a compiler infrastructure to provide a vehicle for implementation and experimentation for state-of-the-art research in compiler techniques for Instruction Level Parallelism (ILP) [6]. As seen in Figure 5.1, a program written in C flows through `IMPACT`, `Elcor`, and the cycle-level simulator. `IMPACT` applies machine independent classical optimizations and transformations to the source program, whereas `Elcor` is responsible

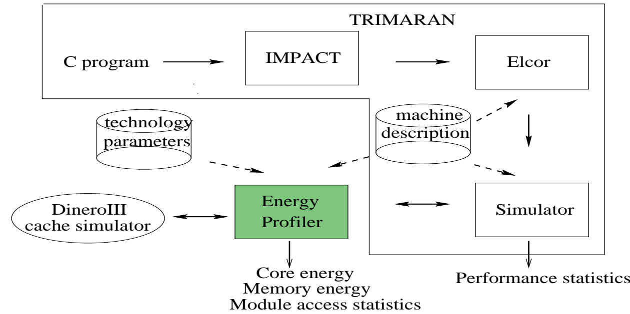


Fig. 5.1. Simulator block diagram.

for machine dependent optimizations and scheduling. The cycle-level simulator generates run-time informations for profile-driven compilations. The cycle-level simulator was modified to instrument the access patterns of different components of the architecture. This profile information was used along with technology dependent energy parameters to obtain the energy consumption of the architecture. Our VLIW energy estimation framework is activity-based; that is, energy consumption is based on number of accesses to the components.

We first enhanced the `Trimaran` framework (see Figure 5.1) to model the cache by incorporating the DineroIII cache simulator [7]. The major components modeled in our energy estimation framework includes the instruction caches, register files, interconnect structures between register files and the functional units, the functional units, data cache, and clock circuitry. We briefly describe the energy modeling of these components.

The energy model proposed in [108] is extended to model different types of memory elements including register files, memory modules and memory conflict buffers (MCB). MCB [38] is a hardware mechanism enabling data dependence speculation to

overcome ambiguous memory dependences [11]. The parameters for the models were extracted from 0.35 μ technology files. It is modeled with a few architectural level parameters such as the number of read and write ports, number of bits per word, number of registers, and several other relatively simple technology parameters. Energy cost of every read and write access is assumed to be independent and, therefore, the total energy is estimated by multiplying the access energy costs by the number of accesses. This model used two bit lines per write port, one bit line per read port, and one word line per every port for the multi-ported memory. Energy for accessing the multiported memory is obtained by summing the energy consumed in wordlines, bitlines, sense amplifiers and cache control circuitry.

The instructions were classified into four types based on the unit used to execute them: integer, floating-point, branch and memory (load/store) instructions. The energy characterization of the different operations performed was extracted from actual layouts of corresponding components. Average power consumption values obtained through HSPICE simulation were used to model the activity based models. This task of characterizing was made easier as we already had access to several of the layouts from our prior energy modeling effort for embedded architectures [96]. The load/store functional units are quite different from other functional units and each of them contains a four entry load/store buffer with 32 bits per entry. In addition to the energy consumed in accessing the buffers and calculating the address, load instructions also account for the data cache access energy and the associated clock energy for the cache. When the load instruction is speculative, MCB write energy is also added. Store instructions are

also treated similarly and account for data cache write energy and the associated clock energy. Load verify instructions for speculative loads are not currently supported.

Interconnects between the functional units and register files are modeled as a multiplexor-based crossbar structure. Clustered and partitioned register file structures are also modeled as explained later in this chapter. The energy consumption of the crossbar structure is estimated using the multiplexor energy extracted from the layouts and wire lengths parameterized based on size of the crossbar (determined by number of ports and functional units).

Clock energy estimation is based on the clock energy model proposed in [33]. The clock generation and distribution circuitry model includes the clock energy consumed by the caches, register files, PLL, clock buffers and wires. It uses analytical models that accept parameters such as number of memory ports, size of cache, number of registers and technology parameters.

Clock energy for the PLL, maindriver, and wiring are added in every clock cycle. Energy costs for instruction cache access and register file precharge energy are also added in every cycle. For each operation, the register file accessed are monitored. For example, if the predicate register file is accessed (i.e. predicate instruction), energy cost for the predicate register file is added. This is done for every source register files accessed and for destination register files which are not predicated or predicated true. A predicate squash logic is used in our simulations to prevent instructions with false predicates from committing [11].

5.3 Evaluation - Software Optimizations

In this section, we investigate energy implications of software optimizations with our simulator described in the previous section. We observe if energy consumption figures show different behavior with those of performance.

5.3.1 High Level Compiler Optimization

In this section, we evaluated high level compiler optimizations in terms of energy and performance. We used a high-level compilation framework based on loop (iteration space) and data (array layout) transformations [50]. This framework applies iteration space tiling and scalar replacement to obtain better temporal and spatial data locality of an input C code. Five benchmarks were evaluated using our energy simulator. A general purpose register (GPR) file size of 32 was used. Other parameters such as general purpose rotating register file size, number of functional units and operation latencies, etc. were remained unchanged from standard `mdes` file in `Trimaran`. 8KB direct-mapped L1 instruction cache and 8KB/256KB two-way associative L1 and L2 data cache were used in all experiments, unless stated otherwise. Cache line size was 32B for all the caches. Inlining and modulo scheduling were activated. Basic block block formation was also applied. Five benchmarks from `Livermore`, `Perfect Club`, `Specfp92`, and `Specfp95` were evaluated using our energy simulator. Energy estimations of optimized benchmarks were scaled over unoptimized ones.

Figure 5.2 and Table 5.1 show the energy and performance result, respectively. `btrix` and `vpenta` show favorable results both in terms of energy and performance by

applying high-level optimizations. Note that total cycles taken is the sum of "non-stall cycles" and "cache stall cycles" column. The optimized bmcm shows a significant drop in cache stall cycles. However, the number of non-stall cycles increases significantly due to more complex loop operations and subscript expressions. It can be observed from the third column of Table 5.1 that the number of operations generally increase after the applied optimizations. Since efficient clock gating can make the dynamic energy consumption in stall cycles negligible, the overall energy consumption of the optimized code increases in spite of the performance enhancement.

The optimized tomcatv exhibits an interesting behavior. The optimized version consumes significantly more energy due to the increased pressure on the registers. The resulting spill code to handle the increased pressure on the registers is observed to account for 60% of the total operations. When the number of GPR is increased to 128 (see Table 5.2), the unoptimized tomcatv takes 70% more cycles than the optimized one and the optimized tomcatv shows less data cache energy consumption because of the reduced data cache access misses.

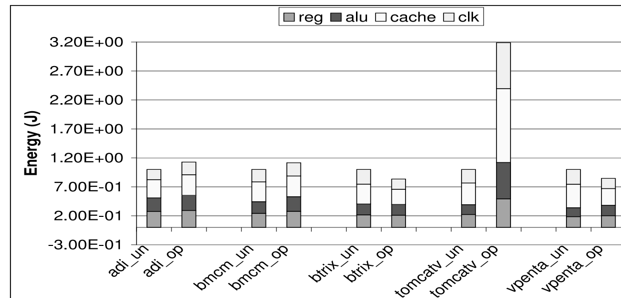


Fig. 5.2. Energy distribution of benchmarks without (benchmark_un) and with (benchmark_op) high-level optimizations.

benchmark	non-stall cycles	cache stall cycles	total # of ops
`adi(un)`	251,844	240,296	1,143,978
`adi(op)`	361,644	288,057	1,203,401
`bmcm(un)`	5,173,871	3,661,301	13,397,774
`bmcm(op)`	6,434,728	3,647	16,418,695
`btrix(un)`	3,129,390	11,021,465	11,316,798
`btrix(op)`	3,053,636	4,988,746	11,186,953
`tomcatv(un)`	450,682	1,113,693	1,795,499
`tomcatv(op)`	3,928,914	256,354	4,968,286
`vpenta(un)`	13,116,352	43,450,792	33,412,265
`vpenta(op)`	14,911,759	1,706,425	38,548,300

Table 5.1. Non-stall cycles, cache stall cycles, and total operations taken for unoptimized/optimized benchmarks.

5.3.2 Block Formation Algorithms

The VLIW processors suffer from insufficient parallelism to fill the functional units available. Block formation algorithms such as superblock and hyperblock are proposed [44, 73]. It is shown that significant performance improvement can be obtained through theses algorithms. We evaluated three block formation algorithms, basic block (BB), superblock (SB), and hyperblock (HB) to see how these algorithms affect the system power.

Superblock Frequently executed paths through the code are selected and optimized at the expense of the less frequently executed paths [44]. Instead of inserting bookkeeping instructions where two traces join, part of the trace is duplicated to optimize the original copy. This scheduling scheme provides an easier way to find parallelism

beyond the basic block boundaries, especially for the control-intensive benchmarks, because the parallelism within a basic block is very limited.

Hyperblock The idea is to group many basic blocks from different control flow paths into a single manageable block for compiler optimization and scheduling using if-conversion [73].

