Enhancing Compiler Techniques for Memory Energy Optimizations

By

Joseph Anthony Zambreno

A thesis submitted in partial fulfillment of

the requirements for the degree of

MASTER OF SCIENCE

Northwestern University
Evanston, Illinois

June 2002

Acknowledgement

I would first like to express thanks to my advisor, Professor Alok Choudhary, not only for providing the direction to this research, but for assisting me in my transition from undergraduate to graduate study. Also, much credit is due to Professor Mahmut Kandemir from Penn State, whose clarity of thinking has been especially helpful in analyzing my results and in supplying sincere feedback. I would also like to thank Professors Prith Banerjee and Renato Figuriedo for serving on my Final Examination committee. Lastly, I would be remiss if I didn't mention my fellow researchers in the Center for Parallel and Distributed Computing who have assisted me on numerous occasions, including but not limited to: Jay, Nikos, Keenan, Avery, Steve, Alex, and Greg. To each of the above, I extend my deepest appreciation.

Dedication

This thesis is dedicated to my parents, who have always trusted me to make my own mistakes, and to my girlfriend Mary, whose love and support make me believe that anything is possible. Not in a million years could I have done this without you.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

- Don Knuth

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

- Maurice Wilkes

Abstract

As both chip densities and clock frequencies steadily rise in modern microprocessors, energy consumption is quickly joining performance as a key design constraint. Power issues are increasingly important in embedded systems, especially those found in portable devices. Much research has focused on the memory subsystems of these devices since they are a leading energy consumer. Compiler optimizations that are traditionally used to increase performance have shown much promise in also reducing cache energy consumption. However, many of these optimizations result in a tradeoff between a lower instruction count and a larger executable size, and it is not clear as to what extent these optimizations should be applied in order to meet power and performance constraints.

In this work, we present energy consumption models that we have modified for a memory hierarchy such as would be found in typical embedded systems. Using these models, we single out some optimizations that a power-aware compiler should focus on by examining the energy-reducing effect they have on various media and array-dominated benchmarks. We present a simple yet accurate metric that such a power-aware compiler can use in order to estimate the effect of potential optimizations on energy consumption. Next, we present heuristic algorithms that determine a suitable optimization strategy given a memory energy upper bound. Finally, we demonstrate that our strategies will gain even more importance in the future where leakage energy is expected to play an even larger role in the total energy consumption equation.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Problem Statement

Until very recently, performance has been the main focus of modern microprocessor design. Indeed, major improvements in VLSI technology have allowed for denser transistors and higher clock frequencies than previously thought possible. The traditional path to performance has followed Moore's law, with the unintended side effect that power dissipation is also increasing exponentially (Figure 1.1). The total power consumption of the processors of the world's supply of personal computers has increased from 160 MW in 1992 to an estimated 9000 MW in 2001 [36]. Besides the obvious environmental burden of this burgeoning power demand, there is also an increased cost to design systems that can handle higher power consumption.

While power issues are certainly important during the design of personal computers and high-end servers, it is the embedded market where methods of reducing power dissipation are most necessary. Embedded processors, such as those that are found in PDAs, laptops, and cell phones, typically rely on a portable power supply. As these systems have become more complex

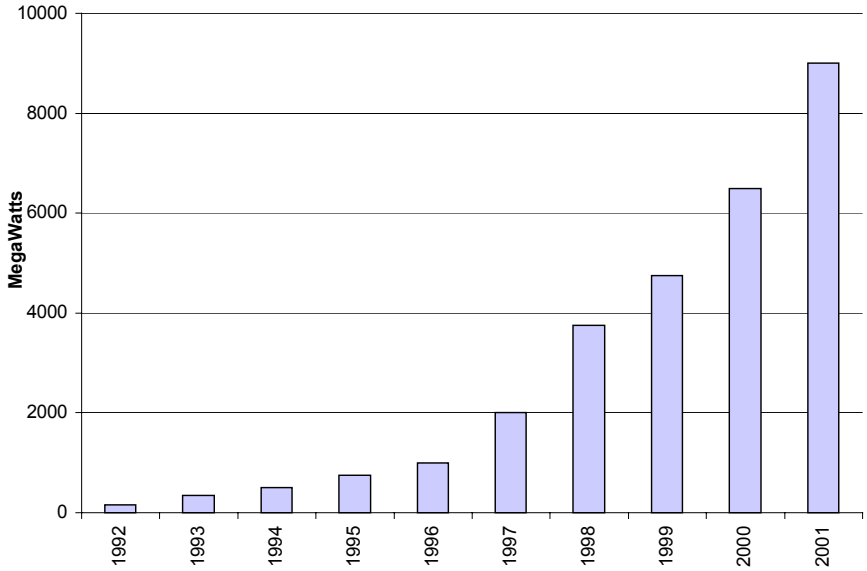

Figure 1.1. Exponential processor power increase over time (total world PCs output) [36]

communication products, their battery life is becoming as important of a price metric as their performance.

Much research has been done on improving the power dissipation of memory subsystems, since it has been shown that they are a significant contributor of power dissipation in embedded systems [27] and high-performance processors [12]. Many embedded systems, especially those found in SoC (System-on-Chip) designs, have customized memory hardware tailored to the needs of a small set of applications. The amount of data and instruction memory included tend to be no larger than necessary, since the per access energy consumption increases with memory size [6]. Since the size of the instruction memory is customized based on the application at hand, software techniques are able to directly affect the amount of power dissipated by these devices. For these reasons, we have focused this thesis on analyzing the effects of various compiler optimizations on a fixed memory architecture such as would be found in embedded systems.

Standard compiler optimizations, such as loop unrolling and tiling, are used to increase the performance of memory systems by increasing spatial and temporal data locality [29,42]. A

common side effect of these optimizations is an increased code size. Therefore, we cannot apply all these optimizations aggressively; the increase in code size needs to be accounted for. In this thesis, we concentrate on determining which optimizations have favorable code size and performance tradeoffs to determine the extent to which they should be applied.

In order to fully understand research in the field of low-power computing, it is crucial to have a general sense of how power and energy relate to each other, and also how a digital device dissipates power. Equation 1.1 shows the basic relationship between power and energy:

$$P = dE/dt \qquad \text{(Eqn. 1.1)}$$

The power consumption of a process is equivalent to the rate of change of its energy dissipation. For embedded devices such as cell phones and PDAs, the battery life is determined by this equation. The relationship is linear; a new technique that reduces the power consumption in half will accordingly double the battery life assuming that the technique doesn't affect the time required to execute any given task. Each time a binary signal on a metal wire switches from a 0 to a 1 or vice versa, there is power consumption proportional to the capacitance of that wire and the voltage across it. In a more generalized sense for this thesis, we use the following definition for dynamic power consumption, also known as switching power consumption:

$$P_d = C_{eff} \cdot V_{dd}^{\,2} \cdot f_{clk} \qquad \text{(Eqn. 1.2)}$$

where C_{eff} is the effective switching capacitance of the circuit [33], V_{dd} is the operating voltage, and f_{clk} is the clock frequency. Many of the techniques to reduce power consumption in modern microprocessors can be traced back to the manipulation of this equation (Figure 1.2). For example, a computer architect may attempt to:

1) reduce C_{eff} by eliminating unnecessary hardware or by limiting the switching frequency of the different components.

2) reduce V_{dd} by operating the circuit at a lower voltage.

3) reduce f_{clk} by gating the clock input to different components, effectively shutting them off.

Much low-power architectural research has concentrated on option 1. As will be seen in Chapter 2, the effective switching capacitance is a determined by low-level transistor technology, architectural arrangement, and frequency of use. Because of this last contributor, software techniques are able to directly alter the effective switching capacitance and consequently the work for this thesis falls under option 1. Option 2 has been used successfully in the past, but its effectiveness is reaching limits that will be discussed in Chapter 2. Option 3 can be effective in reducing dynamic power consumption,

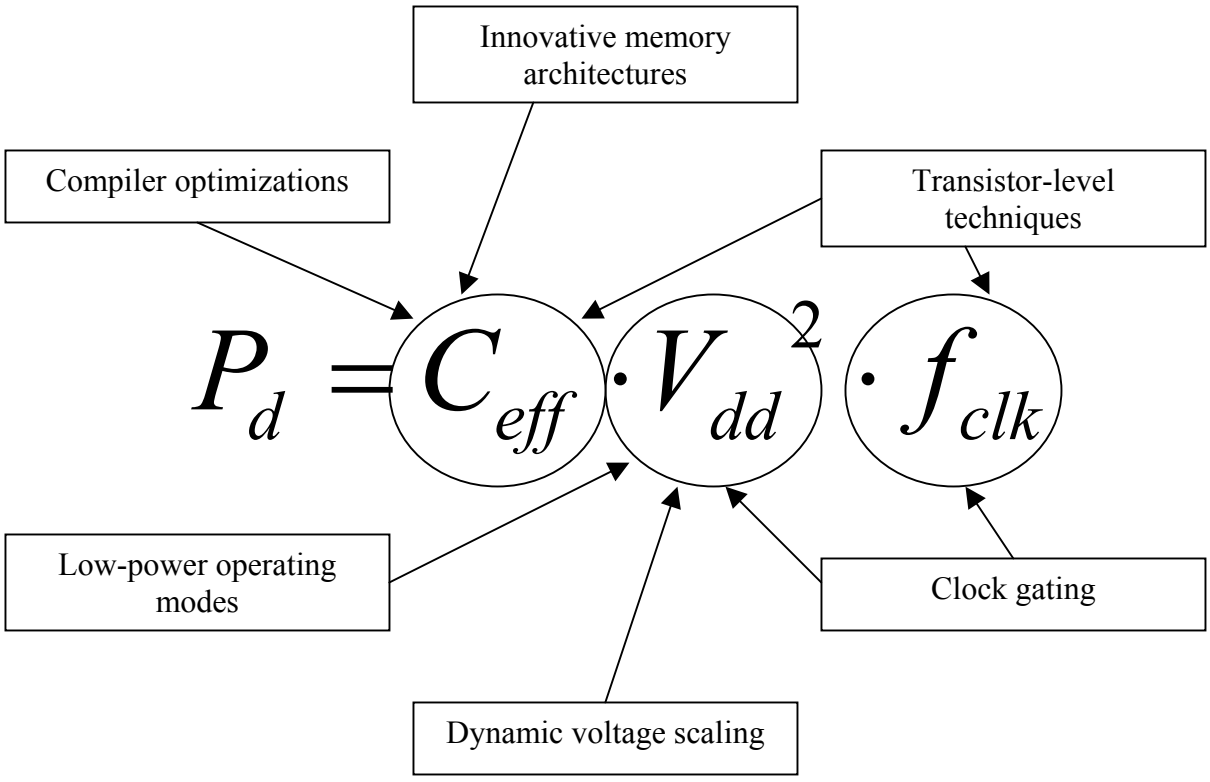

Figure 1.2. Common methods to reduce memory power consumption

but usually involves adding extra gate delays into clock lines, increasing clock skew and complicating the design analysis. For these reasons option 3 is often limited to cases where reducing power is crucial [23]. Note that as an introduction here we only discuss dynamic power; in Chapter 2 we will go into much further detail about modeling both dynamic and static (leakage) power consumption.

1.2 Related Work

1.2.1 Power Reduction

We discuss the related research in the field of low-power computing as it fits into three categories: the circuit-level, the architectural-level, and the software-level.

At the *circuit-level*, there have been numerous optimizations proposed for minimizing energy consumption. Yang et. al. [43] present a gated supply voltage design that interacts with a dynamically resizable instruction cache. By turning off the supply voltage to unused sections of the cache, their method effectively eliminates the leakage power consumption in those sections. Ye et. al. [44] developed a method of transistor stacking in order to reduce leakage energy dissipation while maintaining high performance. Chandrakasan and Brodersen [7] have written the definitive reference on low-power circuit design techniques.

At the *architectural-level*, much work that has been done to improve memory and CPU performance with the added expectation that power consumption will also improve. Also in this area several techniques have been proposed to reduce switching and leakage energy consumption at the cost of small performance losses. Hajj et. al. [14] present instruction cache energy reduction by

using an intermediate cache between the instruction cache and the main memory. Their research shows that this smaller intermediate cache allows the main instruction cache to remain disabled most of the time. Delaluz, Kandemir et. al. [11] discuss using low-power operating modes for DRAMs to conserve energy dissipation by effectively shutting off the DRAM when not in use. They present compilation techniques to analyze and exploit memory idleness and also a method by which the memory system can use self-detection to switch to a lower-power operating mode. In [2], Balasubramonian et. al. suggest a cache and TLB layout that significantly decreases energy consumption while increasing performance. Their suggested layout allows for dynamic memory configuration that analyzes size and speed tradeoffs on a per-application basis. Kaxiras, Hu and Martonosi [18] present a method to reduce cache leakage energy consumption by turning off cache lines that likely will not be used again. By realizing that most cache lines typically have a flurry of frequent use when first introduced and then a period of "dead time" before they are evicted, Kaxiras et. al. were able to reduce L1 cache leakage energy by 5x for certain benchmarks with only a negligible performance decrease. In [32], Powell et. al. suggested a method of gating the voltage to unused SRAM cells to reduce leakage dissipation. By integrating circuit and architecture approaches, they were able to demonstrate a leakage energy reduction of 60% accompanied by a minimal negative impact on performance. We feel that the approach discussed in this paper is complementary to these architecture and circuit-based techniques, and that the best energy results can be obtained by integrating software and hardware techniques.

At the *software-level*, many preliminary investigations have been conducted into compiler techniques, more specifically to analyze how optimizations developed to increase performance can also improve power consumption. In [5], Catthoor et. al. offer a methodology for analyzing the effect of compiler optimizations on memory power consumption. Mehta et. al. [24] investigate the

effect of loop unrolling, software pipelining and recursion elimination on CPU energy consumption. They also present an algorithm for register relabeling that attempts to minimize the energy consumption of the register file decoder and instruction register by reducing the amount of switching in those structures. An introductory look into other high-level optimizations such as loop fusion, loop distribution and scalar replacement is performed with SimplePower in [39]. Hajj et. al. examine function inlining in [14], but only in the context of its effectiveness with custom cache architectural modifications. Ramanujam et. al. [34] present an algorithm to estimate the actual memory requirement for data transfers in embedded systems. They also present loop transformations that attempt to minimize the amount of memory required. Muchnick [29], Morgan [28], and Leupers [21] propose techniques for limiting the aggressiveness of function inlining. Our work is different from theirs in a number of ways: first, we focus on energy consumption; second, we present a metric that captures the energy behavior of the application being optimized; and third, in addition to inlining, we also study other classical performance-oriented techniques.

1.2.2 Power Estimation

It is not a trivial problem to measure the power consumption of modern microprocessors. Each of the techniques listed above also required a predetermined method of estimating their effectiveness. Much work has been done in the area of power estimation, and in this section we review some of the more popular methods.

Most research in the field of low-power hardware and software techniques leverage cycle-level simulators. Much work has been done on extending the popular SimpleScalar simulator [4] to include power-estimation models (Figure 1.3). The Wattch simulator [3] and the SimplePower

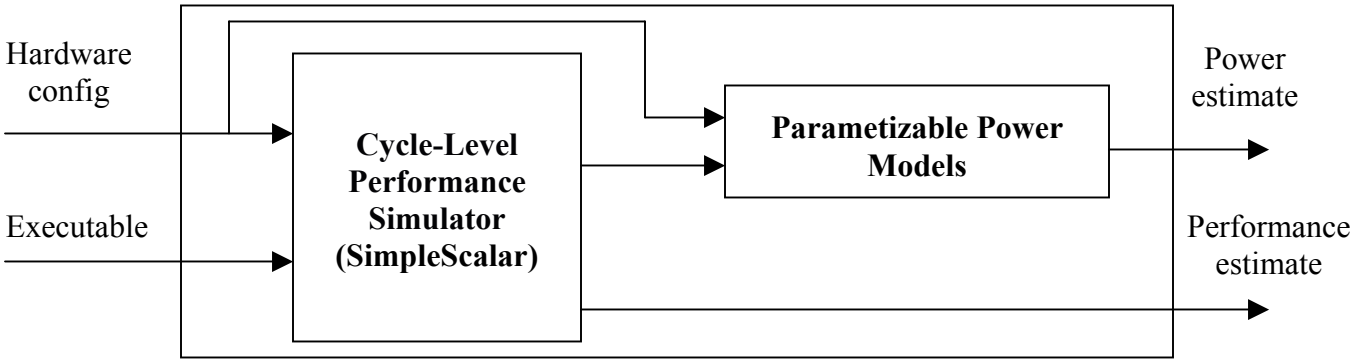

Figure 1.3. Structure of a power simulator based on SimpleScalar

simulator [39] leverage the SimpleScalar framework to model power consumption in a standard 5-stage pipelined RISC datapath. The SimplePower simulator uses a table-lookup system based on power models for memory and functional units, while Wattch relies on more detailed parameterized models. These simulators provide a solid framework for investigating low-power architecture, but are less useful for analyzing software techniques.