We selected two benchmarks from `Spec95Int` (`129.compress` and `130.li`), two benchmarks from `Mediabench` (`adpcmdec`, `mpeg2dec`), and `DSPStone` benchmarks. For the sake of clarity, the numbers for the `DSPStone` benchmarks were averaged. 128 GPR and 4 integer ALUs were used. Other parameters such as instruction latencies were from `Trimaran`'s standard `mdes` file. In these experiments, the modulo scheduling was used.

Fig 5.3 shows the number of cycles and energy taken for each benchmark. All values are scaled to those of BB with no modulo scheduling case. For `129.compress`, `DSPStone`, and `mpeg2dec`, both energy and performance show similar trend. For `130.li` and `adpcmdec`, there is an anomaly in the SB case. On close examination, it is observed that there is a 15% - 40% increase in the number of instructions executed after the SB formation as compared to the BB. Note that this does not translate to an increase in the number of cycles as the average ILP is increased. However, the increased number of instructions executed manifests itself in the form of increased energy. Another trend that we observed was that SB and HB techniques were not that successful for the DSPStone benchmarks. These benchmarks are quite small and regular. Hence, they do not gain from the more powerful block formation techniques.

Figure 5.4 shows the component-wise breakdown of the energy graph in Figure 5.3 for selected benchmarks. It is observed that the data cache and register file energy costs

increase with SB and HB due to the increased number of instructions executed. However, the instruction cache clock energy decreases, because of the reduction in the number of clock cycles caused by increased ILP.

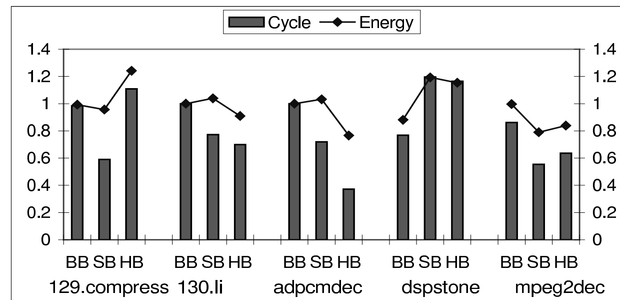


Fig. 5.3. Energy and performance comparison of BB, SB, and HB when modulo scheduling is activated.

5.3.3 Predication

Conditional branches create a control dependency problem limiting the available ILP. Predicated execution is a mechanism of executing instructions conditionally based on the value of a boolean source operand, referred to as the `predicate` [73, 11]. An instruction is executed only when the predicate has `True` value. When the predicated execution support is provided in the architecture, the compiler can eliminate many of the conditional branches in an application, thereby enabling more efficient form for execution on a wide-issue processor. A hyperblock is formed by grouping many basic blocks using

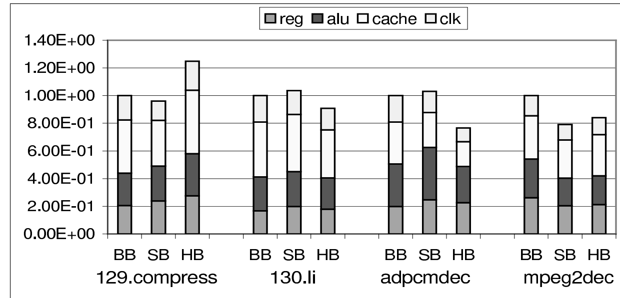


Fig. 5.4. Energy distribution when modulo scheduling is activated. Each bar represents relative energy cost for register files, ALUs, caches, and clock circuitry from bottom to top.

predication [73]. Modulo scheduling in `Trimaran` also utilizes predication in Prolog and Epilog of iterations [6].

We conducted experiments to see how much of the energy is wasted by predicate false instructions (instructions with their predicate register value of `False`). A GPR size of 128 was assumed and `Trimaran`'s `std` parameter file was used. Modulo scheduling was activated. Table 5.3 shows energy consumed by predicate true and false instructions of the total energy. In general, predicate false instructions are not a major energy consumer. The `adpcmdec`, however, consumes more energy in predicate false instructions than predicate true instructions when hyperblock formation is used. We also found that percentage of predicated instructions of the total number of instructions is higher than that corresponding percentage of the energy consumed by the predicate instructions. For example, in `129.compress`, 20% of the total instructions were predicate true instructions, whereas their energy consumption was only 11%.

5.4 Evaluation - Hardware Optimizations

In this section, we show an example of how our tool can be used as an aid in embedded system design.

5.4.1 Multiple Register Banks

With the increased transistor budget and high ILP enabled, more and more functional units are put into processors. This places an excessive pressure on the register file as the number of ports and registers within the register file needs to be large enough to sustain the large number of functional units. This leads to a performance bottleneck. A solution is to partition the register file into multiple register banks. In [84], a taxonomy of register architectures across the data-parallel, instruction-level parallel, and memory hierarchy axes was developed for media processors. They concluded that the most compact of these organizations reduces the register file area, delay, and power dissipation. A register file architecture composed of multiple banks was proposed in [28]. They focused on a two level organization, which is called a register file cache among different multiple-banked organizations. Among the commercial DSP processors, TI C6201 [5] has two clusters of functional units with their local register files. Two cross paths exist to access non-local registers.

We consider a register file organization that trades space for improved energy consumption behavior. Instead of a single monolithic register file for all functional units, the functional units are partitioned into two parts with their own local register file to form two clusters. Additionally, both the clusters have access to a common register file

as shown in Figure 5.5. The common register file is used to store variables that are accessed by functional units in both the clusters. In contrast, the local register files are accessible only to the functional units in the cluster. While the number of registers in the resulting architecture is three times more, the clustered architecture reduces the complexity of the local register files. The number of ports in the local register files are reduced by half as compared to that of the single monolithic register file, since local register files are accessed only by half the number of functional units. The common register file has the same number of ports as the monolithic register file. As the energy consumption of the register file is a function of both the number of ports and the number of registers, the energy cost per access to the local register files is less than that of the common register file and the original monolithic register file. When most of the accesses are confined to the local register file, we can anticipate improvements. We modified the register allocation to exploit the local register file organization.

Figure 5.6 shows the relative energy consumption of the register file architecture compared to the monolithic register file. Hyperblock and modulo scheduling were activated and `Trimaran`'s `std` parameter set is used. All benchmarks show reduced energy consumption as compared to a single monolithic register file. In particular, when the GPR size is 32, it consumes less than half energy than the monolithic register file. It should be noted, however, that we are trading area for energy because we duplicated register files into three pieces. The energy saving comes from the reduced number of read ports (reduced by 4) and write ports (reduced by 2) and less complex interconnects for the local register files.

We also tracked the number of registers that are actually utilized in the local register files and the common register file. We observe that there is scope for reducing the energy consumption of our clustered architecture further by reducing the sizes of the local and common register files. For example, `129.compress` requires only 58 local registers and 24 common registers (see the first column in Table 5.4) as opposed to 64 local registers and 32 common registers as used in our current evaluation.

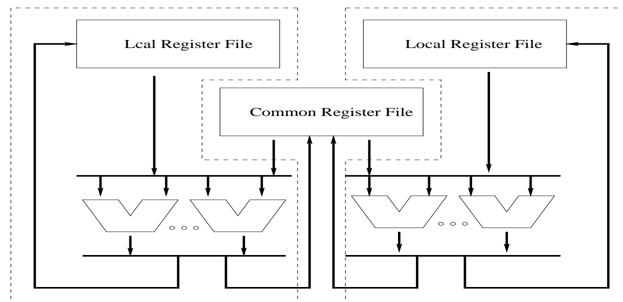


Fig. 5.5. The register file architecture evaluated.

5.5 Conclusion

With the proliferation of portable consumer products, energy consumption has become an important issue. The VLIW architectures are now being used in embedded processor such as TI 'C6x [5] chips as many compilation techniques have succeeded in improving ILP to increase their performance. In this chapter, we presented an energy estimation framework built over the `Trimaran` compilation tool-set and evaluated the

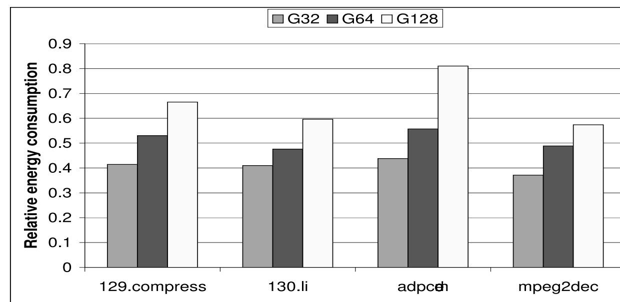


Fig. 5.6. Relative energy consumption of register files (including interconnects between register files and functional units). Each bar represents GPR size of 32, 64 and 128, respectively.

influence of both architectural and compiler optimizations on energy efficiency using the proposed framework.