Before these high-level simulators can be built, detailed power models must be developed that can accurately estimate power consumption at the architectural-level. To this end, much work has been done on estimating dynamic power consumption. Kamble and Ghose [17] provide a good derivation of dynamic power models for low-power caches, and verify them against a low-level simulator. The models in [17] are used to investigate architectural-level cache changes. In contrast, in this work, we exclusively focus on studying the impact of code optimizations on energy and performance. As supply voltages have been lowered over recent years to decrease dynamic power consumption, the analysis of leakage power consumption has become increasingly important. Butts and Sohi [5] present a detailed derivation of a model for leakage power estimation for CMOS circuits. Chen et. al. [9] also attack this problem and supply an algorithm for estimating leakage

power in circuits that are in low-power standby mode. The power models used for the purposes of this thesis are derived from [17] and will be discussed in further detail in Chapter 2.

Joseph and Martonosi [16] present a different approach to estimating power consumption. In their Castle tool, they feed data from run-time hardware performance counters into power models to estimate the overall consumption in the main CPU components. The Castle tool has been shown to be extremely accurate (when verified against other power estimation methods), but suffers from the fact that different CPUs have completely different hardware event counters, making portability almost impossible. For our work, we use a variation of the Castle concept. For our power estimation runs, we take hardware counters from one machine that are deemed general enough, and adapt the power models for our target architecture.

In summary, there are several inherent limitations to the popular methods of power-estimation. In theory, the perfect method of measuring power consumption would involve setting up an ammeter on the CPUs incoming power supply lines. Although theoretically feasible, this method is never used since it is impossible to determine the power distribution in each CPU component, which is often necessary in such investigations. The simulator method attempts to bridge this gap by using power models to estimate the consumption in each component. However, many of these models don't accurately depict the real power contributions, and these simulators force a reasonable distribution. Likewise, the models for dynamic and leakage power consumption are usually so dependent on low-level circuit parameters that only a relative comparison is of any value. In this work we embrace these limitations and mainly concentrate on relative power consumption.

1.3 Contributions

The contributions of this thesis can be summarized as follows.

- We develop detailed energy consumption models for an instruction and data memory hierarchy. We present models for both dynamic (switching) and static (leakage) energy consumption.
- We adapt these models to a memory hierarchy such as would be found in System-on-Chip (SoC) embedded devices. We present three common memory configurations that allow us to measure the relative effectiveness of our techniques.
- We discuss practical concerns that arise regarding the limitation of our approach and how it affects the absolute and relative accuracy of our models.
- We present a description of the appropriate hardware performance counters required by our energy consumption models. We present a practical example architecture and toolset that can be used to gather the appropriate run-time data, the MIPS R10000 and the *perfex* tool.
- We present a simple yet accurate metric for estimating instruction memory power consumption that a power-aware compiler could leverage when making optimization decisions.
- We analyze the relative effectiveness of our metric by comparing with our original run-time power model results gathered from executing several media and array-dominated benchmarks.

- We present an overview of standard loop-nest and interprocedural optimizations and investigate their effect on code size and performance.
- We perform optimization experiments by leveraging the SGI MIPSPro compiler. In these experiments we examine both isolated optimizations and optimization suites.
- We perform experiments tailoring the aggressiveness of function inlining and show how a heuristic can be applied to improve performance while keeping overall instruction memory energy consumption beneath a predetermined upper bound.
- We perform similar experiments tailoring the aggressiveness of data-centric optimizations such as loop unrolling and loop tiling and apply a similar heuristic to a data memory hierarchy.
- We demonstrate that our heuristics will be of even greater importance under futuristic scenarios where leakage energy will comprise a larger percentage of the overall energy budget.
- We present recommendations as to what optimizations are effective for reducing power consumption, and examine the extent to which these optimizations should be applied. Using this information we finally then make general recommendations as to how a power-aware compiler should leverage performance optimizations when targeting embedded systems.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we derive detailed analytical power dissipation models. We examine both dynamic and static power models, and adapt them both

to instruction and data memories such as would be found in embedded devices. In this chapter we also discuss how our general investigative approach utilizes these models. Chapter 3 presents our experimental methodology, detailing the advantages and limitations of the SGI hardware and software that we utilized. In Chapter 4 we discuss loop restructuring optimizations, and how in attempting to improve data memory performance they affect data and instruction memory energy. It is in this chapter that we present results detailing the effect of these optimizations on performance and resulting code size, and present a heuristic that allows us to improve performance while keeping the total energy under a predetermined upper-bound. Chapter 5 follows a similar pattern while looking at interprocedural optimizations and their effect on code size, performance, and energy consumption. In Chapter 6 conclusions are presented. It is in this chapter that we provide recommendations as to the role an optimizing compiler can play in reducing memory energy consumption. Finally, in this chapter we outline future work that is planned in this area, noting the potential application of our research to the PACT project at Northwestern University.

CHAPTER 2

Modeling Energy Consumption

2.1 Motivation

In the quest to accurately model run-time cache energy consumption, one has a wide range of possible directions to follow. At the low end of the spectrum are detailed circuit-level equations that have the potential for extremely high accuracy but are difficult to work with. At the other end are the cycle-level performance simulators fitted to output power results. While relatively quick and easy to use, many of these simulators produce power estimates of questionable accuracy. For our work, we adapt an intermediate approach that captures most of the accuracy of the circuit-level models while still maintaining a high level of usability.

In this chapter, we derive analytical cache power dissipation models that are highly customizable to different memory architectures. We model both dynamic and static power dissipation, and provide adaptations for both instruction and data memory hierarchies. In doing so we provide a framework to analyze software-level optimizations without requiring high-level simulators or low-level circuit equations. Our models require only simple run-time performance

information such as cache hit and miss ratios in order to perform energy estimations. Since this data is easily available from common hardware performance counters, our models allow for an optimizing compiler to analyze power and performance tradeoffs.

2.1.1 General Investigative Methodology

It is important at this time to discuss in general terms our experimental methodology in order to give a sense of validity to our approach. A more thorough treatment of this topic can be found in Chapter 3. For our experiments we chose to utilize a SGI Origin machine [35], which is a shared memory multiprocessor computer. We chose to use this machine not for its multiprocessing capabilities, but for its extremely advanced optimizing compiler, the MIPSPro compiler. The Origin machine contains 8 MIPS R10000 processors, each containing separate instruction and data caches. Each processor contains a logically separated memory hierarchy, a fact that allows us to target code for a single processor while leveraging the power of the MIPSPro compiler.

To run an experiment, we compile a given benchmark using the MIPSPro C compiler, and then run it with the hardware counter profiling tool, *perfex*. The MIPS R10000 processor contains several hardware counters that measure statistics such as cache hit and miss ratios, and instructions executed. As we will explain in more depth in Chapter 3, we are able to use the SGI machine to calculate energy consumption estimates for memory hierarchies that would be found in embedded systems by plugging values for these hardware counters into our energy equations that are derived below. Despite the fixed nature of the memory hierarchy of the MIPS R10000 processor, we are able to estimate energy values for several theoretical memory hierarchies by carefully choosing the hardware counters that correspond to the desired cache statistics.

2.1.2 Notes on Accuracy

The accuracy of our approach as detailed above is potentially limited by two main factors. Firstly, we are limited by the ability of the MIPS R10000 processor to emulate the behavior of an embedded processor. The R10000 is a speculative processor, meaning that it attempts to execute instructions that it might need in the future in an attempt to improve performance. Since speculative execution can have an undeterminable effect on cache hit-miss ratios and embedded processors rarely feature this ability, it is a factor that limits the accuracy of our methodology. However, the R10000 contains hardware counters that give an idea as to how much speculation existed during a program run. As will be shown in Chapter 3, these counters can be used to present a fairly accurate estimation as to how a program would dissipate energy on a similar non-speculative processor. Also as noted above, the R10000 contains a fixed memory architecture, making it difficult to emulate the effect of an optimization on different embedded systems. This problem is alleviated by the fact that the first-level cache structure contained in the R10000 is not unlike those that would be found in embedded systems. There are higher-level caches found in the R10000 that would not be found in most embedded systems, but we are able to ignore their effects by concentrating only on the hardware counters that relate to the first-level caches.

Secondly, our approach's effectiveness is bounded by the inherent limitations of the energy models that we derive below. The equations that we work with only deal with on-chip cache structures, and are not suitable for off-chip data transfers. This requires us to focus our work only on memory hierarchies such as would be found in System-on-Chip (SoC) devices. Another potential limitation is the fact that the accuracy of the models we use is closely tied to low-level circuit

parameters (capacitive coefficients, operating voltage, etc.). In this work we embrace this limitation and after initial validation to demonstrate the feasibility of the obtained absolute energy values, we concentrate mostly on relative energy calculations and comparisons.

2.2 SRAM Memory Structures

In this section we explore different memory architectures that could be found in System-on-Chip (SoC) devices. We analyze both a set-associative SRAM cache architecture and a SRAM memory architecture, which for our purposes we realize as a fully-associative cache. The purpose of this section is to introduce the structures to which we later apply our analytic energy dissipation models.

2.2.1 Set-associative SRAM Cache Architecture

Figure 2.1 shows a traditional set-associative cache architecture. When a read request is made to a set-associative cache, an address decoder is used to generate an enable signal for a single set that corresponds to the index field in the input address. For an m-way set associative-cache, there are m banks, each containing S sets of data. Each set contains St bits for status information, T bits for tag information, and L bytes (or $8*L$ bits) of actual data. The tag information along with the index of the set is decoded from the instruction reference address.

All of the blocks in the selected set are read in parallel, and the tag field of each set is compared to the tag field in the input address. If there is a tag match and the status bits indicate that the data is valid, the required data is in the cache, and the access is considered a *cache hit*. The data corresponding to the requested block is read out through an output multiplexor, which uses the word-

select field in the input address to determine which word to read back to the processor. If there is no match between the requested tag and any tag in the selected set, the access is considered a *cache miss*, and the cache must request the necessary data from a higher level in the memory hierarchy. We refer the reader to [15] for a more detailed explanation of set-associative cache functionality.

2.2.2 SRAM Memory Architecture

Figure 2.2 shows an example main memory architecture, realized as a fully-associative cache. Although it is possible to implement the tag store of a fully-associative cache using content addressable memory (CAM) [13], for our purposes it is easier to visualize the main memory structure as a standard set-associative cache, where the associativity is as large as possible. The characteristic variables of a fully-associative cache are very similar to those of a set-associative cache with L bytes of data in each set along with T bits of tag information and St bits for status information.

When a read request is made to a fully-associative cache, all the tag fields are read in parallel and compared to the tag field supplied in the address. If there is a match, the data related to the chosen tag field is read out after passing through an output word multiplexor, similar to the operation for a standard set-associative cache. For our purposes the main memory and cache structures are nearly identical, the only differences being the main memory has no need for an input row decoder and that since it is the highest level of our memory hierarchy, all accesses will be considered hits.

2.3 Dynamic Energy

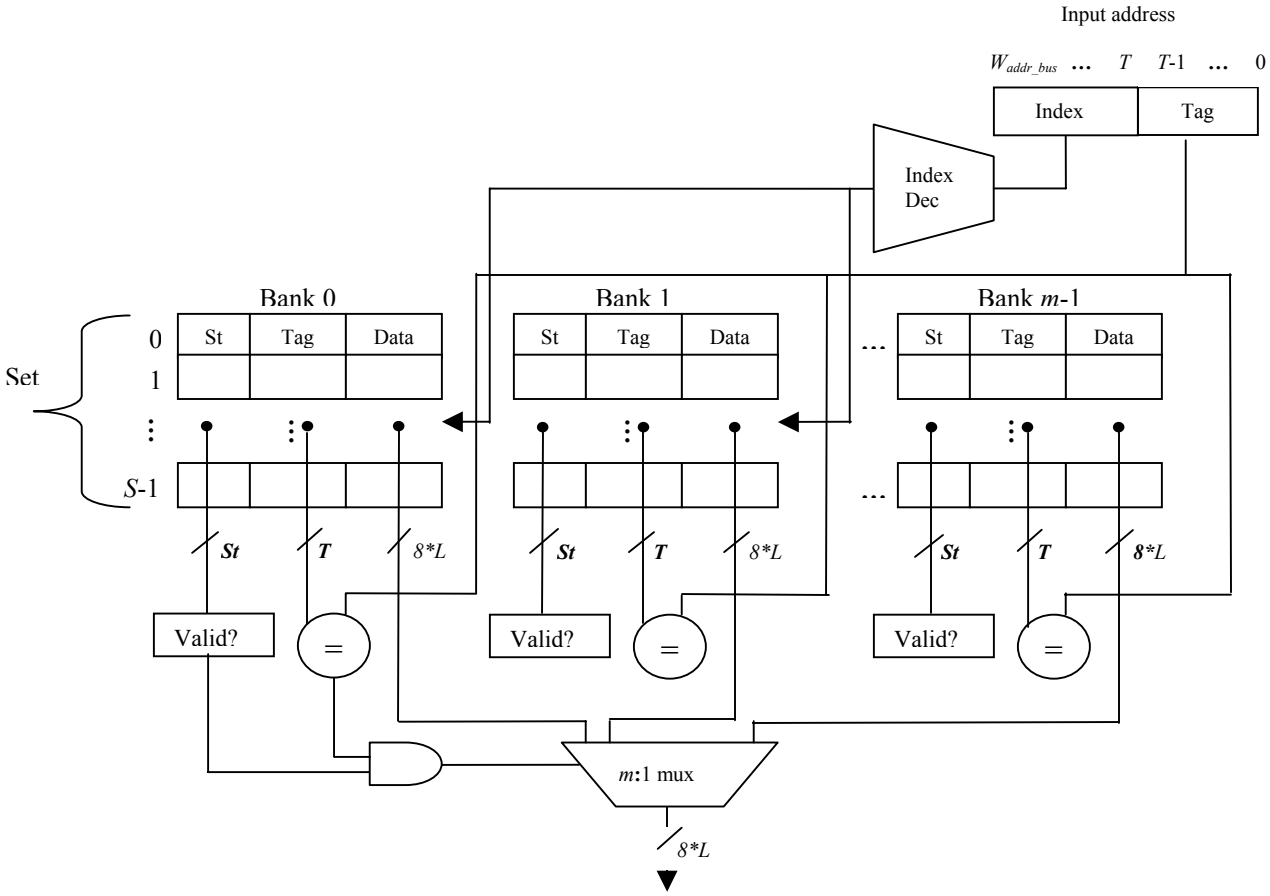

Figure 2.1. Standard set-associative cache architecture

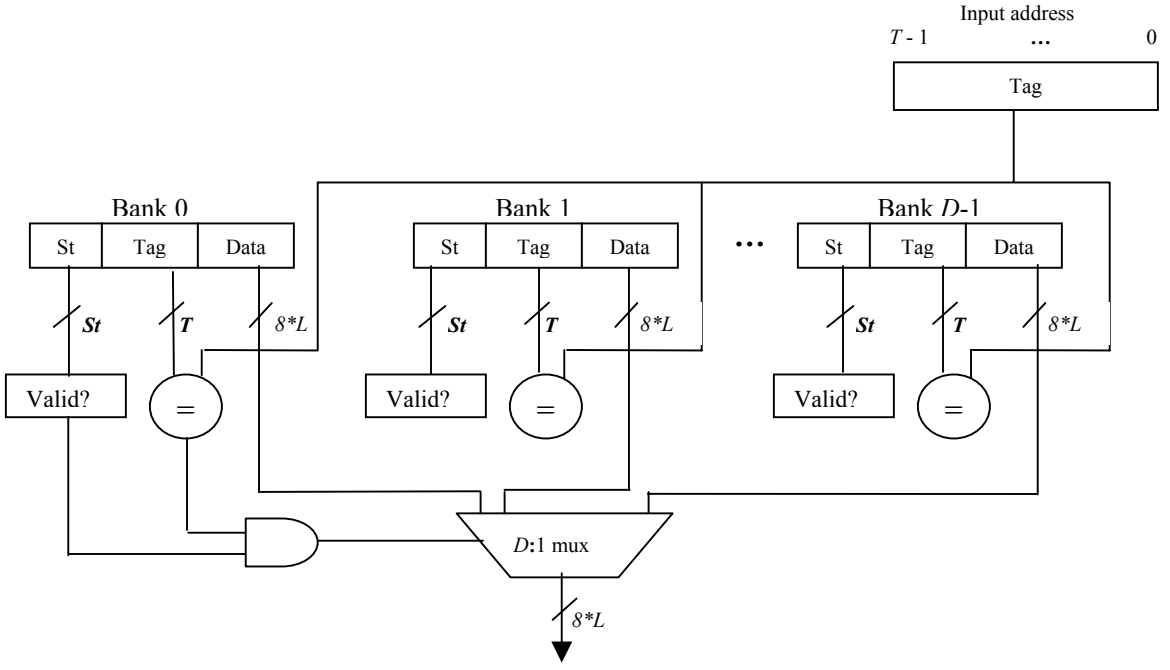

Figure 2.2. On-chip memory modeled as a fully-associative cache architecture

2.3.1 Background

Dynamic, or switching energy consumption is the result of a voltage change over a physical wire during the transition of a logic bit between boolean states. In previous design generations dynamic energy has been the overwhelmingly dominant contributor to the total energy equation, and as such it has been the main target of most architecture-level [3] and software-level [24] energy reducing techniques. Even though current trends indicate that the percentage of energy coming from switching will begin to decrease rapidly in upcoming years, it is still very important to be able to model dynamic energy efficiently and effectively.

2.3.2 Analytical Energy Equations

In this section we will enhance and clarify the cache energy dissipation models found in [17]. Most of the logic components in Figure 2.1 contribute a relatively small amount of energy dissipation, and the standard model only considers the following four factors:

- Address input-line dissipations: caused by the external address input signal transitions in the word select line decoder.
- Word-line dissipations: caused by the assertion of the word select line.
- Output-line dissipations: caused by the external output signal transitions of address and data signals sent to the CPU and other levels of the memory hierarchy.
- Bit-line dissipations: caused during the pre-charging and the actual read and write operations.

The energy dissipation in each component is a function of the average switching capacity, the estimated capacitance, and the operating voltage. In general, we can estimate the dynamic energy dissipation across each component by the following equation:

$$E_{comp} = 0.5 \cdot N_{comp} \cdot C_{comp} \cdot V_{comp}^{2} \qquad \text{(Eqn. 2.1)}$$

2.3.2.1 Address Input-line Energy Dissipation

The address input lines dissipate energy through the input gates of the index decoder. N_{ain} is the average switching of the decoder, which is active on every read attempt. Pr is the probability that any given bit will need to transition from a logical 0 to a 1 during any operation, and W_{addr_bus} is the number of bits in the address input signal:

$$N_{ain} = Pr \cdot N_{read} \cdot W_{addr\_bus} \qquad \text{(Eqn. 2.2)}$$

The address decoder needs to drive two lines in every memory bank (one each for *bit* and *bit_bar*) and the equivalent capacitance is proportional to the wire capacitance of those lines, C_{word}. We also need to factor in the drain capacitance of the input lines to the decoder, $C_{d,in}$ and the gate capacitance of the decoder transistors, $C_{g,dec}$:

$$C_{ain} = 2 \cdot L \cdot m \cdot C_{word} + C_{d,in} + 4 \cdot C_{g,dec} \qquad \text{(Eqn. 2.3)}$$

The energy dissipated in the address decoder is proportional to the equivalent capacitance, the switching frequency, and the operating voltage, and by direct application of Equation 2.1 we have:

$$E_{ain} = 0.5 \cdot V_{dd}^{2} \cdot Pr \cdot N_{read} \cdot W_{addr\_bus} \cdot \cdot (2 \cdot L \cdot m \cdot C_{word} + C_{d,in} + 4 \cdot C_{g,dec}) \qquad \text{(Eqn. 2.4)}$$

2.3.2.2 Word-line Energy Dissipation

Turning our focus to the word lines, we see that they dissipate energy during every cache access due to the assertion of the word select line by the word-line drivers. The word lines will have signal transitions during every read *and* write. Note that for the word select line we do not calculate any signal probabilities, since for each access exactly two word lines will have a signal transition:

$$N_{wordline} = 2 \cdot (N_{read} + N_{write}) \qquad \text{(Eqn. 2.5)}$$

The word line capacitance $C_{wordline}$ is a function of the wire capacitance of the word line C_{word} and also the gate capacitance of the bit-cell transistors, $C_{g,Q1}$. Each word-line drives the gates of two transistors in each bit cell:

$$C_{wordline} = m \cdot (8 \cdot L + T + St) \cdot (2 \cdot C_{g,Q1} + C_{word}) \qquad \text{(Eqn. 2.6)}$$

The total energy dissipated in the word lines is proportional to the equivalent capacitance, the switching frequency, and the operating voltage as per Equation 2.1:

$$E_{wordline} = V_{dd}^{\,2} \cdot (N_{read} + N_{write}) \cdot m \cdot \cdot (8 \cdot L + T + St) \cdot (2 \cdot C_{g,Q1} + C_{word}) \qquad \text{(Eqn. 2.7)}$$

2.3.2.3 Output-line Energy Dissipation

The output-lines dissipate energy when driving an address or data value towards the CPU or a different level in the memory hierarchy. The cache needs to drive an address towards different parts of the memory whenever there is a read or write miss or a write-back request:

$$N_{out,a2m} = Pr \cdot (N_{rmiss} + N_{wmiss} + N_{wb}) \cdot W_{addr\_bus} \quad \text{(Eqn. 2.8)}$$

The corresponding energy dissipation by the output address lines is a function of that switching activity and the capacity of those lines:

$$E_{aoutput} = 0.5 \cdot V_{dd}{}^2 \cdot Pr \cdot (N_{rmiss} + N_{wmiss} + N_{wb}) \cdot \cdot W_{addr\_bus} \cdot C_{out,a2m} \quad \text{(Eqn. 2.9)}$$

The cache needs to drive a data output to the CPU on every read access, where $W_{bus,d2c}$ is the number of bits in the data bus between the cache and the CPU:

$$N_{out,d2c} = Pr \cdot N_{read} \cdot W_{bus,d2c} \quad \text{(Eqn. 2.10)}$$

On any given write miss or write-back request, the memory-side data bus would be driven towards a different level of the memory hierarchy. On a write miss, the cache will need to send $W_{bus,d2m}$ bits to the lower-level of the memory hierarchy, and on a write-back the cache will request an entire set to be sent, 8**L* bits:

$$N_{out,d2m} = Pr \cdot (N_{wmiss} \cdot W_{bus,d2m} + N_{wb} \cdot 8 \cdot L) \quad \text{(Eqn. 2.11)}$$

The energy dissipation in the data output lines is proportional to the switching activity and the capacitance of those lines:

$$E_{doutput} = 0.5 \cdot V_{dd}{}^2 \cdot Pr \cdot (N_{rmiss} \cdot W_{bus,d2c} \cdot C_{out,d2c} + + (N_{wmiss} \cdot W_{bus,d2m} + N_{wb} \cdot 8 \cdot L) \cdot C_{out,d2m}) \quad \text{(Eqn. 2.12)}$$

The total energy dissipation in the output lines is the sum of the address line dissipation and the data line dissipation:

$$E_{output} = E_{aoutput} + E_{doutput}$$
$$= 0.5 \cdot V_{dd}{}^{2} \cdot Pr \cdot ((N_{rmiss} + N_{wmiss} + N_{wb}) \cdot$$
$$\cdot W_{addr\_bus} \cdot C_{out,a2m} + N_{read} \cdot W_{bus,d2c} \cdot C_{out,d2c} +$$
$$+ (N_{wmiss} \cdot W_{bus,d2m} + N_{wb} \cdot 8 \cdot L) \cdot C_{out,d2m}) \qquad \text{(Eqn. 2.13)}$$

2.3.2.4 Bit-line Energy Dissipation

The bit lines dissipate energy during precharging operations. We need to precharge each bit line every time we attempt a read:

$$N_{bit,\,pr} = Pr \cdot N_{read} \cdot 2 \cdot m \cdot (8 \cdot L + T + St) \qquad \text{(Eqn. 2.14)}$$

We can calculate the effective precharging capacitance $C_{bit,pr}$ as a function of the drain capacitance of the precharging transistor $C_{d,Qp.}$ Both the *bit* and the *bit_bar* lines need to be precharged:

$$C_{bit,\,pr} = S \cdot C_{d,Qp} \qquad \text{(Eqn. 2.15)}$$

Also, during read and write operations, the bit lines dissipate energy. A cache will need to perform a data read during every read request *and* every write request:

$$N_{bit,\,r} = Pr \cdot (N_{read} + N_{write}) \cdot 2 \cdot m \cdot$$
$$\cdot (8 \cdot L + T + St) \qquad \text{(Eqn. 2.16)}$$

We need to perform a write operation every time there is a successful write request, where W_{data_bus} is the width of the data that would be written to the cache. We also need to write a new line to the cache every time there is a read miss. For each write operation both the *bit* and *bit_bar* cells need to be written:

$$N_{bit,w} = Pr \cdot 2 \cdot (N_{write} \cdot (St + W_{data\_bus}) + \\ + N_{rmiss} \cdot (8 \cdot L + T + St)) \quad \text{(Eqn. 2.17)}$$

We can calculate the effective read/write capacitance $C_{bit,r/w}$ as a function of the drain capacitance in the read/write transistor $C_{d,Q1}$ and the wire capacitance of the bit line C_{bit}. The drain capacitance of the transistor is divided by two since each contact is shared between two adjacent bit cells:

$$C_{bit,r/w} = 2 \cdot S \cdot (0.5 \cdot C_{d,Q1} + C_{bit}) \quad \text{(Eqn. 2.18)}$$

The total energy dissipated in the bit lines is proportional to the operating voltage V_{dd} and the factors mentioned above:

$$\begin{aligned} E_{bit} &= 0.5 \cdot V_{dd}^{\ 2} \cdot (N_{bit,pr} \cdot C_{bit,pr} + N_{bit,w} \cdot C_{bit,r/w} + \\ &\quad + N_{bit,r} \cdot C_{bit,r/w}) \\ &= V_{dd}^{\ 2} \cdot Pr \cdot S \cdot ((8 \cdot L + T + St) \cdot (m \cdot N_{read} \cdot C_{d,Qp} + \\ &\quad + (C_{d,Q1} + C_{bit}) \cdot (m \cdot (N_{read} + N_{write}) + N_{rmiss})) + \\ &\quad + N_{write} \cdot (St + W_{data\_bus}) \cdot (C_{d,Q1} + C_{bit})) \end{aligned} \quad \text{(Eqn. 2.19)}$$

2.3.2.5 Total Energy Dissipation

There are certainly other sources of energy dissipation in the cache, such as the various logic gates required for the multiplexors and comparators. For this study, they are considered insignificant compared to the previously described factors. Therefore, for our model the total energy dissipation in the cache is given by:

$$E_{cache} = E_{ain} + E_{wordline} + E_{output} + E_{bit} \quad \text{(Eqn. 2.20)}$$

2.4 Static Energy

2.4.1 Background

As the name suggests, static, or leakage energy is consumed at all times and is not dependent on switching activity. In previous design technologies, leakage energy was a comparatively small component of overall energy consumption. However, many trends [7,18] indicate that leakage energy will soon dominate switching energy. This is because leakage power is generally considered to be the product of the supply voltage and leakage current [5], and as supply voltage is being lowered in current designs, leakage power is increasing exponentially due to technology scaling. For example, recent energy estimates for 0.13 micron process indicate that leakage energy accounts for 30% of L1 cache energy and as much as 80% of L2 cache energy [32]. It is for this reason that we need to consider the contributions of leakage energy in our models if our experimental methodology is to be of use in future years.

2.4.2 Leakage Energy Approximation

Similar to the dynamic power models from [17] that we adapted for our own purposes, there have been several attempts to model static power at the architectural-level. We could use some of these analytic formulas, such as those found in [5], to model our target memory structures. However, many if not all of these architectural-level static power models contain a technology constant k, which attempts to correct the leakage power estimation by acting as a technology scale. More recent

circuit fabrication technologies would be expected to have greater leakage power, and so when modeling those designs the value of *k* would be increased.

Therefore, with an understanding that we care only about the relative weight of leakage energy to dynamic energy, we can use the *k* constant to force a ratio between the two values. In other words, as an approximation we can take the per-cycle leakage energy consumption to be a fraction of the per-access switching energy consumption. Remembering the relationship between power and energy from Equation 1.1, we can write:

$$P_{switch} = E_{switch} / num\_access$$
$$P_{leak} = E_{leak} / num\_cycle$$
$$P_{leak} = k \cdot P_{switch}$$
$$\therefore E_{leak} = k \cdot E_{switch} \cdot num\_cycle / num\_access \qquad \text{(Eqn. 2.21)}$$

Taking $k = 0$ represents a fabrication technology where leakage energy is an insignificant factor, taking $0.1 <= k <= 0.2$ represents current trends, and taking $0.5 < k <= 1.0$ represents a futuristic scenario where leakage energy constitutes a sizeable portion of the overall memory energy consumption. Note that similar approaches have been used by previous researchers as well (e.g., [8]).

2.5 Application of Energy Models

In this section we will take the analytical models for dynamic energy consumption and apply them to three unique memory hierarchies that are commonly found in System-on-Chip (SoC) devices. In

doing so we will specifically determine the run-time data required to estimate energy, and we will be able to greatly simplify Equation 2.20 for use in our optimization experiments.