#. GPR	tomcatv(un) cycles (K)	stalls (K)	total #. ops (K)	tomcatv(op) cycles (K)	stalls (K)	total #. ops (K)
32	450	1,113	1,795	3,928	256	4,968
64	438	1,108	1,781	2,274	346	3,583
128	438	1,108	1,781	542	343	1,983
	data cache energy (J)			data cache energy (J)		
32	1.705854e-04			4.111437e-04		
64	1.689281e-04			2.821131e-04		
128	1.689116e-04			1.219093e-04		
	#. dcache read accesses L1/L2/Main (K)			#. dcache read accesses L1/L2/Main (K)		
32	372	161	18	1252	21	14
64	367	160	18	906	21	14
128	367	160	18	394	21	14

Table 5.2. Performance and energy numbers of unoptimized `tomcatv` and optimized `tomcatv` as GPR size grows.

Benchmark	predicate true (%) BB	SB	HB	predicate false (%) BB	SB	HB
129.compress	10.99	0.00	5.49	0.65	0.00	2.35
130.li	0.01	0.00	4.30	0.00	0.00	3.15
adpcmdec	0.00	0.00	6.62	0.00	0.00	9.26
DSPStone	54.28	3.57	6.04	1.62	1.63	3.40
mpeg2dec	42.47	0.00	0.54	5.91	0.00	0.57

Table 5.3. Energy (%) of predicate true and false instructions when modulo scheduling is activated.

benchmark	maximum number of registers					
	GPR=32		GPR=64		GPR=128	
	local reg.	com. reg.	local reg.	com. reg.	local reg.	com. reg.
`129. compress`	58	24	98	50	165	81
`130.li`	50	29	85	40	131	64
`adpcmdec`	18	29	41	40	62	53
`mpegdec`	50	24	84	51	150	81

Table 5.4. Maximum number of registers used dynamically. Each column represents number of maximum registers required for local register files and common register file as depicted in Figure 5.5.

Chapter 6

ADAPTING INSTRUCTION LEVEL PARALLELISM FOR OPTIMIZING LEAKAGE IN VLIW ARCHITECTURES

6.1 Introduction

The continuing quest for faster and more powerful processors has exacerbated the power problem. With several million transistors switching at very fast clock rates in current microprocessors, power consumption is considered the primary limiter to designing more powerful computing systems. In addition, the technology trend of smaller minimum feature sizes and threshold voltages has made the leakage power consumed by the millions of transistors a major concern. Thus, the need for new features that limit the power consumption of a processor without an adverse impact on performance is imperative.

The current approaches to designing energy-efficient systems comprise a wide spectrum of optimizations spanning circuit design, architectural design, compilation techniques, operating system design, and application tuning [14]. Many researchers have also looked at the interaction between the optimizations at different levels to maximize the energy benefits. In this work, we utilize the interaction between the compiler and the architecture in effecting power control mechanisms that exploit application characteristics. Specifically, our work is based on the observation that there is a wide variation in

the utilization of the processor resources during different phases of executing an application. Many techniques have been proposed recently that adapt the configuration of processor resources such as the cache and instruction issue queue in consonance with the changing application requirements [40, 47, 78, 13, 12]. The goal of these techniques is to transform the processor resources to a minimal configuration from a power consumption perspective while minimizing any adverse impact on performance.

While most existing techniques focus on runtime monitoring techniques to adapt the system configuration, we use an adaptation policy determined at compile time and applied it at run time. This results in eliminating any overhead associated with the runtime monitoring and control mechanisms. Further, our results show that the compile time adaptation policy tracks changes effectively. The focus of our work is on exploiting the idleness in functional units of wide-issue VLIW architectures. The basis of our work stems from the observation that there is an inherent variation in the maximum number of instructions that can be executed per cycle when executing an application.

Figure 6.1 shows this variation for one of the applications used in this work. The degree of this variation depends on both the application characteristics as well as the processor resources. Whenever fewer instructions are issued in a cycle, many of the functional units are idle and can be transitioned to a low-power state to conserve energy. Transitioning to a low-power state can be as simple as clock-gating the inputs of the unused unit to conserve dynamic energy or more sophisticated such as the supply-gating of the functional unit to reduce leakage energy consumption. In this work, our focus is on reducing leakage energy that is projected to become the dominant part of the chip power budget at high temperatures beyond the 0.1 micron feature sizes [22]. The application of

the different low-power control mechanisms has different energy reduction potential and overheads. Typically, the more the energy savings, the more the performance penalty for recovering from the low-power state to the active state. Thus, a larger duration of idleness can employ a more energy efficient low power mode.

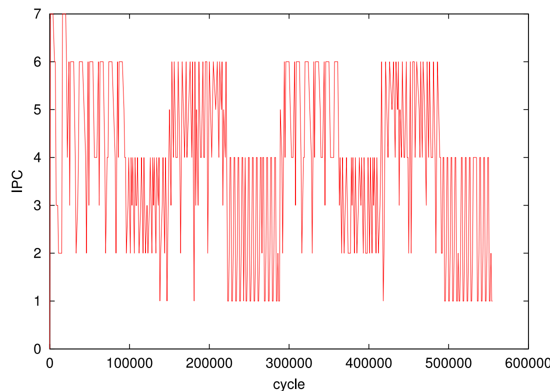


Fig. 6.1. IPC variation (`eflux`).

Using the Trimaran simulation framework [6] and a set of array-dominated applications, we use the compiler to identify the number of functional units to activate for the different loops in the application. Such array-dominated applications and VLIW architectures are frequently used in mobile embedded environments. Based on compiler analysis, we apply leakage control mechanism to the unused units to conserve energy. Our simulation results show that our technique can reduce up to 38% of the functional unit leakage energy averaged across a range of system configurations. Our results also show that our loop based IPC detection strategy gives better energy-delay product than

finer-granularity (basic block level) and coarser-granularity (whole application level) IPC detection schemes.

The rest of this chapter is organized as follows. The next section describes related work. Section 6.3 discusses our approach that adapts the IPC to the needs of a given application. Section 6.4 presents the experimental results showing the effectiveness of our approach. Finally, Section 6.5 presents our conclusions.

6.2 Related Work

While leakage reduction of storage structures [103, 54, 107] has been the focus of bulk of the architectural investigation due to their dominant transistor budget in a processor, there have been various recent efforts at focusing on mitigating leakage in the functional units. This trend is because of the following reasons. The leakage current in the combinational logic circuits employed in functional units is an order of magnitude larger than that of logic circuits employed for cache RAM transistors (based on the model proposed by [18]). In addition, leakage current increases with increase in temperature; and the functional units, due to their heavy usage, have a higher temperature profile. Thus, the functional units contribute to a noticeable fraction of leakage power consumption despite their relatively fewer number of transistors. The exact percentage is difficult to estimate without detailed knowledge of the processor configuration, the underlying circuit styles and thermal profile. This percentage was around 30% of static leakage in a target embedded processor configuration with on-chip L1 caches for which we had access.

In [105, 81, 106], the slacks are exploited for leakage reduction in functional units. All these techniques are based on using a compiler to insert instructions to control the leakage modes using either input vector control or supply gating. While these techniques exploit existing idleness in the schedule, our approach changes the IPC and attempts to balance the tradeoff between performance and leakage energy. In addition, in contrast to these previous efforts, our approach specifically targets loops which serve as natural boundaries for IPC variation. [32, 95] propose leakage reduction techniques for super-scalar architectures, whereas our focus is on a VLIW architecture. The technique in [13] is a micro-architectural level throttling mechanism activated dynamically, in contrast to our static compiler-based strategy. Note that our strategy is more suitable for a VLIW architecture as the compiler has the complete schedule.

6.3 IPC Adaptation

In this work, a program is represented using a control-flow graph (CFG). In a CFG, each node is a basic block (i.e., a sequence of instructions which can be entered from one point and exited from one point) and a directed edge from a node to another indicates the possibility of a control flow from the basic block represented by the former node to the one represented by the latter. Each basic block in the CFG can in turn be represented using a data-flow graph (DFG). Each node in a DFG is an operation (that will be executed in a VLIW functional unit) and an edge from one to another indicates a data dependence between the corresponding operations. We use the term instruction to refer to a VLIW instruction which might contain a number of operations scheduled to be executed in the same cycle. Also, we use the term VLIW slot to denote a point in

the two-dimensional scheduling plane, where the y-axis corresponds to execution cycles and the x-axis corresponds to the number of functional units.

Not all parts of a given VLIW program can issue the maximum instructions per cycle. Due to data dependences and other reasons, it is typical that many VLIW cycles can execute fewer operations than can be accommodated by the physical resources. In this chapter, we exploit this characteristic and present a compilation strategy that adapts IPC to the needs of the application being executed.

To demonstrate why such an approach might be successful, let us consider the DFG shown in Figure 6.2(a) to be scheduled in a VLIW machine with 4 integer ALUs. Assume that each operation is an integer operation and once scheduled executes in a single cycle. We observe two things from this DFG. First, its inherent IPC is 2; that is, at most two instructions can be scheduled in each cycle. Second, if we schedule two instructions per cycle, we obtain a highly balanced schedule. These two observations motivate us to use only two integer ALUs in executing this DFG. The remaining two integer ALUs can be placed into a leakage control mode to save energy. Note that such a strategy allows us to execute this DFG in an energy-efficient manner without loss of instruction level parallelism. Although the DFG shown in Figure 6.2(b) does not use all four integer ALUs in each cycle either, trying to use only two ALUs here would increase the schedule length, as the inherent IPC in this DFG enables execution of four operations in some cycles. Consequently, it is not possible to shut off functional units here without incurring some performance penalty.