2.5.1 Target Memory Hierarchy

Typically, an embedded processor would not contain as many levels of cache as a high-performance processor. This is due to the fact that embedded devices often need to adhere to a small form-factor, and cache structures require large amounts of chip space. Also, since caches are a dominating contributor to the overall chip power consumption [27], embedded systems designers may choose to simplify the overall memory hierarchy to reduce power, at the cost of performance. In this section we discuss three different memory hierarchies of increasing levels of complexity:

- Memory configuration I describes a cache-less memory hiearchy (Figure 2.3). In this configuration both the instruction and data memories are connected directly to the CPU with address and data buses. This configuration would be used in systems where simplicity, low-cost, and low-power are important design constraints. Another important advantage of this configuration is its deterministic behavior [13]. Embedded systems with caches can have complex instruction latencies, since instruction and data fetch misses can cause huge delays

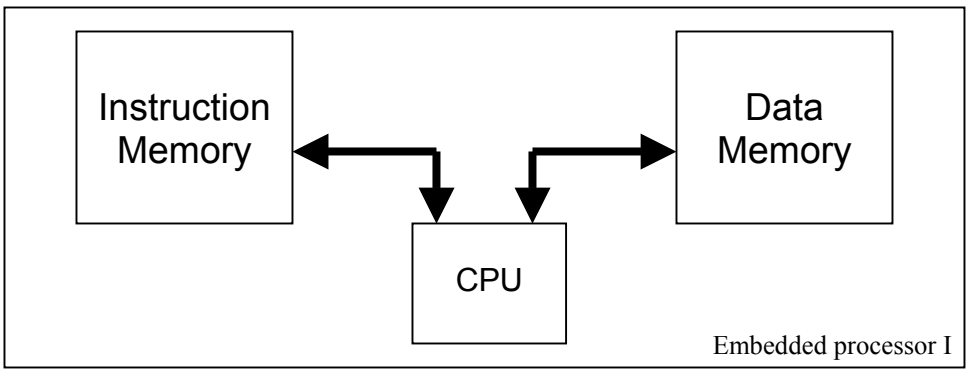

Figure 2.3. Configuration I: Cache-less memory hierarchy

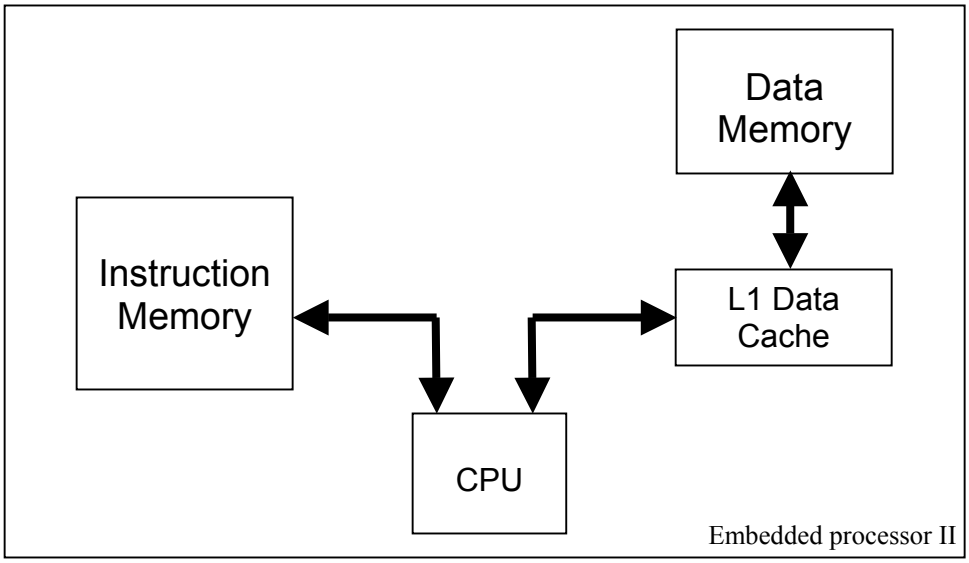

Figure 2.4. Configuration II: Memory hierarchy with data caching only

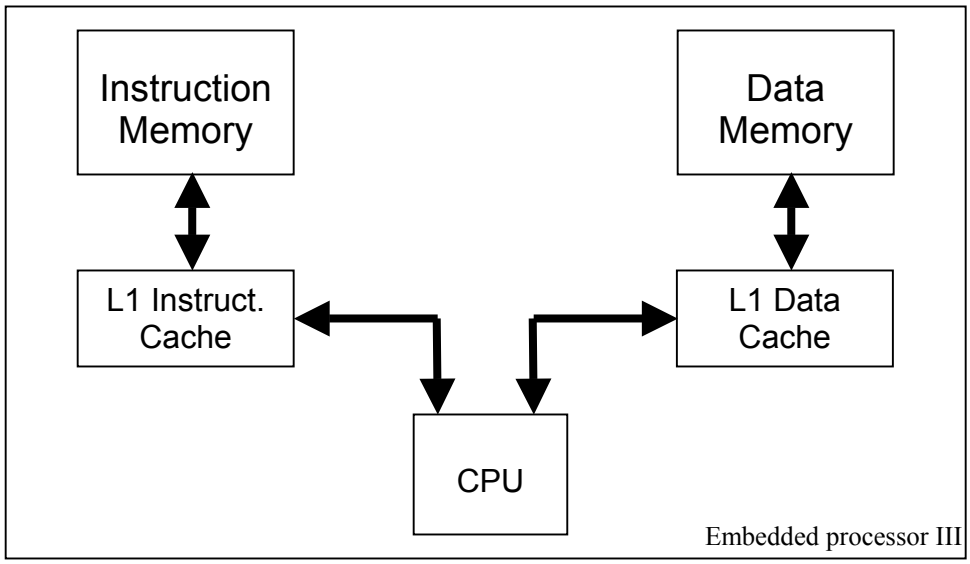

Figure 2.5. Configuration III: Memory hierarchy with instruction and data caching

for individual instructions. Therefore system designers that require consistent energy and time consumption per instruction would consider this memory configuration.

- Memory configuration II describes a memory hiearachy with a data cache but no instruction cache (Figure 2.4). This configuration would be utilized by a system designer who wants to

leverage data locality to improve application performance but still wants to ensure some of the deterministic behavior of the instruction fetch unit.

- Memory configuration III describes a memory hiearachy with both an instruction and data cache (Figure 2.5). This configuration is what can be found in many current embedded processors, for examples one can examine SoCs that utilize the ARM920T [13]. Having both an instruction and a data cache would greatly increase the processor power requirement, but would also lead to non-trivial speedups over the first two configurations.

2.5.2 Deriving Final Energy Equations

Before deriving the values for the run-time data in our memory hierarchies, we can first simplify our equations by setting values for the information that is consistent across all three configurations. Making the assumption that signal values are independent and have a equal probability of being 0 or 1, we can set $Pr = 0.5$. We can set the system address width and the CPU data request width to be 32 bits, a common configuration for embedded systems. Using our notation from above, we can set $W_{addr\_bus} = 32$ and $W_{bus,d2c} = 32$. Also for each memory structure across all configurations it is valid to assume that we will need only one status bit to be set active when the requested line has valid data and therefore we can set $St = 1$.

As will be discussed in more detail in Chapter 3, we set the instruction memory size to be the code (executable) size of the application running on it. We also set the data memory size to be the size of the variables required to run the application. In real systems the memory will consist of several small memory modules, and therefore the memory size can only be set to a multiple of the

module size. Our approximation allows us to highlight the effect of different optimizations that change the code (executable) size and the number of data variables.

2.5.2.1 Cache-less Memory Hierarchy

Since the CPU-side data bus is 32 bits wide, we can implement our instruction and data memories with a 32-bit line size, and therefore we set $L = 4$ bytes. Also, assuming that our memories are word addressable, we would use all our available address bits for tag information in our fully-associative cache scheme, and therefore we set $T = 32$ bits. For a general set-associative cache, we can calculate the number of index rows as $S = D / (L \cdot m)$ and given that for our data and instruction memories $S = 1$, we have $m = D / 4$.

We can greatly simplify the energy consumption equations for our instruction memory by noting that instruction memories generally do not allow for write operations. Consequently, we can set $N_{write} = 0$. For this configuration N_{read} for the instruction memory will be set to the number of instructions executed by they CPU. Given our cache-less scheme, there is no possibility for any type of cache misses, since all of the application instructions are resident in instruction memory. Therefore for the instruction memory we can set $N_{rmiss} = 0$ and $N_{wb} = 0$. By direct application of Equation 2.20 we can determine the energy consumption of the instruction memory in our cache-less configuration as:

$$\begin{aligned} E_{imem} = {V_{dd}}^2 \cdot N_{read} \cdot (D_{imem} \cdot (32.25 \cdot C_{word} + 32.5 \cdot \\ \cdot C_{g,Q1} + 8.125 \cdot (C_{d,Qp} + C_{d,Q1} + C_{bit})) + \\ + 8 \cdot C_{d,in} + 32 \cdot C_{g,dec} + 8 \cdot C_{out,d2c}) \end{aligned} \qquad \text{(Eqn. 2.22)}$$

We cannot make the same write-access simplification for the data memory component of configuration I as we did for the instruction memory. For the data memory, N_{read} and N_{write} will be set to the number of data reads and writes requested by they CPU. There is still no possibility for data memory misses, and therefore for our data memory we can set $N_{rmiss} = 0$, $N_{wmiss} = 0$, and $N_{wb} = 0$. By direct application of Equation 2.20 we can determine the energy consumption of the data memory in our cache-less configuration as:

$$\begin{aligned} E_{dmem} = V_{dd}^{\,2} \cdot (N_{read} \cdot (D_{dmem} \cdot (32.25 \cdot C_{word} + 32.5 \cdot \\ \cdot C_{g,Q1} + 8.125 \cdot (C_{d,Qp} + C_{d,Q1} + C_{bit})) + \\ + 8 \cdot C_{d,in} + 32 \cdot C_{g,dec} + 8 \cdot C_{out,d2c}) + N_{write} \cdot \\ \cdot (D_{dmem} \cdot (16.25 \cdot C_{word} + 32.5 \cdot C_{g,Q1} + 8.125 \cdot \\ \cdot (C_{d,Q1} + C_{bit})) + 16.5 \cdot (C_{d,Q1} + C_{bit}))) \end{aligned} \qquad \text{(Eqn. 2.23)}$$

The total dynamic energy consumption for configuration I is the sum of the energy consumptions of the instruction and data memories:

$$E_{configI} = E_{imem} + E_{dmem} \qquad \text{(Eqn. 2.24)}$$

2.5.2.2 Memory Hierarchy with Data Caching

The instruction memory for configuration II is equivalent to the instruction memory for configuration I, and consequently Equation 2.22 is still valid. This configuration now contains an L1 data cache, whose parameters are set by the hardware in the MIPS R10000 processor, as will be seen in chapter 3. The 32-Kbyte L1 data cache is two-way set associative with a block size of 8 words. Translating that description into our cache architecture variables, we have D = 32Kb, m = 2, and L = 32 bytes. As before, the required number of index rows is determined by $S = D / (L \cdot m)$, and

consequently for the L1 data cache we have $S = 512$. To index the rows we require $I = log_2(S) = 9$ bits. The tag bits are required for the rest of the address bus, so $T = W_{addr\_bus} - I = 23$ bits. For the data cache, N_{read} and N_{write} will be set to the number of data reads and writes requested by the CPU. By direct application of Equation 2.20 we can determine the energy consumption of the data cache, understanding that we now have to worry about cache misses:

$$\begin{aligned} E_{dcache} = V_{dd}^2 \cdot (N_{read} \cdot (1584 \cdot C_{word} + 8 \cdot C_{d,in} + 32 \cdot C_{g,dec} + \\ + 1120 \cdot C_{g,Q1} + 8 \cdot C_{out,d2c} + 143360 \cdot (C_{d,Qp} + C_{d,Q1} + \\ + C_{bit})) + N_{write} \cdot (1120 \cdot C_{g,Q1} + 560 \cdot C_{word} + 151808 \cdot \\ \cdot (C_{d,Q1} + C_{bit})) + N_{rmiss} \cdot (8 \cdot C_{out,a2m} + 71680 \cdot (C_{d,Q1} + \\ + C_{bit})) + N_{wmiss} \cdot (8 \cdot C_{out,a2m} + 8 \cdot C_{out,d2m}) + N_{wb} \cdot (8 \cdot \\ \cdot C_{out,a2m} + 64 \cdot C_{out,d2m})) \end{aligned} \quad \text{(Eqn. 2.25)}$$

For the data memory, N_{read} and N_{write} will now be set by the read misses and write misses of the data cache, and it makes sense to set the line-size L to be the line-size of the data cache. The dynamic energy consumption is now:

$$\begin{aligned} E_{dmem} = V_{dd}^{\ 2} \cdot (N_{read} \cdot (D_{dmem} \cdot (24.75 \cdot C_{word} + 17.5 \cdot \\ \cdot C_{g,Q1} + 4.375 \cdot (C_{d,Qp} + C_{d,Q1} + C_{bit})) + \\ + 8 \cdot C_{d,in} + 32 \cdot C_{g,dec} + 64 \cdot C_{out,d2c}) + N_{write} \cdot \\ \cdot (D_{dmem} \cdot (8.75 \cdot C_{word} + 17.5 \cdot C_{g,Q1} + 4.375 \cdot \\ \cdot (C_{d,Q1} + C_{bit})) + 16.5 \cdot (C_{d,Q1} + C_{bit}))) \end{aligned} \quad \text{(Eqn. 2.26)}$$

The total dynamic energy consumption for configuration II is the sum of the energy consumptions of the instruction memory, the data cache, and the data memory:

$$E_{configII} = E_{imem} + E_{dcache} + E_{dmem} \quad \text{(Eqn. 2.27)}$$

2.5.2.3 Memory Hierarchy with Instruction and Data Caching

The data side of the memory hierarchy for configuration III is equivalent to that for configuration II, and consequently Equations 2.25 and 2.26 are still valid. The configuration now contains an L1 instruction cache, whose parameters are set by the hardware in the MIPS R10000 processor, as will be seen in Chapter 3. The 32-Kbyte L1 instruction cache is two-way set associative with a block size of 16 words. Translating that description into our cache architecture variables, we have D = 32Kb, m = 2, and L = 64 bytes. As before, the required number of index rows is determined by $S = D / (L \cdot m)$, and consequently for the L1 instruction cache we have S = 256. To index the rows we require $I = log_2(S)$ = 8 bits. The tag bits are required for the rest of the address bus, so $T = W_{addr\_bus} - I$ = 24 bits. For the instruction cache, N_{read} will be set to the number of instructions executed by they CPU. By direct application of Equation 2.20 we can determine the energy consumption of the instruction cache as:

$$\begin{aligned} E_{icache} = V_{dd}{}^2 \cdot (N_{read} \cdot (&3122 \cdot C_{word} + 8 \cdot C_{d,in} + \\ &+ 32 \cdot C_{g,dec} + 2148 \cdot C_{g,Q1} + 8 \cdot C_{out,d2c} + \\ &+ 137472 \cdot (C_{d,Qp} + C_{d,Q1} + C_{bit})) + \\ &+ N_{rmiss} \cdot (8 \cdot C_{out,a2m} + 68736 \cdot (C_{d,Q1} + C_{bit}))) \end{aligned} \qquad \text{(Eqn. 2.28)}$$

For the instruction memory, N_{read} will be set by the number of read misses, N_{rmiss}, of the instruction cache. Again we note that for the instruction memory hierarchy, there are no write requests and consequently N_{write} = 0 and N_{wb} = 0 for both the instruction cache and instruction memory. It makes sense to set the line-size L to be the line-size of the instruction cache. The dynamic energy consumption is now:

$$E_{imem} = V_{dd}^{\ 2} \cdot N_{read} \cdot (D_{imem} \cdot (24.5 \cdot C_{word} + 16.8 \cdot \cdot C_{g,Q1} + 4.2 \cdot (C_{d,Qp} + C_{d,Q1} + C_{bit})) + + 8 \cdot C_{d,in} + 32 \cdot C_{g,dec} + 128 \cdot C_{out,d2c}) \quad \text{(Eqn. 2.29)}$$

The total dynamic energy consumption for configuration III is the sum of the energy consumptions of the instruction memory, the instruction cache, the data memory, and the data cache:

$$E_{configIII} = E_{icache} + E_{imem} + E_{dcache} + E_{dmem} \quad \text{(Eqn. 2.30)}$$

2.5.2.4 Evaluating Capacitive Coefficients

As can be seen from our equations above, there are numerous capacitive coefficients that need to be evaluated in order to use our model. Table 2.1 details what values for the various coefficients we are using. These values come from the data for the 0.8-micron transistor implementation found in [40]. We also decided to set V_{dd} = 3.3 (V) for our testing although as long as it is set constant its value makes little difference.

Constant	Description	Value (units)
$C_{d,in}$	address decoder input drain capacitance	.12 (pF)
C_{word}	word line wire capacitance	1.8 (fF/bit)
$C_{g,dec}$	address decoder gate capacitance (first level only)	.236 (pF)
$C_{g,Q1}$	memory cell gate capacitance	11.6 (fF)
$C_{out,a2m}$	capacitive load on the mem-side address line driver	0.5 (pF)
$C_{out,d2c}$	capacitive load on the cpu-side data line driver	0.5 (pF)
$C_{out,d2m}$	capacitive load on the mem-side data line driver	0.5 (pF)
$C_{d,Qp}$	precharge transistor drain capacitance	72.3 (fF)
$C_{d,Q1}$	memory cell drain capacitance	2.32 (fF)
C_{bit}	bit line wire capacitance	4.4 (fF/bit)

Table 2.1. Common values for capacitive coefficients

CHAPTER 3

Experimental Methodology

3.1 System Hardware

In this section we present an overview of the hardware that we utilized for our compiler optimization experiments. The purpose of this overview is to provide enough background so that later in this chapter we can suitably explain how we were able to leverage a parallel computing resource in order to simulate the behavior of an embedded chip.