In this chapter, we try to save leakage energy by shutting off functional units when they are not needed. While in principle this idea can be applied to any functional

unit, in this chapter, we focus on integer ALUs as they are the most frequently exercised functional units in applications that manipulate integer data. Therefore, when we mention IPC in the remainder of this chapter, we refer to IPC considering only integer ALUs. So, in this context, if we have k integer ALUs in the architecture, we say that the maximum IPC is k. If we shut off k' of these integer ALUs during the execution of a program region, we say that the maximum IPC in that region is $k - k'$.

Our approach is composed of the following three steps which will be detailed in the remainder of this section:

- Loop identification
- IPC assignment
- Adaptive scheduling

In this work, we explicitly focus on array-intensive applications. In these applications, bulk of execution time is spent within loops, and a suitable IPC assignment for basic blocks in the loop can lead to large savings in leakage energy.

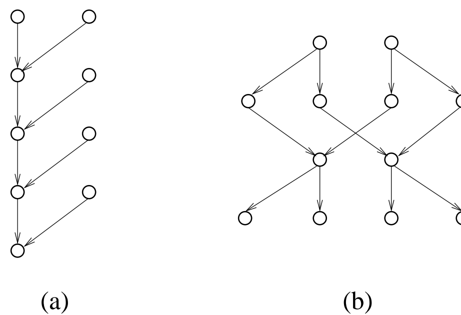


Fig. 6.2. Example DFGs.

6.3.1 Loop Identification

Our approach takes input a CFG which might come in two different forms. If no hyperblock formation [73] has been performed on the CFG, it is easy to find the loops in the code. There are numerous algorithms published on this topic; we refer the reader to [74] for an in-depth discussion of this subject. If, on the other hand, the hyperblock formation has been performed on the CFG, we first need to break hyperblocks into basic blocks and then find the loops in the CFG. Fortunately, Trimaran, our experimental platform, provides a module to achieve this. In either case, at the end of this step, we identify all loops in the input CFG.

6.3.2 IPC Assignment

In the second step, we assign an IPC to different program regions. While it is possible to design a very fine-granular IPC assignment strategy, frequent turning on/off activities can also consume significant dynamic energy as well as extra execution cycles. Consequently, we chose to leakage-control ALUs at the loop granularity. More specifically, for each loop, we assigned a single IPC. Since the number of loops in a given application is rather limited, such an approach is expected to incur a small energy and performance overhead in practice.

However, a loop in general might contain multiple basic blocks (e.g., due to conditional control flow within the loop body). Consequently, we need to have a conflict resolution mechanism when different basic blocks in the loop body demand different IPCs. We solve this problem by profiling the loop and determining the most frequently executed basic block. Once this basic block has been determined, we compute an IPC

(as explained below) which is most suitable for it and use the same IPC for the other basic blocks in the loop as well.

In selecting an IPC for the most frequently executed basic block, we take the following approach. First, we determine the critical path in the DFG of the basic block. The critical path corresponds to the longest dependence chain in the DFG and determines, in a sense, the minimum achievable execution time (if one does not consider data speculation). We record the operations in the critical path and (optimistically) assume that the other operations (i.e., those not on the critical path) exhibit a uniform distribution along the time slots. Let us consider the following scenario to clarify this concept. If we have n operations in the DFG and $m \leq n$ of these are on the critical path, we (optimistically) assume that data dependences in the DFG would allow execution of approximately n/m operations per cycle. If this is the case, then an IPC of n/m operations should be sufficient to maintain the level of parallelism that would be achieved even if we use all $k > n/m$ functional units in the architecture. To summarize, our approach determines m and and shuts off $k - n/m$ ALUs.

In cases that there are multiple basic blocks in a given loop with the highest execution frequency (the same frequency), we select the one with the largest inherent IPC. In handling a nested loop, our current implementation selects a (potentially) different IPC for each level. Our experimentation shows that such an approach in general generates better results than selecting a single IPC for all loops in the nest. Finally, for basic blocks that are not enclosed by any loop in the CFG, we use the maximum available IPC supported by the architecture.

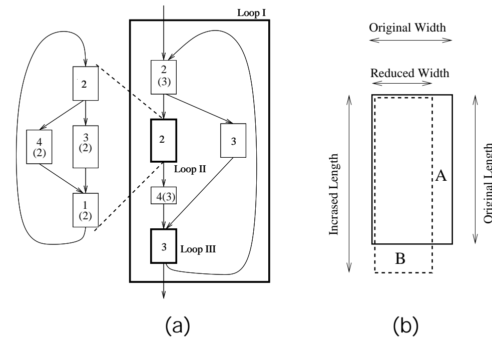


Fig. 6.3. (a) Example CFG fragment. (b) Original and modified schedules.

As an example, let us consider the CFG fragment shown in Figure 6.3.(a). In this figure, there are three loops (delimited by bold rectangles). The other rectangles in the figure denote basic blocks. The numbers inside rectangles (without parentheses) denote the IPCs most suitable for the corresponding basic blocks. Let us first focus on Loop II, which is expanded to the left hand side. This loop has four basic blocks and the first and the last basic blocks are executed with the same frequency. Since the determined IPC of the first basic block is higher, the IPC for the entire loop is set to 2. This value is written inside the block that represents Loop II within Loop I. Let us assume that using a similar method we determined that the IPC for Loop III should be 3. Now, we concentrate on Loop I. This loop has five blocks and two of them are loops. In determining the IPC for this loop, we check the execution frequencies of these five blocks and select the IPC of the most frequently accessed basic block. Assuming that the execution frequency of Loop III is higher than that of Loop II, we determine the IPC for Loop I as 3. Note, however, that this IPC does not apply to Loop II as each loop has its own IPC (determined by considering the basic blocks it contains). The IPC numbers

determined for each block are written inside parentheses if they are different from the original ones in the Figure 6.3.(a).

This energy-saving strategy can in general increase execution time as it is not always possible to find a suitable IPC (lower than the underlying machine can sustain) for each loop without increasing its schedule length. Increasing the schedule length also has an impact on energy consumption as all active functional units consume leakage energy during the extra cycles (coming from the reduced IPC). In order for our strategy to be beneficial from an energy perspective, this energy increase (due to extra execution cycles) should not offset the energy gains obtained through shutting off some ALUs. To illustrate this last point, let us consider Figure 6.3.(b). This figure shows a schematic description of the original schedule and the modified (optimized) schedule. As compared to the original schedule, the optimized schedule reduces the width (by restricting the maximum IPC) and possibly increases the length. Let A denote the total number of slots for which ALUs remain active in the original schedule but not in the optimized schedule and let B denote the total number of slots required due to the extra cycles in the optimized schedule. Let A' and B' denote the corresponding leakage energy expended due to slots in A and B. Clearly, our strategy makes sense if A' > B' and the number of extra cycles (i.e., the performance penalty) is within a tolerable limit. Consequently, in our evaluations, we need to measure both energy gains and performance impact.

6.3.3 Adaptive Scheduling

After assigning an IPC to each loop, the next step is scheduling the code being optimized considering these IPCs. Our scheduling strategy is adaptive as it changes the

maximum IPC during the course of execution. In this step, we traverse the CFG block by block, and for each block we schedule its operations using the IPC assigned to it. We also insert functional unit turn on and turn off instructions at the beginning and end of the associated loop considering its assigned IPC and the available integer ALUs in the architecture. As compared to dynamic IPC regulation strategies, our approach is expected to incur much less overhead as IPCs are determined at compile time. The only runtime overhead is the execution of turn on/off instructions and extra execution cycles and energy due to increased schedule length. Our scheduling strategy is loop-based, and schedules each loop separately. In Trimaran, it has been implemented as a pass between modulo scheduling [80] and acyclic scheduling. More specifically, after modulo scheduling is run, our adaptive scheduling is activated, and turn on/off instructions are inserted in the code. Following these, the acyclic scheduling module is run.

6.3.4 Hardware Requirements and ISA Support

As discussed earlier, when we set the IPC to k', the $k - k'$ integer ALUs are shut off (assuming that the machine contains a maximum of k integer ALUs). When (e.g., in executing the next nest in the code) we increase the IPC from k' to k'' where $k' < k'' < k$, we need to turn on $k'' - k'$ integer ALUs. To support explicit functional unit turn on and off, all the compiler needs is a sleep signal per integer ALU. Using this sleep signal, the compiler transitions an integer ALU from the active mode to a leakage control mode or vice versa. We also assume that this sleep signal is controlled using special instructions. This can be achieved by augmenting the instruction set architecture (ISA) of the architecture.

Three different techniques can be used for runtime leakage control of the functional units, namely, input vector control (IVC), substrate biasing (SB), and supply gating (SG) [22]. The effectiveness of these schemes varies based on the functional unit and the exact implementation mechanism. In all these mechanisms, we assume the availability of a sleep signal that can be set under the control of the VLIW schedule word to initiate the low leakage mode transition and the reactivation to the normal active mode.