3.1.1 SGI Origin2000

The SGI Origin2000 [31] machine located at the Center for Parallel Computing [30] at Northwestern University is a shared memory multiprocessor with eight MIPS R10000 processors and 2 GB of memory. It runs the IRIX operating system, which provides a standard multi-user time-sharing UNIX operating system along with a wealth of software development tools. These development tools, especially the compilation and profiling tools, make the Origin an attractive target for any type of compiler optimization research.

In the Origin shared memory architecture, CPUs, memory, and I/O devices are not connected to a central bus [19]. Instead, hub and router chips connect the CPUs to one another, while memory and peripherals are distributed among the CPUs (Figure 3.1). This unique architecture allows us to easily target designs to a single MIPS R10000 processor with its own memory and I/O system.

3.1.2 MIPS R10000

The MIPS R10000 is a four-way superscalar RISC CPU [26] that implements the MIPS IV instruction set architecture. Running at a clock speed of 195 MHz, the R10000 has a 5-stage integer pipeline and a 7-stage floating-point pipeline. The R10000 pipeline supports both out-of-order and speculative execution.

The R10000 contains a L1 cache split into separate instruction and data caches. The L1 instruction cache is 32 KB two-way set associative cache with a line size of 64 bytes. The L1 data cache is also 32 KB and two-way set associative, with a line size of 32 bytes. The R10000 also

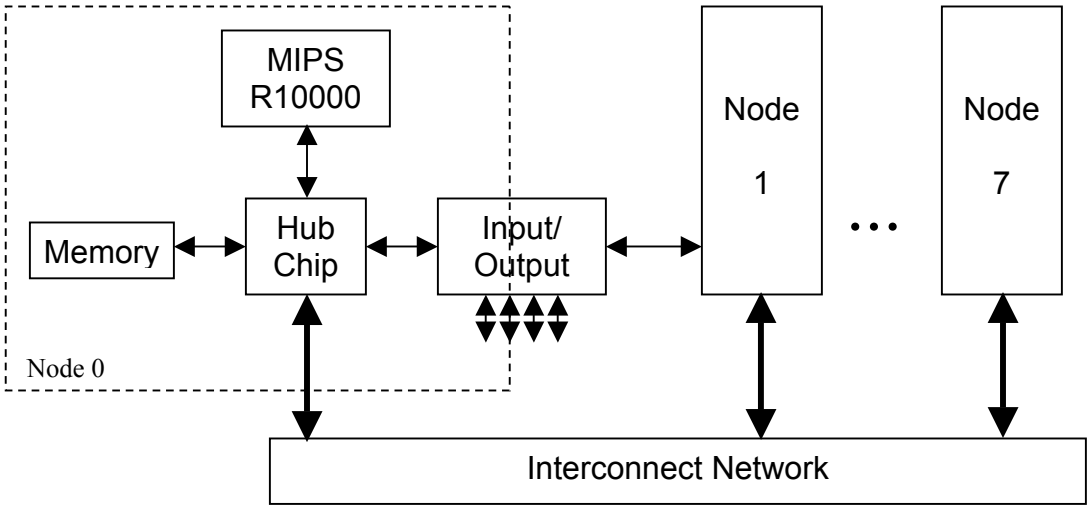

Figure 3.1. SGI Origin2000 architecture

contains a unified L2 cache located off-chip, which is a 4 MB two-way set associative cache with a line size of 128 bytes.

The MIPS R10000 contains two 32-bit hardware event counters, each of which can be assigned to monitor any one of 16 events. The counters can either be used to tabulate the occurrence of a hardware event or they can be conditioned to create an interrupt when the counter overflows. The IRIX kernel creates a set of virtual counter registers for every process, so that events can be accumulated accurately at the process-level. As will be explained in further detail in the next section, the SGI execution profiling tools make use of these hardware event counters.

3.2 System Software

As stated previously, our main reason for using the SGI Origin2000 for our research is the fact that it comes with a wealth of utilities for software development. In this section we discuss some of the more popular SGI tools used for performance tuning and execution profiling while showing example outputs. Later in this chapter we will discuss in more detail how we target the use of these tools to perform our energy optimization experiments on the MIPS R10000 processor.

3.2.1 MIPSPro

The MIPSPro from SGI is an extremely advanced compiling system that allows for a wide range of automatic and user-directed optimizations. Both Fortran and C compilation front-ends are available but to simplify matters for our work we care only about the C side. The MIPSPro compiler allows us to pick and choose both loop nest optimizations and interprocedural analyzing optimizations.

There are four major optimizing modes of the MIPSPro compiler that perform different performance optimizations:

- **–O0**: No code optimization is done.
- **–O1**: Performs copy propagation, dead code elimination, and other local optimizations.
- **–O2**: Performs non-loop if conversion, with some cross-iteration optimizations (no write/write elimination on loops without trip counts). This mode also performs loop unrolling and recurrence fixing. Basic blocks are reordered to minimize the number of taken branches.
- **–O3**: Performs more if conversion and software pipelining. This mode also activates the Loop Nest Optimizer (LNO) that aggressively attempts locality-enhancing optimizations such as tiling, fission/fusion, and loop interchange [29,42].

Used in conjunction with these four optimization modes is the **–IPA** flag that turns on the interprocedural analysis optimizations, which include function inlining, interprocedural constant propagation, and dead function elimination. More details on these optimizations can be found in Chapters 4 and 5 and also in [28,41,42,29].

One of the more advanced features of the MIPSPro compiler is the ability to see the effect of optimizations on both assembly source code and on the original C source code. While this is useful in that code annotations in the form of comments are added signifying specific optimizations, the resultant C code is difficult to read and is not written in a portable way.

For our energy optimization experiments we used compiler flags to tailor different interprocedural and loop nest optimizations, and in Chapters 4 and 5 we will go into more detail

about those flags. In general, however, the operation of the MIPSPro compiler is too hearty of a subject to be given a worthy treatment here; we refer the reader to [25] for an in-depth description of its features and optimizations.

3.2.2 Perfex

The *perfex* tool [31] is a profiling tool that runs a supplied program and records data about the run. The goal of *perfex* is to identify what types of problems are hurting application performance. As mentioned before, the MIPS R10000 processor contains two 32-bit counters that can be used to measure the values of 32 different hardware events. When run, *perfex* sets up the IRIX kernel interface to the hardware event counters and forks the supplied program.

Table 3.1 gives a list of the countable hardware events on the MIPS R10000. The *perfex* tool can either give an exact count of any two hardware events (one from each column) or can time-multiplex all 32 events and give an extrapolated total of each. Figure 3.2 shows an example output when the *perfex* tool was run on the *mesa* benchmark from the SPEC CFP2000 benchmark suite. Later in this chapter we will discuss exactly which hardware event counters are important to us in our energy optimization investigations.

3.2.3 Speedshop

The SpeedShop tool, also known as the *ssrun* command [31], runs a supplied program while sampling the state of the program counter and stack at predefined intervals. The goal of Speedshop is to identify the sections of code where there are performance problems in order to target aggressive optimizations. The output of a Speedshop run is a data file representing the percentage of time

Event Number	Counter 0 Event	Event Number	Counter 1 Event
0	cycles	16	cycles
1	instructions issued	17	instructions graduated
2	memory data access	18	memory data loads graduated
3	memory stores issued	19	memory data stores graduated
4	store-conditionals issued	20	store conditionals graduated
5	store-conditionals failed	21	FP instructions graduated
6	branches decoded	22	quadwords written from L1
7	quadwords written from L2	23	TLB refill exceptions
8	correctable ECC errors on L2	24	branches mispredicted
9	L1 cache misses (instruction)	25	L1 cache misses (data)
10	L2 cache misses (instruction)	26	L2 cache misses (data)
11	L2 cache way mispred (instr.)	27	L2 cache way mispred (data)
12	external intervention requests	28	external intervention hits in L2
13	external invalidate requests	29	external invalidate hits in L2
14	instructions done	30	stores to CleanExclusive L2 blocks
15	instructions graduated	31	stores to Shared L2 blocks

Table 3.1. MIPS R10000 countable events

spent in different code sections that can be displayed using the SGI tool *prof*. Figure 3.3 shows an example output when the *ssrun*/*prof* tools were run on the *mesa* benchmark.

In order to gather its data, Speedshop takes statistical samples of hardware counters in order to interrupt the program at specified intervals. There are different sampling methods that can reflect different execution trends of the supplied program. For example, executing *ssrun* with the –fgi_hwc flag interrupts execution every 6553 graduated instructions, the profiling output being likely to

```
> perfex -e 15 -e 25 mesa -frames 10 -meshfile mesa.in -ppmfile mesa.ppm
Summary for execution of mesa -frames 10 -meshfile mesa.in -ppmfile mesa.ppm

15 Graduated instructions.....................................   3588334477
25 Primary data cache misses..................................      3986991
>
```

Figure 3.2. Sample *perfex* output for the *mesa* benchmark

```
> ssrun -fgi_hwc mesa -frames 10 -meshfile mesa.in -ppmfile mesa.ppm
> prof mesa.fgi_hwc.m1644328
------------------------------------------------------------------------
Summary of R10K perf. counter overflow PC sampling data (fgi_hwc)--
                      546989: Total samples
   Graduated instructions (17): Counter name (number)
                        6553: Counter overflow value
                  3584418917: Total counts
------------------------------------------------------------------------
Function list, in descending order by counts
------------------------------------------------------------------------
 [index]        counts     %   cum.%   samples  function (dso: file, line)

     [1]    1015334926  28.3%  28.3%    154942  _doprnt (libc.so.1: doprnt.c, 227)
     [2]     476789727  13.3%  41.6%     72759  sample_1d_linear (mesa: texture.c, 511)
     [3]     417707879  11.7%  53.3%     63743  general_textured_triangle (mesa: triangle.c, 473)
     [4]     346476769   9.7%  62.9%     52873  get_1d_texel (mesa: texture.c, 382)
     [5]     197382913   5.5%  68.5%     30121  apply_texture (mesa: texture.c, 1978)
     [6]     196557235   5.5%  73.9%     29995  clear (mesa: osmesa.c, 620)
     [7]     192992403   5.4%  79.3%     29451  __floor (libm.so: floor.s, 51)
     [8]     159205135   4.4%  83.8%     24295  sample_linear_1d (mesa: texture.c, 702)
     [9]     151230134   4.2%  88.0%     23078  gl_depth_test_span_less (mesa: depth.c, 422)
    [10]     150542069   4.2%  92.2%     22973  write_color_span (mesa: osmesa.c, 730)
    [11]      98242576   2.7%  94.9%     14992  memcpy (libc.so.1: bcopy.s, 329)
    [12]      72253378   2.0%  96.9%     11026  fprintf (libc.so.1: fprintf.c, 23)
...
              19659   0.0% 100.0%         3  **OTHER** (includes excluded DSOs, rld, etc.)
         3584418917 100.0% 100.0%    546989  TOTAL
```

Figure 3.3. Sample abbreviated *ssrun*/*prof* output for the *mesa* benchmark

emphasize functions that execute a large number of instructions. The sampling methods can be set by any of the 32 R10000 hardware events, including clock cycles and cache misses.

Besides profiling by sampling hardware event counters, Speedshop can create an ideal time profile when it is executed with the –ideal flag. During ideal time, or basic block profiling, a copy of the supplied program is modified to generate interrupts at the end of every function call. Speedshop then can provide a count of every function and call-site pair.

3.2.4 Dprof

The *dprof* tool runs a supplied program while sampling memory addresses. The goal of *dprof* is to identify data segments that cause performance problems. The output of *dprof* is a histogram of reads and writes to data addresses that are grouped by virtual page (16KB under IRIX). Figure 3.4 shows an example output when the *dprof* tool was run on the *mesa* benchmark.

Similar to Speedshop *dprof* performs statistical sampling of hardware counters in order to interrupt program execution. However, while Speedshop enjoys a relatively small performance overhead, running *dprof* using any type of fine grain sampling will lead to extremely slow program execution.

3.3 Benchmarks

We measured the effect of compiler optimizations on power and performance using benchmarks

```
> dprof -hwpc mesa -frames 10 -meshfile mesa.in -ppmfile mesa.ppm
-------------------------------------------------
   address      thread       reads          writes
-------------------------------------------------
0x00100fd000      0           5430               0
0x0010101000      0              1               0
0x0010105000      0              2               2
0x0010109000      0              1               1
0x001010d000      0              1               2
0x0010111000      0              0               1
0x0010115000      0              0               1
0x0010221000      0              0              24
0x0010225000      0              0              30
0x0010229000      0              0              20
0x001022d000      0              0              18
0x0010231000      0              0              17
...
0x007ffe8000      0          71399           43335
0x007ffec000      0          30100           17685
0x007fff0000      0           1944            1273
0x007fff4000      0          23416            1924
```

Figure 3.4. Sample abbreviated *dprof* output for the *mesa* benchmark

from the SPEC CPU2000 [38] and MediaBench [20] suites. Table 3.2 gives a brief overview of our selected benchmarks. To gather the data for this table, we compiled each of the benchmarks with no optimizations, and then executed them with the *test* inputs for the SPEC benchmarks and the provided inputs for the MediaBench benchmarks. Using the SGI tools from above, we measured how many (graduated) instructions were executed, the size of the benchmark executable, and the size of the data variables. As can be seen from Table 3.2 we picked a set of benchmarks that have a broad range of instruction counts, code sizes, and data sizes. Our reasoning for choosing our benchmarks along with a more detailed description is given below.

3.3.1 SPEC Benchmarks

The Standard Performance Evaluation Corporation (SPEC) [38] is a non-profit industry consortium whose goal is to produce computing benchmarks. To this end they released the CPU2000 benchmark suite, which contains both floating-point and integer benchmarks. These benchmarks are a solid workload approximation for embedded systems that are required to do heavy number crunching. We selected the following three benchmarks from the CFP2000 floating-point subgroup of the SPEC CPU2000 benchmark suite:

- *art* – the Adaptive Resonance Theory 2 (ART 2) neural network uses thermal imaging to recognize objects. The *art* benchmark implements this algorithm and attempts to recognize both a helicopter and an airplane in a thermal image. We choose this benchmark as our longest running CFP2000 benchmark, as it demonstrates a floating-point workload of over 10 billion instructions. The *art* benchmark is also our smallest CFP2000 benchmark in terms of executable size, needing just over 40KB of instruction memory.

- *equake* – simulates the propagation of seismic waves in large heterogeneous valleys using a finite element method. The *equake* benchmark implements this algorithm with a sparse matrix input that simulates the 1994 Northridge Earthquake aftershock in the San Fernando Valley of Southern California. We chose this benchmark as our largest data size CFP2000 benchmark, as it demonstrates a floating-point workload requiring over 10 MB of data memory.

- *mesa* – implements a 3-D graphics library similar to the OpenGL library. The *mesa*

Benchmark	Source	Description	Instructions Executed (Millions)	Executable Size (KB)	Data Size (MB)
art	CFP2000	image recognition	14,000	43	2.3
equake	CFP2000	seismic wave simulator	2,200	53	11.0
mesa	CFP2000	3-D graphics library	3,600	1,300	9.2
gzip	CINT2000	data compression	5,500	140	6.7
vortex	CINT2000	object-oriented database	15,000	1,200	22.0
twolf	CINT2000	place and route simulator	450	550	.21
mpeg2decode	MediaBench	MPEG-2 decoder	340	150	.46
mpeg2encode	MediaBench	MPEG-2 encoder	2,900	180	1.4
rawcaudio	MediaBench	adpcm encoder	18	18	.05

Table 3.2. Benchmarks used for optimization experiments

benchmark uses this library and attempts to map a 2-D scalar field into a 3-D contour mapped object. We chose this benchmark as our largest executable size CFP2000 benchmark, as it demonstrates a floating-point workload that requires over 1 MB of instruction memory.

We also selected the following three benchmarks from the CINT2000 integer subgroup of the SPEC CPU2000 benchmark suite:

- *gzip* – a popular compression program that implements Lempel-Ziv coding (LZ77). The *gzip* benchmark reads random input and compresses and decompresses at several different blocking factors, performing no file I/O except for the initial file read in order to isolate memory performance. We chose this benchmark as our smallest executable size CINT2000 benchmark, as it demonstrates an integer workload that requires less than 140 KB of instruction memory.

- *vortex* – VORTEx, or Virtual Object Runtime Expository, is an object-oriented database transaction program. The *vortex* benchmark implements a subset of the VORTEx database that handles schema related to three different databases: a mailing list, a parts list, and geometric data. We chose this benchmark as our largest data size CINT2000 benchmark, as it demonstrates an integer workload requiring over 22 MB of data memory.

- *twolf* – TimberWolfSC is a placement and global routing package that uses a simulated annealing algorithm. The *twolf* benchmark is a version of TimberWolfSC customized to increase cache misses that is run with three different commonly used benchmark circuits. We chose this benchmark because its memory access patterns were created in such a way to highlight the differences between memory hierarchies.

3.3.2 MediaBench Benchmarks

MediaBench [20] is a set of integer benchmarks developed at UCLA the goal of which is to accurately represent the workload of multimedia and communication systems. The MediaBench benchmarks are appropriate for our target embedded system when one considers the application of that system in multimedia devices such as cell phones and mp3 players. These benchmarks are notoriously difficult to compile, and we were limited to the subset of the MediaBench benchmarks that the MIPSPro compiler could handle. We selected the following benchmarks from the MediaBench benchmark suite:

- *mpeg2encode* – MPEG-2 is a popular video compression standard. The benchmark *mpeg2encode* implements the discrete cosine transform used for encoding MPEG-2 video. It takes as its input static frames to be combined into video. We chose this benchmark as our longest running MediaBench benchmark, as it demonstrates a multimedia workload of almost 3 billion instructions.

- *mpeg2decode* – as the companion benchmark to *mpeg2encode*, *mpeg2decode* implements the inverse discrete cosine transform used for decoding MPEG-2 video. We chose this benchmark for direct comparison to the *mpeg2encode* benchmark.

- *rawcaudio* – Adaptive Differential Pulse Code Modulation (ADPCM) is a simple form of audio coding. The *rawcaudio* benchmark implements an APDCM encoder that is run with a raw audio file as its input. We chose this benchmark because it has a smaller overall memory footprint and runs faster than any of our other benchmarks.

3.4 Investigative Approach

3.4.1 Effectively Utilizing the R10000

There are several features of the MIPS R10000 processor that make it difficult to use for our energy optimizations. Several high-performance design features, such as out-of-order execution, multi-instruction issuing, and speculative execution, make for a quality general-purpose processor but are generally not found in embedded processors. Therefore, a program's execution on the R10000 might not accurately reflect what we could expect when the same program was executed on an embedded CPU. Fortunately, recent advances in embedded computer architecture have brought processors that are both out-of-order and multi-issue, for an example see the ARM10TDMI [13].

However, speculative execution is not a feature that is necessarily desirable for embedded processors, and it is important to ignore the effects of speculation in any way possible. The MIPS R10000 has hardware event counters for both issued instructions and graduated instructions. When a branch is found, the R10000 issues several instructions on both sides of the *branch taken*/*branch not taken* code fork. Graduated instructions are the only ones who have an effect on the state of the CPU (in terms of register and memory values), and consequently issued instructions who are products of an incorrectly speculated branch would never become graduated. We can therefore measure the amount of speculation during a program run as:

$$spec\_ratio = {instr_{graduated}} \Big/ {instr_{issued}} \qquad \text{(Eqn. 3.1)}$$

As an example a program that speculates for 10 instructions and executes another 90 instructions would have a *spec_ratio* of 0.9. There are no equivalent hardware event counters for graduated loads and stores (note that there are such names for counters in Table 3.1, these counters do not have to do with speculative execution; they refer to memory accesses that are not recalled). We can therefore estimate the number of memory accesses by taking the value obtained by profiling and multiplying it by the *spec_ratio*. Assuming that the cache miss rate remains relatively constant during program execution, we can extend our use of the *spec_ratio* to cache misses and writeback requests.

Another issue with the MIPS R10000 is the secondary instruction and data cache. Normally, embedded systems would not use such a costly cache structure and consequently we would like to ignore its effects. Although it is possible to disable the L2 cache using the MIPSPro compiler, doing so often significantly lowers the performance of already slow profiling experiments. As it turns out, however, we are easily able to ignore the L2 cache by just assuming that every data or instruction reference that misses in the L1 cache automatically goes to main memory. Whether or not this actually happens in the processor makes little difference, since the behavior of the memory hierarchy is transparent to both the R10000 and our target embedded processor. Consequently we can use *perfex* to grab the appropriate cache hit/miss counts in order to simulate the performance of our embedded memory hierarchy.

3.4.2 Energy Estimation Toolflow

With an understanding of the issues involved in targeting the SGI Origin2000 machine and specifically the MIPS R10000 processor, we can now discuss how we utilize the SGI software for

our energy optimization experiments. We start by compiling a selected benchmark, usually by using a supplied Makefile, with the MIPSPro compiler. The flags passed to the compiler are dependent on what kind of optimizations we are investigating. For any of our three target memory hierarchies, we need to know the size of the data and instruction memories. Since we set the size of the instruction memory to the size of the executable running in it, we can determine our value for D_{imem} using the UNIX *ls* command. We can then determine the data memory size D_{dmem} by the size of the run-time variables by using the *dprof* command and counting the virtual pages. Alternately one can use the *dlook* command, which also performs data profiling and provides a listing of where the program data is placed in memory.

Next, we perform execution profiling using the *perfex* tool in order to measure the run-time data needed for the dynamic energy dissipation models. As an example, we can measure the number of instruction memory accesses in target architecture I by profiling hardware event 15 to obtain data for N_{read}. For the same target architecture, we can obtain data for N_{read} in the data memory by profiling hardware event 18 and multiplying by our *spec_ratio*, first calculating the speculation ratio by dividing hardware event 15 by hardware event 1. As can be seen from the previous two examples, there are several hardware event counters that need to be profiled in order to obtain all of the run-time data for the energy models. The *perfex* tool can either exactly profile two events at a time or can estimate the values for all 32 events. Since we need the data for more than two events, we decided to run *perfex* as many times as necessary to get all of the required events. Of course, there are often subtle differences in the execution of a program that will alter the value of the hardware counters from run to run; we decided that those differences would not be as large as the inaccuracies inherent in the time multiplexing of all 32 events.

In the end, we plug the run-time data and memory size data into our analytical energy equations and estimate the energy output for a given hardware configuration and code optimization.

CHAPTER 4

Loop Restructuring Optimizations for Low-Energy Software

4.1 Motivation

It is a general rule in computer architecture design and programming that in order to increase performance it is of the most benefit to make the common case fast. Following this rule, when optimizing a program for performance the best improvements will come from targeting the repetitive regions of the code; the iterative loop structures.

It is for this reason that we examine *loop restructuring* optimizations, or optimizations that reorder the execution of statements inside a loop in order to improve performance. In general, the main benefit from a loop restructuring optimization is that the corresponding data or instructions in the newly shaped loop can better fit inside the system cache. By increasing performance, these optimizations have the added benefit of also decreasing memory energy consumption in most cases. However, what is not clear is to what effect the most aggressive performance-based loop restructuring optimizations have on energy consumption.

In this chapter, we first provide an overview of the loop restructuring optimizations which we are investigating. Before analyzing these optimizations, we provide a validation of our energy estimation approach, and provide simple metrics that a power-optimizing compiler could utilize to capture the energy impact of potential optimizations. Then, we analyze the effectiveness of the selected loop restructuring optimizations on performance and energy consumption for our three target memory hierarchies. We investigate both isolated loop restructuring optimizations and suites of loop restructuring optimizations using the MIPSPro compiler. Noting that when we over-aggressively apply some of these optimizations such as loop unrolling we produce code with an undesirable tradeoff between improved performance and increased energy consumption, we investigate tailoring the aggressiveness of these optimizations, and present a loop unrolling heuristic algorithm that a compiler could leverage to improve performance while maintaining an energy consumption upper bound. Finally, we demonstrate that the effects of aggressive loop restructuring optimizations on increasing energy consumption become enhanced when considering leakage energy, especially when considering futuristic scenarios where chip fabrication technologies have advanced to the point where leakage energy contributes a substantial portion of the overall energy consumption budget. We also show that for these reasons our loop unrolling heuristic gains greater importance when considering leakage energy.

4.2 Overview of Loop Restructuring Optimizations

In this section we provide a quick introduction to several commonly utilized loop restructuring optimizations. We discuss these optimizations in terms of their potential effect on code size and performance, while providing code examples at both the C level and the assembly level. It is

important to have a thorough understanding of the low-level characteristics of these optimizations in order to develop compiler strategies to leverage these optimizations for performance gains while considering energy consumption.

4.2.1 Loop Unrolling

Loop unrolling is a commonly used optimization whereby the loop body is replaced by several copies of the loop body [29]. The loop iteration boundaries are also adjusted accordingly. The number of created copies is known as the unrolling factor. Figure 4.1 illustrates a relatively simple example of a C code transformation after unrolling by a factor of 3.

The main performance benefit of loop unrolling is the removal of the execution of many of the branches found in the loop iteration limit test code. Unrolling code also has the potential to improve the effectiveness of other optimizations such as software pipelining. For these reasons it is generally expected that applying loop unrolling will lead to both performance and energy improvements. However, the unrolled version of a code is in general larger than the original version. Consequently, loop unrolling will increase the per-access energy cost of instruction memory and may also negatively impact the effectiveness of a small instruction cache, thereby

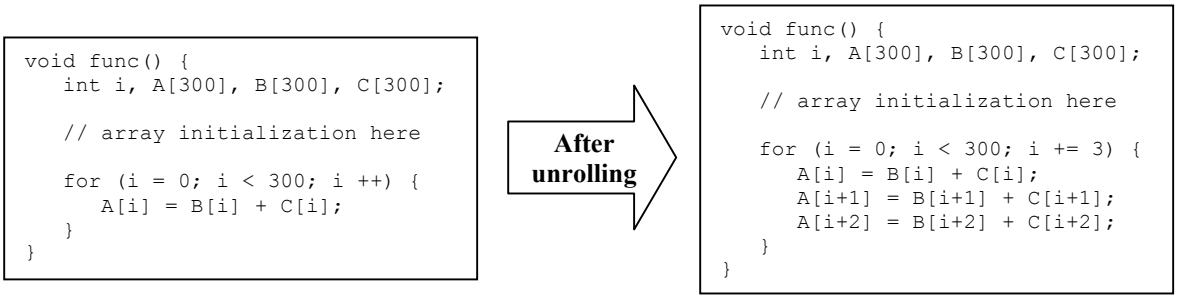

Figure 4.1. Effect of loop unrolling on sample C code

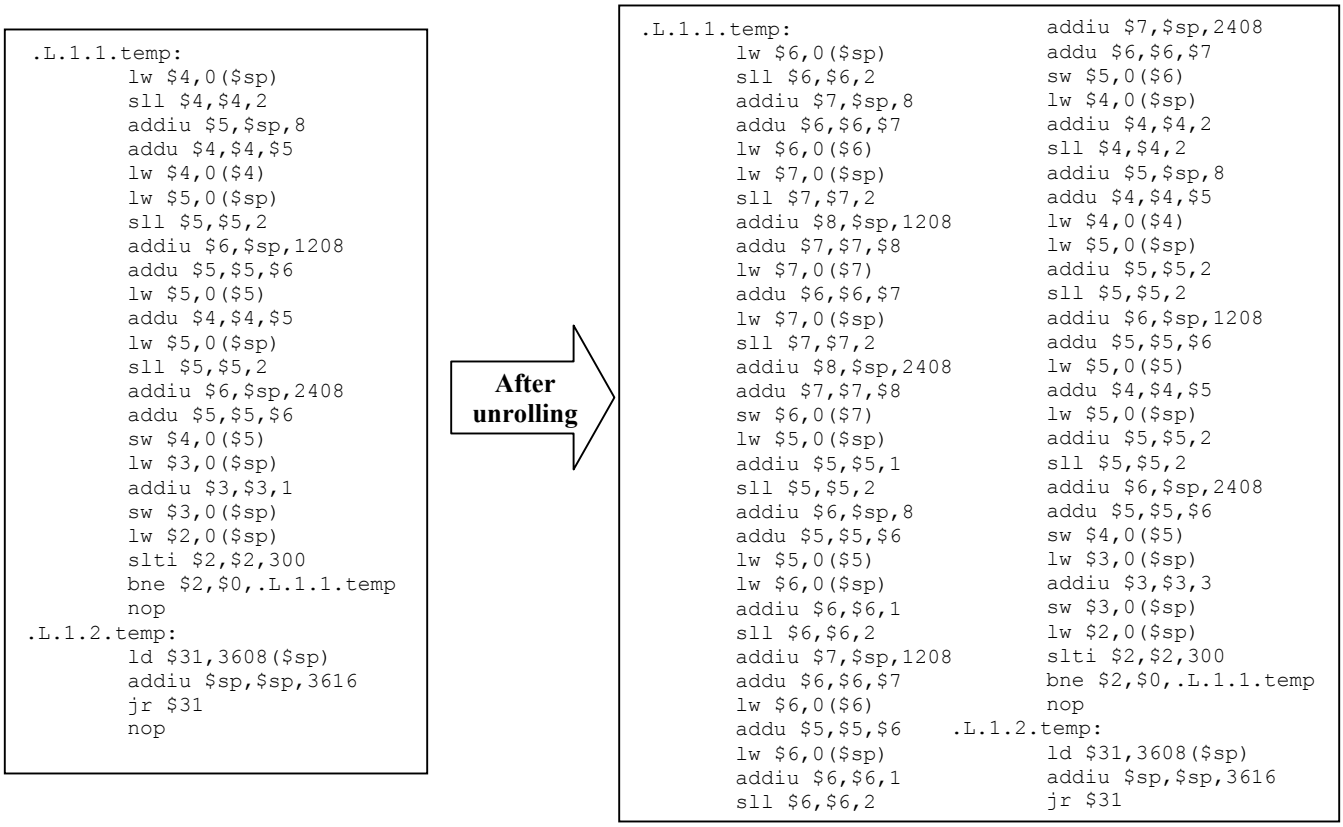

Figure 4.2. Effect of loop unrolling on equivalent MIPS IV assembly code

increasing the number of instruction memory accesses. Therefore, it is important to consider tailoring the aggressiveness of loop unrolling to limit energy increases, and we approach the subject of energy-constrained loop unrolling later in this chapter.

Figure 4.2 shows the MIPS IV assembly code that corresponds to the loop unrolling example of Figure 4.1. From our simple example we can clearly see the potential for an increase in code size (from 27 to 64 instructions) and consequently a potential increase in instruction memory energy consumption.

4.2.2 Loop Tiling

Loop tiling is an optimization where a nested loop is decomposed into a deeper nested loop in order to increase spatial locality [29]. Similar to loop unrolling, during tiling the loop iteration boundaries are adjusted so that the resultant code is equivalent to the unoptimizied version. Figure 4.3 illustrates a simple example of a C code transform after loop tiling is applied.

In general, loop tiling is most effective on arrays that are larger than data cache blocks. Tiling allows the iteration space of the nested loop to operate on subsections of arrays, lowering data cache conflict misses. For this reason, tiling is extremely effective at increasing the performance of data memory hierarchies, with the added bonus of a lowered energy consumption. Although tiling can lead to an increased code size by a nominal amount, in general the savings in data memory energy consumption outweighs the extra energy consumed in instruction memory.