The goal of the input vector scheme is to find an input to the functional unit that maximizes the leakage reduction benefit from the transistor stacking effect. According to this effect, sub-threshold leakage can reduce by a factor of up to 10 when two transistors in a stack are turned off instead of just one transistor [22]. Once a primary input vector that minimizes the leakage is found, the input to the functional is forced to this whenever the sleep signal is asserted. At other times when sleep signal is deactivated, the unit receives its normal input.

The goal of substrate biasing mechanism is to dynamically modify the threshold voltage during runtime; a classic example is a technique called standby power reduction (SPR) or variable threshold CMOS (VTCMOS). In this technique, the threshold voltage is raised when the sleep is signal is asserted by making the substrate voltage either higher than Vdd (P devices) or lower than ground (N devices). When the sleep signal is deactivated, the bias is removed returning the device to a lower threshold voltage to provide better performance.

The last approach for runtime leakage reduction that we consider is power supply gating. The basic idea is to shut down the power supply so that idle units do not consume leakage power. Here the activation of the sleep signal cuts off the power supply rail to

the functional unit and upon deactivation of the sleep signal, the normal supply voltage is restored.

All these schemes have the following four phases: sleep activation time (SAT), settling time (STT), low power time (LPT), and sleep deactivation time (SDT). SAT is the time required for the sleep signal to propagate and the logic state of the block to change. STT is the time for the internal node voltages of the functional unit to move such that the steady state (low leakage current state) is reached – this time is in the 10-100 ns range for both IVC and SG [22], and it depends strongly on temperature and transistor leakage of the technology. LPT is the time spent in low power mode. And finally, SDT is the time required to propagate the reactivation signal and for the logic state of the functional unit to be restored.

In order to model the leakage energy and overheads associated with our scheme, we have designed different integer ALU components using 0.25 micron technology. The threshold voltage of the P/N devices used in the simulation was 0.47V/.59V respectively and a 2.5V supply voltage was used. The average leakage power consumed by the designs of the three important integer functional unit components adder/subtracter, 8-bit multiplier, shifter were 93.34nW, 348.4nW and 936nW, respectively. We also model the leakage in other logic components and multiplexer components.

Next, to incorporate energy savings and overheads associated with the different schemes, we performed circuit level simulations. First, for IVC, 180 random input patterns were used for each unit to fit a Gaussian distribution of the leakage measurement. The resulting leakage energy distribution was used to determine the average leakage energy savings when using the least leakage current producing input. In [42], 59 random

vectors were shown to provide a 95% confidence of finding the input vector that provides the least leakage current provided that the input pattern generated a Gaussian distribution of the leakage current. The average percentage leakage energy savings obtained by IVC across the different functional units was then evaluated and incorporated in the simulator. An average percentage leakage energy savings using IVC was 70%, ranging from 98% for a 32-bit logic AND design to 24% for an pass-transistor based XOR design. The SAT and SDT values in our design require 1 cycle. Further, we include the dynamic energy overhead associated with switching the inputs using IVC. Note that, since our control is done at loop level granularity, the impact of this overhead is minimal. For SB, we used VTCMOS for body bias control, the percentage leakage energy savings across the different design was 70%. The SAT and SDT values were 40 cycles; these values are primarily determined by the charge pump used and is modified by appropriately sizing the driver. The additional energy overhead in this case is the energy consumed by the charge pump. As mentioned earlier, this overhead is not significant as it is amortized over the entire loop execution. Finally, in our implementation of SG, we use a Phase Locked Loop(PLL) connected with a voltage follower as the voltage regulator. The PLL is shared by all functional units and can regulate the supply voltage level globally while a voltage follower is assigned to each functional unit and can be controlled locally. Note that our intent is not on studying supply voltage regulation but on using it to gate power supply to the functional units. Supply voltage regulation is an orthogonal issue to this study. The percentage leakage energy saving, SAT, and SDT values for SG are 90%, 50 cycles, and 50 cycles, respectively. The SAT, SDT values, and energy savings extracted

from the actual circuit simulation were incorporated into the Trimaran simulator to capture the leakage energy savings as well as the performance penalties associated with the schemes.

Benchmark	**Source**	**# of Nests**	**# BBs**	**% of Int Ops**	**% of BBs**
`adi`	Livermore	2	17	72.9	70.1
`apsi`	Perfect Club	3	25	72.1	70.0
`bmcm`	Perfect Club	4	25	62.0	97.6
`btrix`	Spec	7	43	71.5	82.9
`eflux`	Perfect Club	2	43	66.9	95.6
`mxm`	Spec	2	17	61.9	98.5
`tomcatv`	Spec	9	51	68.6	87.0
`tsf`	Perfect Club	4	38	62.1	99.0
`vpenta`	Spec	8	41	66.2	95.4
`wss`	Perfect Club	7	39	66.7	96.3

Table 6.1. Array-intensive benchmark codes used in our experiments.

6.4 Experiments and Results

To test the effectiveness of our IPC detection and adaptive scheduling strategy, we implemented it using the Trimaran infrastructure [6] and made experiments using several array-dominated applications. In the Trimaran compiler infrastructure, a program written in C flows through IMPACT, Elcor, and the cycle-level simulator. IMPACT applies machine independent classical optimizations to the source program, whereas Elcor is responsible for machine dependent optimizations and scheduling. In our implementation, modifications are performed to the Elcor module. The modified schedule is then fed to

the cycle-level simulator that is used to quantify our energy savings. All applications used in this study are also optimized using loop unrolling [74] to increase instruction level parallelism as much as possible.

Important characteristics of our applications are given in Table 6.1. The third and fourth columns give the number of nests and number of basic blocks, respectively, for each application. The fifth column gives the number of integer ALU operations as a fraction of overall operations in dynamic execution. We see that integer ALU operations dominate the dynamic operation count and constitute as a good target for leakage optimization. The last column, on the other hand, shows the percentage of execution time spent in basic blocks whose DFGs have been used in determining the IPC for the associated loop. We see that 89.2% of the overall execution time, on the average, is spent on these basic blocks. Consequently, we can expect large leakage energy benefits if we are able to select suitable IPCs for these basic blocks.

To evaluate a large number of alternatives, we experimented with different VLIW configurations by varying the number of functional units. A common characteristic of these configurations is that they all have a single branch unit, as our experience showed that increasing the number of branch units did not make any difference in performance and energy behavior of these benchmarks.

6.4.1 Leakage Energy and Performance Impact of Adaptive IPC

We first give in Figure 6.4 the potential savings when our strategy is employed for both basic block (BB) and hyperblock (HB) cases (i.e., when the input code to our optimization module is scheduled using basic blocks and hyperblocks respectively). Ix_Ly

in this figure means that the VLIW architecture considered has a maximum of x integer ALUs and y load/store units. The potential savings here correspond to the percentage number of VLIW slots saved; that is, the ratio, (A-B)/(X) as in Figure 6.3.(b), where X is the number of slots in the original schedule (i.e., the schedule with the maximum number of integer ALUs) when one considers only integer ALUs. We see from Figure 6.4 that the percentage of slots saved varies between 0.6% (resp. 1%) and 83.5% (resp. 74.7%), depending on the specific configuration used when BB (resp. HB) is employed. The potential savings with the hyperblock scheduled code is lower as this scheduling technique utilizes the available VLIW slots more aggressively than basic block scheduling. Also, as expected, the larger the configuration, the higher the potential savings. In fact, the best savings occur with I8_L2. However, even with a more modest configuration such as I4_L1, we achieve 35.8% and 9.3% savings with the basic block and hyperblock cases, respectively.

To demonstrate the impact of our approach on execution cycles, let us consider Figure 6.5. Note that our approach can increase execution time in two ways. First, restricting IPC demands extra execution cycles to complete execution. Second, depending on the leakage control mechanism used, we may incur an extra execution time penalty whenever we want to reactivate an integer ALU placed into the leakage control mode. The graphs in Figure 6.5 show the percentage increase in execution cycles using the SG mechanism. When we consider the basic block scheduling case (the top graph), we see that the increase in execution cycles is less than 1% for seven out of ten benchmarks. This result reveals two facts. First, our IPC assignment strategy is very successful in selecting a suitable IPC for each nest without much impact in execution time. Second,

the performance penalty due to turn on activities is not very high as these activities may only occur at loop boundaries. The reason that we incur a large performance penalty in apsi is the difficulty in assigning IPC for the loops it contains. In this application, when we focus on the most frequently executed basic block (in selecting an IPC for the associated nest), we see that there is a large difference between different cycles as far as the number of parallel operations are concerned. This non-uniformity increases the overall schedule length, and since such basic blocks are frequently executed, we incur a significant performance penalty. When considering the hyperblock scheduling case (the bottom graph), we observe a very small performance penalty due to the fewer opportunities to place functional units into a leakage control mode. In nine of the applications, the performance penalty is less than 1%.

While the potential savings shown in Figure 6.4 are significant, it is also important to study the corresponding energy benefits using a specific leakage control mechanism. Hence, we performed another set of experiments adopting power supply gating (SG) as our leakage control mechanism. We see from the results illustrated in Figure 6.6 that we can achieve leakage energy savings of up to 72.4% when basic block scheduling is used. With hyperblock scheduling, on the other hand, the savings range from 1% to 67.1%.

6.4.2 IPC Comparison

The previous subsection showed that our adaptive IPC approach improves the energy consumption significantly over a version that uses the maximum IPC supported by the architecture. To see whether the IPCs selected using our strategy performs well as compared to other possible IPC selections, we performed another set of experiments

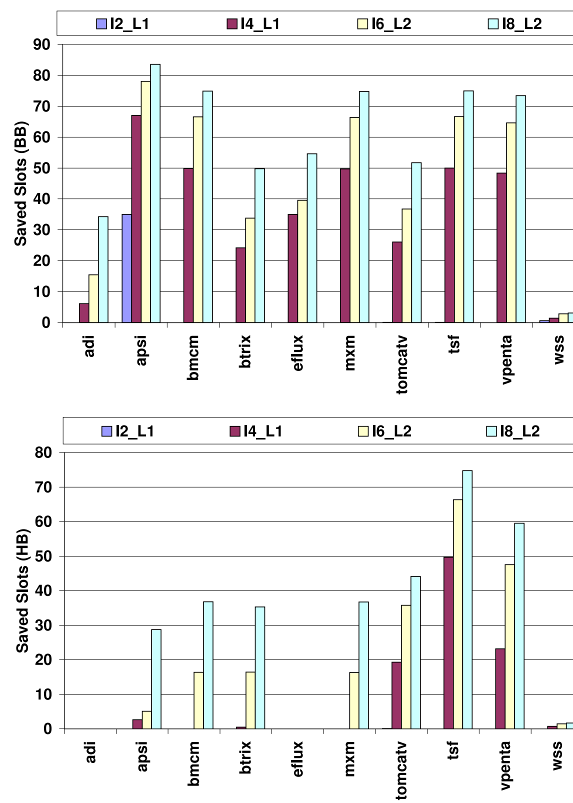


Fig. 6.4. % saved slots ((A-B)/X). This indicates potential for leakage energy reduction using leakage control mechanisms.

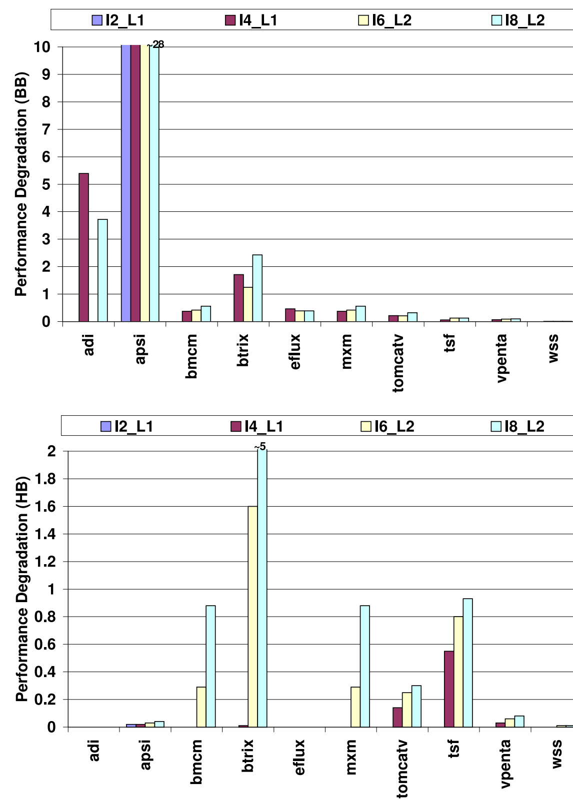


Fig. 6.5. % increase in execution cycles.

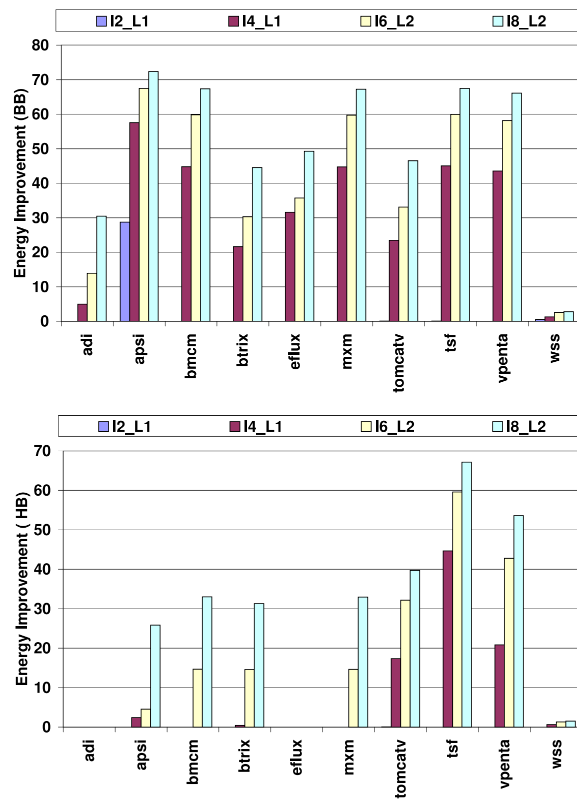


Fig. 6.6. % leakage energy improvements in IALUs.

where we run each application with different IPCs and obtained the energy-delay products (EDPs). The bar with the legend D in Figure 6.7 corresponds to the results from the IPC determined by our approach for the corresponding benchmark. The other bars represent the cases where the IPC used is the one determined by our approach plus the number associated with the bar. All results are for basic block scheduling only and are normalized with respect to the energy-delay product of the original code without any IPC control. We see that the IPC that we selected is the best one for most of the benchmarks. More specifically, the average (across all benchmarks) normalized EDPs for cases -1, D, 1, 2, 3, and 4 are 66.7%, 49.8%, 55.7%, 65.3%, 75.4%, and 83.6%, respectively. In general, a smaller IPC (denoted -1) reduces the energy consumption but increases schedule length. eflux and tomcatv experience dramatical increases in EDP as a result of this. On the other hand, a couple of benchmarks such as bmcm and mxm take advantage of this smaller IPC. This is because our IPC detection process considers all operations in the DAG but applies the selected IPC only for integer ALUs. This, in some cases, results in a larger number of IALUs being activated than necessary. We also see that increasing IPC beyond the value that we determined increases the EDP significantly. This is due to two factors. First, increasing IPC also increases the energy consumption. Second, not many applications take advantage of the increased IPC (mostly due to data and control dependences).

6.4.3 Granularity Analysis

Our loop-based IPC selection scheme determines a single IPC for each loop in the code. There are, however, other granularities that we could potentially work with. Two

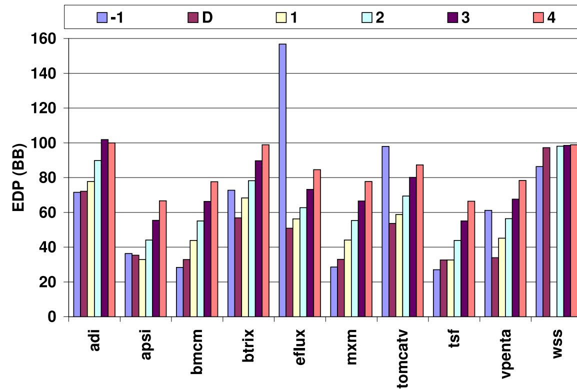


Fig. 6.7. Normalized EDP (energy-delay product) with different IPC settings. All values are normalized with respect to the EDP of the original schedule. The I8_L2 configuration was used for all cases.

extreme cases would be assigning a single IPC to the whole application and assigning an IPC per basic block. The advantage of the former strategy is that it does not incur any SDT penalty as there are no dynamic IPC modulations. The disadvantage is that trying to come up with a single IPC for the entire program might result in an IPC which might be suboptimal for some nests. On the other hand, working with basic block granularity customizes the IPC for each block but might increase the performance penalty (due to frequent IPC switches during execution).

Table 6.2 gives the leakage energy-delay products for three different cases considering all the functional units for IPC selection performed at different granularities: basic block level, loop level, and whole application level. At the block level, it is difficult to employ a more aggressive leakage control mechanism such as SG due to the larger overhead of reactivating the unit. Our results show that IVC provides the best choice at this granularity. For the loop level and whole program level adaptation, we employ

Granularity	LCM	I2_L1	I4_L1	I6_L2	I8_L2
block level	IVC	102.5	80.0	72.9	65.5
loop level	SG	99.1	70.1	59.0	49.9
whole program	SG	117.1	97.6	127.0	98.8

Table 6.2. Normalized EDP (energy-delay product) with different IPC selection schemes (averaged over all applications). All values are normalized with respect to the EDP of the original schedule. LCM means leakage control mode.

the SG leakage control mechanism. The average (over all configurations) normalized EDPs for block level, loop level, and whole application level cases are 80.2%, 69.5%, and 110.1%, respectively. We observe that the loop level granularity of control provides the best energy-delay results due to a balance between the level of adaptation and the overhead associated with the use of leakage control mechanisms. Further, we observe that the IPC setting policy of the whole program is quite stifling due to the coarse-grain adaptation.