4.2.3 Other Optimizations

There are several other loop restructuring optimizations that are implemented by the MIPSPro compiler, but whose performance and energy tradeoffs aren't as interesting as loop unrolling and tiling. Consequently we will not go into much detail about them and instead we just provide a brief

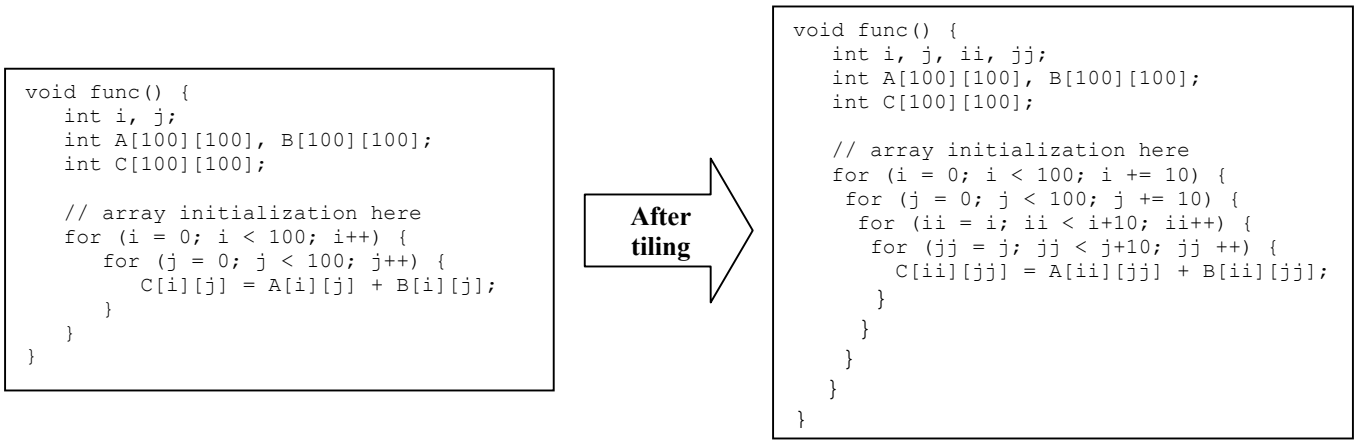

Figure 4.3. Effect of loop tiling on sample C code

overview here:

- *Loop interchange* is an optimization where the order of two adjacent nested loops is switched, the goal of which is to improve spatial and temporal locality of the program [42]. Interchanging loops can have a drastic effect on cache performance, however, applying this optimization does not affect code size. Consequently, a compiler that performs loop interchange with the goal of performance improvement in mind will be guaranteed to also decrease the overall energy consumption.

- *Loop fusion* is the process of combining two adjacent loops with the same trip count into a single loop [42]. An immediately obvious advantage of fusion is the elimination of several tests for branches. Also, fusing together two loops that reference the same data can increase temporal locality and improve data cache performance. A potential disadvantage of fusion is that the combined loop will contain more instructions, reducing instruction spatial locality and hurting instruction cache performance. In general, however, compiler heuristics intended to utilize fusion to improve performance will not make decisions that unintentionally increase instruction memory energy consumption, and subsequently sensible applications of loop fusion have no performance and energy tradeoff.

- *Loop fission*, also known as loop distribution, is the inverse operation of fusion, where a single large loop is split into two smaller loops [42]. The main advantage of loop fission is that large loop bodies that cannot fit inside small instruction caches can be broken into more manageable chunks, thereby increasing instruction memory hierarchy performance and energy consumption. A potential drawback of fission is that the new loops might require extra variables to be created, increasing the data memory size. Fortunately, a proper

compiler implementation of loop fission will not make decisions that will lead to excessive data memory growth and consequently there is little of an inherent performance and energy tradeoff.

4.3 Analyzing Energy Estimation Approach

4.3.1 Energy Model Verification

It is important before giving results on the energy consumption characteristics of compiler optimizations that we provide data confirming the accuracy of our analytical energy estimation models. In Table 4.1 we provide the absolute energy consumption values for our unoptimized benchmarks for the three different memory hierarchy configurations. For configuration III we also include absolute power values for direct comparison with other empirical data.

As can be seen from the table, many of our values for energy and power consumption are exceptionally large. Across all benchmarks chosen, our memory architectures consume 108 (J) on average and for configuration III the average power consumption is 166 (W). There is a quite apparent trend that the SPEC benchmarks, with their larger data sets and longer running times, consume a considerably larger amount of energy than their MediaBench counterparts. The SPEC benchmarks consume 161 (J) on average (across all the architectures), while the MediaBench benchmarks consume only 3.6 (J).

These results, although shocking at first, make perfect sense when considering our energy model. Although having an on-chip memory is not unusual in SoC devices, they are often on the order of kilobytes. Several of our SPEC benchmarks are more workstation-based, and consequently

Benchmark	Energy Consumption Configuration I	Energy Consumption Configuration II	Energy Consumption Configuration III	Power Dissipation Configuration III
art	171 (J)	14.2 (J)	8.69 (J)	734 (W)
equake	116 (J)	2.42 (J)	1.29 (J)	13.2 (W)
mesa	184 (J)	53.5 (J)	764 (mJ)	12.0 (W)
gzip	169 (J)	10.4 (J)	2.12 (J)	51.8 (W)
vortex	1,940 (J)	207 (J)	9.28 (J)	680 (W)
twolf	3.42 (J)	2.97 (J)	80.5 (mJ)	228 (mW)
mpeg2decode	1.25 (J)	619 (mJ)	54.3 (mJ)	88.3 (mW)
mpeg2encode	23.6 (J)	6.06 (J)	465 (mJ)	5.93 (W)
rawcaudio	7.22 (mJ)	4.35 (mJ)	2.75 (mJ)	.258 (mW)

Table 4.1. Energy and power consumption values for our analytic models.

our energy models for many of these applications require on-chip memories on the order of 10 megabytes or more. Clearly this skews our energy / power values. The MediaBench benchmarks, on the other hand, with their manageable memory size, give power consumption values that are on the same order of magnitude as the empirical data collected by Chandrakasan et. al. in their JouleTrack project [37].

Despite the infeasibility of many of our energy results, we feel that our energy models are still appropriate for performing relative calculations. Consequently, for the remainder of this paper, we analyze the effect of optimizations on memory energy consumption on a normalized scale, with only a passing glance at the absolute values.

4.3.2 Simple Metrics for Energy Estimation

In general, the per-access energy cost is directly related to the memory size, which in embedded devices is determined by the number of bytes required to store code / data. Also, the total number of instructions executed is an accurate measure of the number of times that an instruction memory without caching would need to be accessed. For this reason, we explored using the product of the code size and the instruction count as an early estimate to how much energy a given benchmark would be consuming in instruction memory.

Figure 4.4 depicts the effects of the MIPSPro optimization modes on this metric and compares the observed trends with those obtained through actual energy calculations for the instruction memory hierarchy of our three memory configurations. As can be seen from the figure, our energy consumption estimation metric shows similar trends when compared to the calculated values for our three configurations. The normalized metric is within 1.3% of the normalized estimated instruction memory energy values on average across all of our benchmarks and configurations. The metric is most accurate in the configurations without instruction caches (configurations I and II), where it is within .01% of the calculated energy values. When adding the

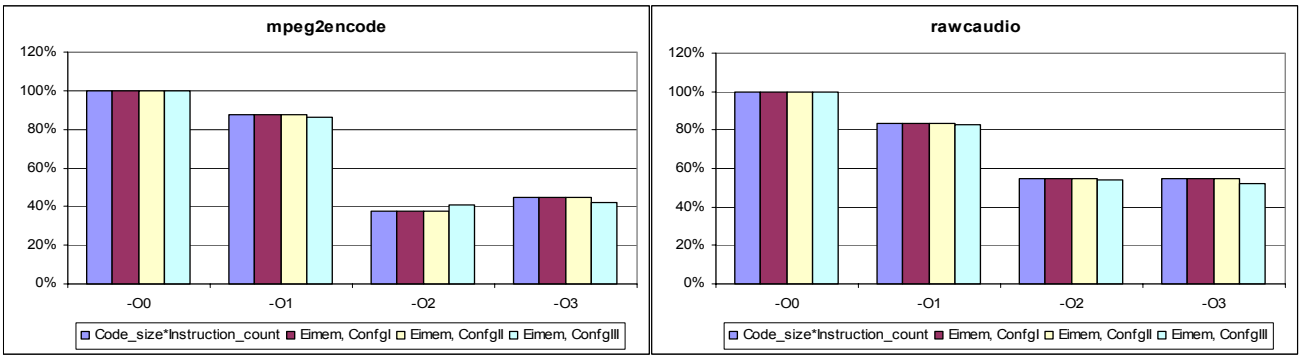

Figure 4.4. Normalized *Code_size*Instruction_count* compared to normalized estimated instruction memory energy consumption for the three target memory configurations. These results show that the metric is a fairly accurate energy estimate, being within 1.3% on average.

instruction cache for configuration III the average error jumps to 4%.

Similar to the instruction memory, for the data memory we can take the number of accesses to be the total number of read and write requests. We can explore a similar metric of the product of the data size and the read and write requests as an early estimate to how much energy a given benchmark would be dissipating in data memory. Figure 4.5 depicts the effects of the MIPSPro optimization modes on this metric and compares the observed trends with those obtained through actual energy calculations for our three memory configurations. The results for this metric are not as promising as those for the instruction memory energy estimation metric. Across all configurations, the data memory metric is within 10% of the normalized estimated data memory energy values on average. The metric is most accurate in the configuration without a data cache (configuration I), where it is within 1.3% of the calculated energy values. When adding the data cache for configurations II and III the average error jumps to 19%. However, even in these cases, the relationship between values of the metric corresponds to the trends seen in the estimated energy values.

Based on these observations, we can conclude that a compiler optimization technique that

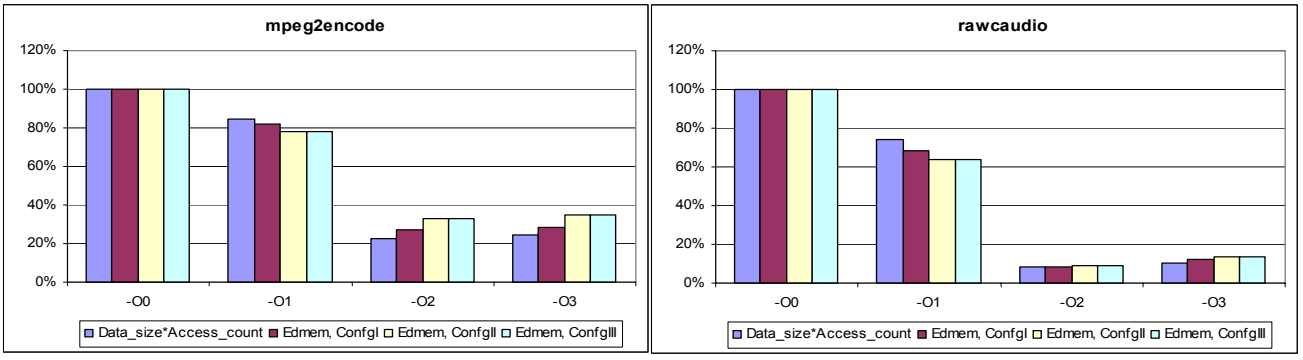

Figure 4.5. Normalized *Data_size*Access_count* compared to normalized estimated data memory energy consumption for the three target memory configurations. These results show that the metric is not as accurate as the instruction memory energy metric, on average being within 10%. In general, the trends among optimizations for the estimation metric hold true for the calculated energy values.

minimizes the metric of *Code_size* * *Instruction_count* will also minimize the instruction memory energy consumption in most cases. That is to say, the *Code_size* * *Instruction_count* metric can be utilized to rank the instruction memory energy consumptions of different optimized versions of a given code. The same statement can be made with lesser confidence about the metric of *Data_size* * *(Data_read + Data_write)* and data memory energy consumption. This is an important conclusion as it indicates that, instead of using complex energy calculations, a compiler can adopt an energy estimation strategy based on the estimation of dynamic instruction count, data access count, executable size, and data variable size. Previous work [41] shows that accurately estimating static and dynamic instruction count at compile time is possible even for sophisticated superscalar processors such as the MIPS R10000. Similar techniques could be developed to also estimate the data access count by either analyzing instruction types or assuming a constant ratio (30% is a common choice) of data accesses to total instructions. Therefore, such estimates can be used for obtaining an idea about instruction and data memory energy consumption of a given code under a set of optimizations.

4.4 Analyzing Loop Restructuring Optimization Modes

In this section we analyze the loop restructuring optimizing modes of the MIPSPro compiler. As was first mentioned in Chapter 3, there are four major modes of the MIPSPro that perform different performance optimizations. The –O0 mode is the base case, where no optimizations are performed. The aggressiveness of optimizations increases in the modes from –O1 to –O2, where in the –O2 mode we first see loop unrolling performed. The –O3 is the most aggressive optimizing mode, where the Loop Nest Optimizer (LNO) is activated that attempts many of the loop transformation

optimizations mentioned previously, such as loop tiling, fission/fusion, and loop interchange. These main optimization modes attempt to optimize performance with no restrictions on the growth of generated executables. For this reason it is of interest for us to examine the tradeoffs between code size and performance demonstrated by these loop restructuring optimizations, and to analyze the effect on energy consumption for our three target memory hierarchies.

4.4.1 Code Size/Performance Tradeoffs

Figure 4.6 shows the normalized code size and graduated instruction count for six of our benchmarks when compiled with the major loop transforming modes of the MIPSPro compiler. From these results, we can observe several trends. Analyzing performance, we can note that each optimization mode from –O1 to –O2 shows (in general) a smaller dynamic instruction count. On average, compiling with the –O1 mode leads to a 9% performance improvement while using the –O2 mode shows a 49% performance improvement across all benchmarks used. The MIPSPro compiler appears to be more successful at optimizing smaller applications, an example being the *mpeg2decode* benchmark that shows a 76% decrease in instruction count when the –O2 mode is

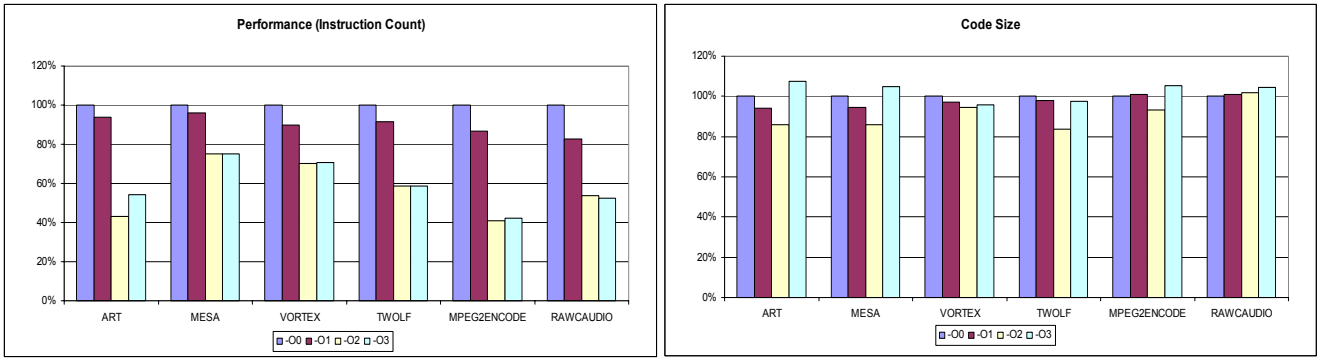

Figure 4.6. Performance and code size for the main MIPSPro optimization modes. These results show that the most aggressive mode (-O3) leads to a 10% code size increase over the –O2 mode, with only a negligible performance benefit.

used. As a contrast, the relatively large *mesa* benchmark shows only a 25% performance improvement in this mode.

Analyzing code size, it is clear that on top of improving performance, each optimization mode form –O1 to –O2 demonstrates a smaller code size. This is due to the fact that these levels perform many optimizations that either remove unnecessary code or optimize for performance without adding code. On average, compiling with the –O1 mode leads to a 2% code size decrease while using the –O2 mode shows a 10% decrease across all benchmarks used. It is clear that in lowering the instruction count while decreasing the code size, the –O1 and –O2 optimization modes will decrease the instruction memory energy consumption in our target embedded system.