6.4.4 Impact of Leakage Control Mechanism

In this subsection, we evaluate the effectiveness of the loop level IPC selection, employing different leakage control mechanisms that provide different leakage energy savings while incurring different SAT and SDT times. In addition to the IVC, SB and SG mechanisms described earlier, we add an additional (hypothetical) configuration (denoted NEW) that can reduce all leakage energy while incurring a SAT and SDT times of 75 cycles. These four schemes correspond to the four bars in Figure 6.8 for the I4_L1 configuration. We observe that the normalized leakage energy-delay product

(over all applications) is 78.5%, 79.2%, 70.1% and 65.75% for these four schemes. This shows that our technique is quite effective when using currently available leakage control mechanisms.

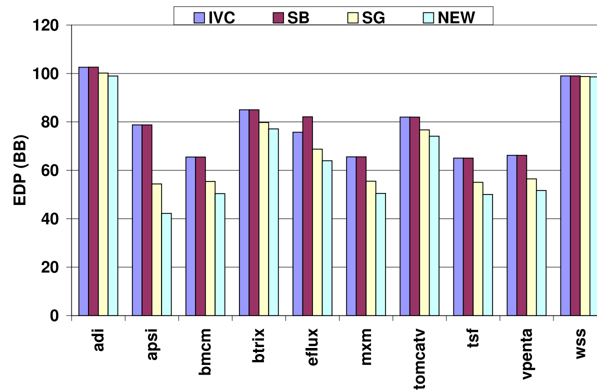


Fig. 6.8. Normalized EDP (energy-delay product) with different leakage control mechanisms. The I4_L1 configuration was used for all cases.

6.5 Conclusions and Ongoing Work

Energy consumption is a major design constraint that requires new optimizations at the various levels of the system. While many efforts have focused on optimizing dynamic energy at various levels, optimizing leakage energy is fast becoming important as threshold voltages continue to become smaller. In this work, we have exploited the inherent variations in the demand for functional units in an application and have adapted the number of active functional units to conserve leakage energy. More specifically, we presented a compiler-based scheme to identify the number of functional units to

activate for the different loops in the application and applied leakage control mechanism to the unused units to conserve energy. This scheme has been implemented for a VLIW architecture in the Trimaran framework along with performance penalties to apply the leakage control mechanisms, and evaluated using different applications. Our simulation results show that our technique can reduce up to 38% of the functional unit leakage energy averaged across a range of system configurations. Our results also show that our loop-based adaptation strategy gives better energy-delay product than finer-granular (basic block level) and coarser-granular (whole application level) adaptation schemes, which balances between the level of adaptation and the overhead associated with the application of leakage control mechanisms. We also show that our approach is effective when using currently available leakage control mechanisms. We believe that this work demonstrates the benefits from exploiting the synergy between application characteristics, machine architecture and the leakage control circuit primitives in conserving energy.

Chapter 7

CONCLUSIONS AND FUTURE WORK

With proliferation of portable embedded systems, power dissipation has become a key design metric. Systems that are powered by lightweight batteries increasingly demand more complex embedded functionalities. The lifetime of the batteries depends on average power consumption of the system; therefore, embedded systems designers are asked to pay close attention to the systems power consumption.

In this thesis, system energy consumption is evaluated using our proposed energy estimation models, and energy optimization techniques are proposed in architectural and software levels. The thesis focuses on two main components of the embedded systems: memory and datapath.

The memory related issues cover energy characteristics of different cache architectures, a software technique to optimize cache performance – loop-tiling–, and off-chip memory accesses. More specifically, first, three multiple access cache architectures are evaluated in terms of energy, using our proposed energy estimation models. The results show that the energy reductions obtained by using the multiple access caches can become more important as the cost of main memory accesses is reduced by the emerging eDRAM technology. Second, loop-tiling technique, which has been studied extensively to reduce memory latency, is evaluted in terms of power to show that the best performing tile size is different from the minimum energy consuming tile size. Third, an estimation

model to predict the number of page breaks of an application in SDRAMs is proposed using Polyhedral modeling technique. This estimation model can be used to estimate main memory energy consumption of various compiler optimization techniques. As an example, the blocked data layout scheme is evaluated to show that different block sizes can lead to very different main memory energy consumptions.

The datapath related issues deal with VLIW architectures, since they are becoming popular in DSP arena from their reduced hardware complexity and support for wider instruction level parallelism. First, an architectural energy estimation tool is designed, based on a publicly available VLIW compilation toolset, Trimaran. This enables us to evaluate a variety of VLIW compilation techniques on a set of different architectural parameters such as the number of functional units, register file sizes, and operation latencies, in terms of both power and performance. As technology feature size decreases, the leakage energy consumption increases dramatically. Particularly, leakage energy reduction of functional units is crucial due to the temperature surge caused by heavy usage of the modules. Therefore, second, we present a compiler-based scheme to identify the number of functional units to activate for different loops in the application and apply leakage control mechanisms to the unused units to conserve energy. This scheme has been implemented for a VLIW architecture using the Trimaran framework and evaluated using different applications. Our simulation results show that our technique can reduce up to 38% of the functional unit leakage energy averaged across a range of system configurations.

Future work includes:

- SDRAM energy and performance estimation framework

  Foremost, the estimation framework needs to be automated. It should have automatic data layout transformation step and automatic tile selection mechanism enabled by incorporation of the Omega library and Polylib. Currently, the framework supports only two dimensional arrays. It has to be extended to incorporate higher order dimensions. In [23], it was shown that the nonlinear (blocked) data layout improves the performance of loop tiling largely due to cache conflict miss reduction. It also showed good cache performance regardless of array sizes. However, their work did not consider off-chip performance. Our goal is to propose a loop tile selection algorithm which incorporates SDRAM's performance and energy using the proposed estimation framework.

- Trimaran energy simulator

  As the ILP increases in VLIW architectures, the register file size increases to sustain the increased ILP. To tackle speed and power problem caused by the the increased register file size and number of ports, one register file is split into multiple clusters, where each cluster has one local register file with multiple functional units attached to it. Currently, Trimaran framework does not support clusters. However, it provides machine description language support. Our future work is to model clustered architectures with the machine description language and evaluate its energy consumption. Register file is also a main source of leakage in VLIW architectures. We plan to work toward this direction to evaluate leakage energy of the register files, and to selectively turn off register file banks whenever not used.

The compiler's register assignment step needs to be studied to prolong and cluster the turn off period.

References

[1] The Micron System-Power Calculator. In http://www.micron.com/products/category.jsp?path=/DRAM/SDRAM&edID=17594.

[2] The Omega Project. In http://www.cs.umd.edu/projects/omega/.

[3] Rambus Inc. In http://www.rambus.com.

[4] System support for energy management in mobile and embedded workloads: A white paper. In http://www.cs.duke.edu/ carla/research/whitepaper.pdf.

[5] Texas Instruments device information. In http://dspvillage.ti.com/docs/dspproducthome.jhtml.

[6] Trimaran. In http://www.trimaran.org.

[7] Wisconsin architectural research tool-set. In http://www.cs.wisc.edu/ larus/warts.html.

[8] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct mapped caches. In *20th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 179–190, 1993.

[9] G. Albera and R.I. Bahar. Power and performance trade-offs using various cache strategies. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 64–69, 1998.

[10] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 166 – 178, 1995.

[11] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 227–237, 1998.

[12] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 218–229, 2001.

[13] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 16–21, August 2001.

[14] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, June 2000.

[15] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimization. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[16] J. Bunda, W. C. Athas, and D. Fussell. Evaluating power implication of CMOS microprocessor design decisions. In *Proceedings of International Workshop on Low Power Design*, 1994.

[17] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report CS-TR-96-103, Computer Science Dept., University of Wisconsin, Madison, 1996.

[18] J. A. Butts and G. Sohi. A static power model for architects. In *Proceedings of International Symposium on Microarchitecture*, pages 191–201, December 2000.

[19] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA)*, pages 244–253, 1996.

[20] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In *Proceedings of IEEE Workshop on VLSI Signal Processing (SIPS)*, pages 178–187, 1994.

[21] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology – exploration of memory organization for embedded multimedia system design*. Kluwer Academic Publishers, 1998.

[22] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[23] S. Chartterjee, V. V. Jain, A. R. Lebeck, and S. Mundhra. Nonlinear array layouts for hierarchical memory systems. *Proceedings of ACM International Conference on Supercomputing*, pages 444–453, 1999.

[24] S. Chartterjee, A. R. Lebeck, and P. K. Patnala. Recursive array layouts and fast matrix multiplication. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[25] R. Y. Chen, R. M. Owens, , and M. J. Irwin. Validation of an architectural level power analysis technique. In *Proceedings of the 35th Design Automation Conference (DAC)*, pages 242–245, June 1998.

[26] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI TR-93-12, Sun Microsystems Inc, 1993.

[27] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, 1995.

[28] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of The 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 316–325, 2000.

[29] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Proceedings of 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 62–71, 2001.

[30] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of The 26th Annual International Symposium on Computer Architecture (ISCA)*, pages 222–233, 1999.

[31] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Memory energy management using software and hardware directed power mode control. Technical Report CSE-00-004, Department of Computer Science and Engineering, Penn State University, April 2000.

[32] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of 35th International Symposium on Microarchitecture*, pages 321–332, November 2002.

[33] D. Duarte, M. J. Irwin, and V. Narayanan. Modeling energy of the clock generation and distribution circuitry. In *13th Annual IEEE International Conference ASIC/SOC*, pages 261–265, 2000.

[34] G. Esakkimuthu, H. S. Kim, M. Kandemir, N. Vijaykrishnam, and M. J. Irwin. Investigating memory system energy behavior using software and hardware optimizations. *Special issue in Low power system design of VLSI DESIGN*, 12(2):151–165, February 2001.

[35] K. Esseghir. Improving data locality for caches. In *Master's Thesis, Dept. of Computer Science, Rice University*, September 1993.

[36] B. Burgress et al. The powerPCTM603 microprocessor: a high-performance, low-power, super-scalar RISC processor. In *IEEE COMPCON*, February 1994.

[37] S. B. Furber et. al. AMULET1: A micro-pipelined ARM. In *Proceedings of IEEE COMPCON*, February 1994.

[38] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–193, 1994.

[39] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor, and D. Verkest. Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm. In *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS)*, pages 603–612, 2000.

[40] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Proceedings of Workshop on Complexity-Effective Design*, June 2000.

[41] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In *Proceedings of 1999 International Symposium Low Power Electronics and Design (ISLPED)*, pages 70–75, 1999.

[42] J. P. Halter and F. N. Najm. A gate-level leakage power reduction method for ultra-low-power CMOS circuits. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 475–478, 1997.

[43] W. W. Hwu. ECE 412 class notes. In *Dept. of ECE, UIUC,* `http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/index.html`.

[44] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, pages 229–248, May 1993.

[45] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 273 – 275, 1999.

[46] F. Irigoin and R. Triolet. Super-node partitioning. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[47] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the IEEE Design Automation and Test in Europe (DATE)*, pages 190–196, March 2001.

[48] M. B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 143–148, August 1997.

[49] M. B. Kamble and K. Ghose. Energy-efficiency of VLSI caches: a comparative study. In *Proceedings of International Conference on VLSI Design*, pages 261–267, 1997.

[50] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of Microarchitecture (MICRO-31)*, pages 285–297, December 1998.

[51] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim. Experimental evaluation of energy behavior of iteration space tiling. In *Proceedings of the Workshop on Languages and Compilers for high Performance Computing (LCPC)*, pages 142–157, October 2000.

[52] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim. Towards energy-aware iteration space tiling. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 211–215, June 2000.

[53] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Design Automation Conference (DAC)*, pages 304–307, June 2000.

[54] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 240–251, June 2001.

[55] R. R. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of self-associativity. In *16th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 131–139, 1989.

[56] A. Khare, P. R. Panda, N. Dutt, and A. Nicolau. High-level synthesis with synchronous and RAMBUS DRAMs. Technical Report 98-28, University of California, Irvine, 1998.

[57] H. Kim and I.-C. Park. High performance and low-power memory-interface architecture for video processing applications. *IEEE Transaction on Circuits and Systems for Video Technology*, 11(11):1160–1170, November 2001.

[58] H. S. Kim, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Effect of compiler optimizations on memory energy. In *Proceedings of Workshop on Signal Processing System (SIPS)*, pages 663 – 672, October 2000.

[59] H. S. Kim, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. *Characterization of memory energy behavior in Book Chapter for Characterization of Contemporary Workloads*. Kluwer Academic Publishers, 2001.

[60] H. S. Kim, N. Vijaykrishnan, M. Kandemir, E. Brockmeyer, F. Catthoor, and M. J. Irwin. Estimating influence of data layout optimizations on SDRAM energy consumption. In *Proceedings of International Symposium on Low Power Electronics and Design (to appear)*, 2003.

[61] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Multiple Access Caches: Energy implications. In *Proceedings of IEEE CS Annual Workshop on VLSI*, pages 53 – 58, April 2000.

[62] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. A framework for energy estimation of VLIW architecture. In *Proceedings of International Conference on Computer Design (ICCD)*, pages 40 – 45, 2001.

[63] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting instruction level parallelism for optimizing leakage in VLIW architectures. In *Proceedings of International Symposium on Languages, Compilers, and Tools for Embedded Systems (to appear)*, 2003.

[64] J. Kin, G. M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184 – 193, 1997.

[65] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 346–357, June 1997.

[66] G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic.* North-Holland Pub. Co., 1967.

[67] M. S. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 63–74, April 1991.

[68] C. Lee. UTDSP benchmark suite. In `http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html`.

[69] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330 – 335, 1997.

[70] W. Li. Ph. D. Thesis, compiling for NUMA parallel machines. Technical report, Computer Science Dept., Cornell University, Ithaca, NY, 1993.

[71] V. Loechner. PolyLib: A library of polyhedral functions. In `http://icps.u-strasbg.fr/PolyLib/`, 1999.

[72] Y.-H. Lu, L. Benini, and G. De Micheli. Operating-system directed power reduction. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 37–42, 2000.

[73] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.

[74] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Pub., San Francisco, California, 1997.

[75] Y. Nunomura, T. Shimizu, and O. Tomisawa. M32R/D-integrating DRAM and microprocessor. *IEEE Micro*, 17(6):40–48, November/December 1997.

[76] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Pub., 1999.

[77] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of International Symposium on System Synthesis*, pages 90–97, September 1997.

[78] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 90–101, 2001.

[79] J. Ramanujam and P. Sadayappan. Tiling multi-dimensional iteration spaces for multi-computers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[80] B. R. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4(2/3):71–118, 1999.

[81] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimization of static power dissipation by functional units in superscalar processors. In *Proceedings of International Conference on Compiler Construction*, pages 261–275, April 2002.

[82] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, June 1998.

[83] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, pages 168–182, March 1999.

[84] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *Proceedings of the International symposium on High performace Computer Architecture (HPCA)*, pages 375–386, 2000.

[85] K. Roy and M. C. Johnson. *Software design for low power.* Kluwer Academic Press, October 1996, ed. J. Mermet and W. Nebel.

[86] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Power exploration for embedded VLIW architectures. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 498–503, 2000.

[87] V. Sarkar, G. R. Gao, and S. Han. Locality analysis for distributed shared-memory multiprocessors. In *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing (LPCP)*, pages 20–40, August 1996.

[88] V. Seferidis and M. Ghanbari. Hierarchical motion estimation using texture analysis. In *Proceedings of International Conference on Image Processing and its Applications*, pages 61–64, 1992.

[89] W.-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. Technical Report CLPE-TR-9-1999-20, Arizona State University, 1999.

[90] W.-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In *Proceedings of the 36th Design Automation Conference*, pages 140–145, 1999.

[91] P. Song. Embedded DRAM finds growing niche. *Microprocessor Report*, 4:19–23, August 1997.

[92] C. Su and A. Despain. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 63 – 68, 1995.

[93] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of the IEEE Supercomputing*, pages 410–419, November 1993.

[94] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems*, 13(2), August 1996.

[95] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-Fetch: Compiler-enabled power-aware fetch throttling. *ACM Computer Architecture Letters*, 1, 2002.

[96] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 95–106, 2000.

[97] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, W. Ye, and D. Duarte. Evaluating integrated hardware-software optimizations using a unified energy estimation framework. *IEEE Transactions on Computers*, 52(1):59 – 76, January 2003.

[98] S. E. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, DEC WRL Research report, 1994.

[99] M. Wolf. *High Performance Compilers for Parallel Computing*. Addison Wesley, CA, 1996.

[100] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN 91 Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, June 1991.

[101] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of International Symposium on Microarchitecture*, pages 274–286, December 1996.

[102] J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *International Journal of Parallel Programming*, 26(6):671–696, December 1998.

[103] S. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches. In *Proceedings of ACM/IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 147–157, January 2001.

[104] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Design Automation Conference (DAC)*, pages 340–345, June 2000.

[105] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, pages 40–49, September/October 1997.

[106] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, , and V. Delaluz. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe (DATE)*, pages 1146–1147, March 2003.

[107] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive mode control: A static-power-efficient cache design. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 61–70, Sept. 2001.

[108] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, pages 305–310, 1998.

Vita

Hyun Suk Kim was born in Taegu, Korea on September 8, 1971. In 1994, she received her B.S. degree in Computer Science from Kyungpook National University, Taegu, Korea. In 1996, she received the M.S. degree in Computer Science from Pohang University of Science and Technology, Pohang, Korea. From 1996 to 1998, she worked as a software engineer for Samsung Electronics in Seoul, Korea. In 1998, she enrolled in the Ph. D. program in Computer Science and Engineering at the Pennsylvania State University. Since summer 1999, she has worked as a member of the Microsystems Design Laboratory. In Spring 2002, she was a visiting researcher at the DESICS group in Interuniversity Micro Electronics Center, Belgium. She received the Robert-Owens Memorial Fellowship for outstanding graduate student in 2001, and served as an instructor in Spring 2003 in CSE Department at Penn State.