At the –O3 optimization level, the loop nest optimizer performs more aggressive loop unrolling along with other trade-offing optimizations, and the results are mixed. For the benchmark codes in our experimental suite, running the LNO at the –O3 level leads to on average a 1% performance decrease over the –O2 level, at the cost of a 10% increase in code size. This average lowered performance is misleading however, as the benchmarks that have regular loop structures are able to take advantage of the LNO towards performance gains. An example is the *rawcaudio* benchmark, where using the –O3 mode shows a 2% performance improvement when compared to the –O2 mode, for an overall improvement of 48%. It is quite clear when using our instruction energy consumption estimation metric that these small performance gains at the cost of a relatively large code size increase will demonstrate a performance / energy tradeoff.

4.4.2 Energy Consumption – Configuration I

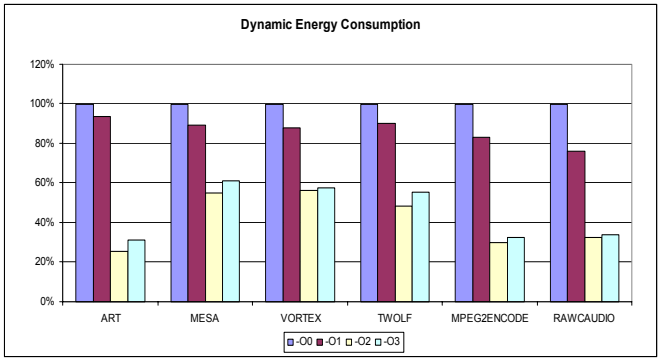

Figure 4.7. Total energy consumption of the main optimization modes for configuration I. These results show that due to the code size increases of the –O3 mode, the total energy consumption is also increased.

The lack of any caches in configuration I means that optimizations that target data cache performance will have little positive effect. This can be seen in Figure 4.7, where the optimization modes that are able to decrease code size (–O1 and –O2) are also able to decrease energy consumption. Across all benchmarks used, the –O1 optimization mode shows a 13% energy consumption decrease on average while the –O2 mode shows an impressive 63% decrease on average. The smaller benchmarks (in terms of unoptimized executable size) show a larger energy benefit from the –O2 mode. An example of this are the MediaBench benchmarks, which show a 72% energy consumption decrease with the –O2 mode. This is in contrast to the three largest benchmarks *mesa*, *gzip,* and *twolf*, which show only a 54% average energy decrease at the same optimization mode.

Due to the fact that the increase in code size is not tempered by any decrease in cache miss rates in configuration I, the –O3 mode demonstrates an energy increase when compared to the –O2 mode, although it should be noted that this most aggressive optimization mode still consumes less power than when no optimizations are applied at –O0. On average compiling with the –O3 mode leads to a 3% energy consumption increase when compared to the –O2 mode. This trend is not

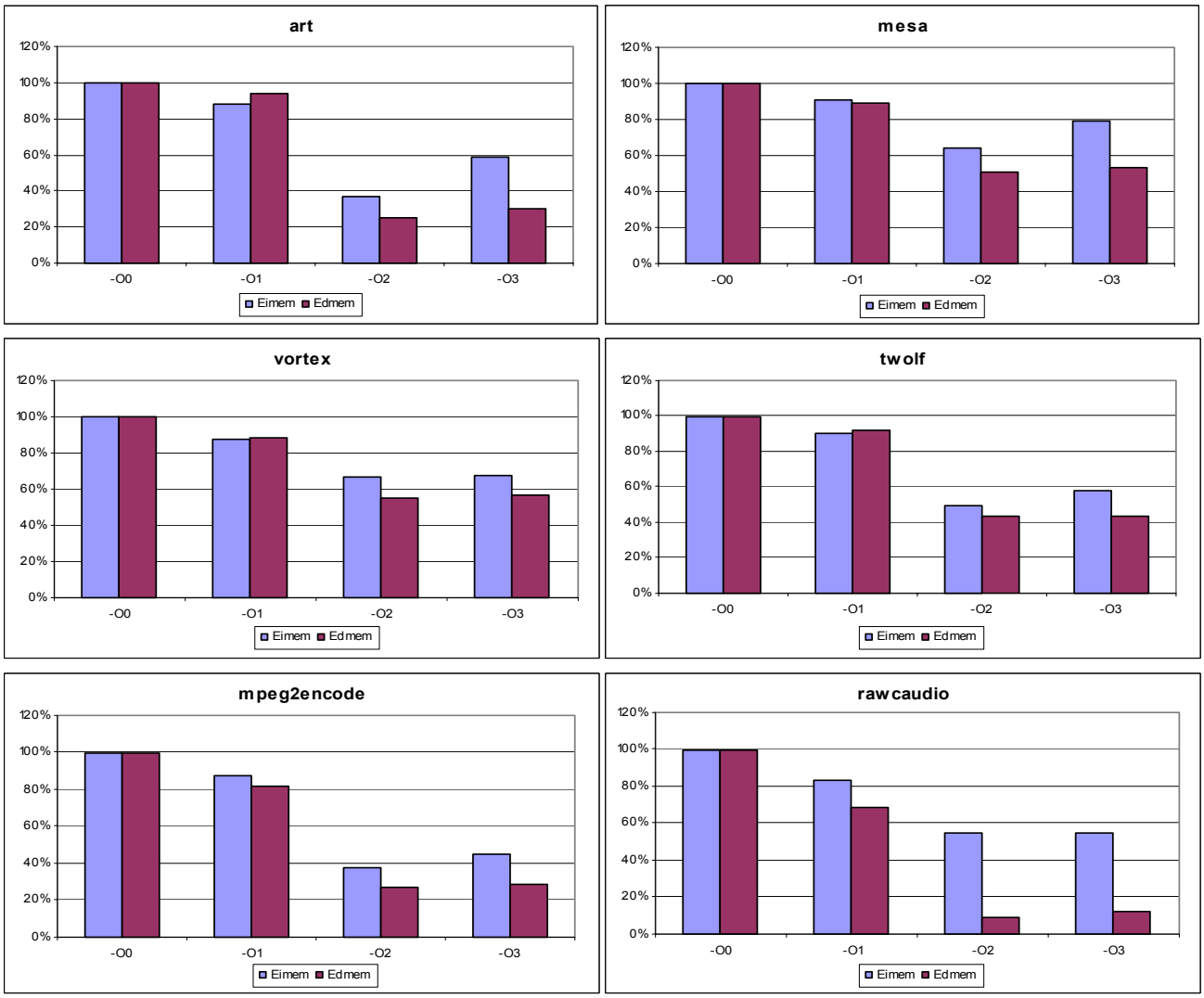

Figure 4.8. Optimization mode energy breakdown by component for configuration I. These results show that due to code size increases, the most aggressive optimization mode (-O3) increases the instruction memory energy consumption by a significant factor.

followed across all benchmarks, however, an example being the *mpeg2decode* benchmark that actually shows a 2% energy consumption decrease at the –O3 level.

Figure 4.8 shows the energy breakdown by component for configuration I. As was to be expected after analyzing the code size data, the optimization modes from –O1 to –O2 show a gradual decrease in instruction memory energy consumption when compared to the –O0 mode. The results from the –O3 are more interesting. When comparing the –O2 mode to the –O3 mode, it can be seen

that the improvements in data memory energy consumption are roughly the same (within 1%). It is in the instruction memory, however, where the –O3 mode shows a 7% energy consumption increase when compared to the –O2 mode. This is clearly an example of how a compiler that concentrates only on performance can generate code that demonstrates only slight speedups with a side effect of an increase in energy consumption due to a larger code size. It is for these reasons that later in this chapter we investigate tailoring the aggressiveness of one of these optimizations (loop unrolling) in order to gain the desired performance improvement while keeping energy consumption in mind.

4.4.3 Energy Consumption – Configuration II

Adding the data cache for configuration II significantly lowers the overall memory energy consumption, and consequently the loop restructuring optimizations have less of an effect. Figure 4.9 shows that the –O1 optimization mode now demonstrates a 11% energy reduction on average, as compared to 13% for configuration I. More pronounced is the difference for the –O2 mode, where the 63% energy consumption reduction in configuration I is now a 49% reduction in configuration II. Again as before the smaller MediaBench benchmarks show a more pronounced energy savings in the

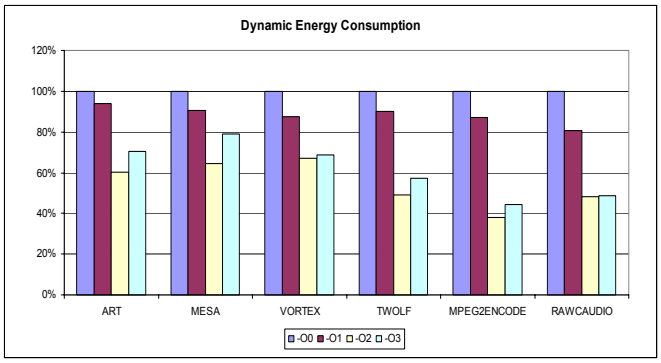

Figure 4.9. Total energy consumption of the main optimization modes for configuration II. These results show that since the data cache lowers the overall cache energy consumption, the loop optimizations are less effective at decreasing energy consumption.

–O2 mode, on average there being a 63% reduction.

Similar to the previous configuration, using the –O3 mode leads to an energy increase when compared to the –O2 mode, the average being around 5%. This number is misleading, however, in that the longer running SPEC benchmarks show a much larger 8% energy increase while the increase in the MediaBench benchmarks is only about 1%. Certain benchmarks, where the increase in code size of the –O3 mode far outweighs the performance benefit show a double-digit increase in

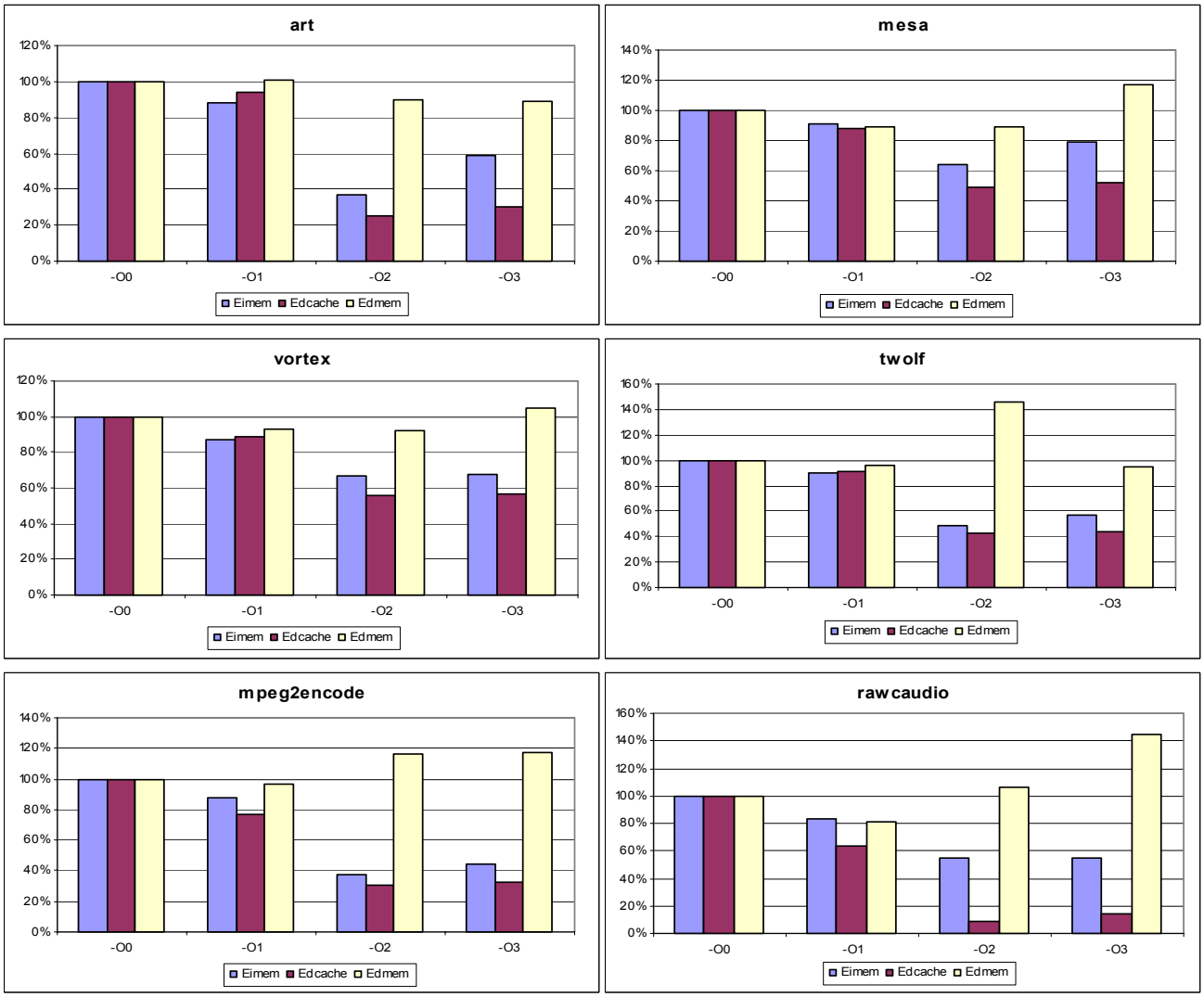

Figure 4.10. Optimization mode energy breakdown by component for configuration II. These results show that the loop restructuring optimizations decrease data cache energy consumption, with occasional increases in data memory energy consumption.

energy consumption: for the *mesa* benchmark there is a 15% energy consumption increase.

Figure 4.10 shows the energy breakdown by component for the main optimization modes for configuration II. The instruction memory energy consumption shows similar trends to the code size values from Figure 4.6. It is in the data memory hierarchy where the results become interesting. One noticeable trend is that while the –O1 optimization mode shows a decrease in data cache energy consumption (15%), it is when the loop nest optimizations are turned on in the –O2 and –O3 modes that there is a significant decrease. On average the –O2 mode demonstrates a 67% data cache energy consumption decrease; while for the –O3 mode the decrease is 65%. These results clearly show that while optimizing for slight performance gains the aggressive loop transformations of the –O3 mode do not decrease even data cache energy consumption.

When looking at Figure 4.10 it is apparent that there are several spikes in the data memory energy consumption occurring at different optimization levels. An example is the *twolf* benchmark, which shows a 50% data memory energy consumption increase between the –O1 and –O2 optimization levels. This is due to the fact that for certain code structures, applying loop optimizations such as tiling can actually lead to an increase in data cache misses and writeback requests.

4.4.4 Energy Consumption – Configuration III

In configuration III we add an instruction cache to the memory hierarchy of configuration II, and on average the MIPSPro optimization modes reduce energy by even less than before. Figure 4.11 shows the total energy consumption for six of our benchmarks as a function of the optimization mode. Since adding an instruction cache lessens the effect of shrinking the code size on decreasing

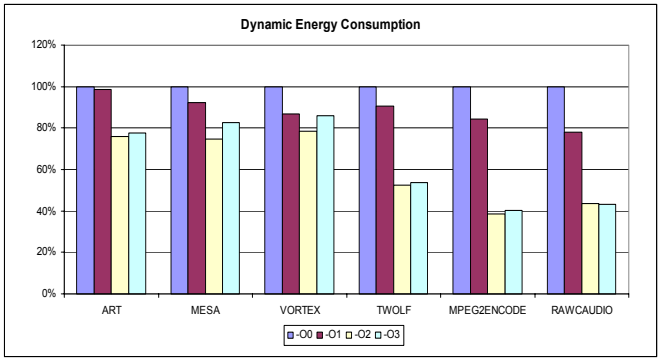

Figure 4.11. Total energy consumption of the main optimization modes for configuration III. These results show that adding the instruction cache lessens the effect that an increased code size has on energy consumption.

instruction memory energy consumption, the two optimization modes (-O1 and –O2) that decrease code size don't decrease energy consumption as much as in configuration II. The –O1 mode decreases the total energy consumption by 10%, as opposed to 11% in configuration II, and the –O2 mode decreases energy by 40%, as opposed to 49% before. This phenomenon does not extend to the smaller MediaBench benchmarks that did not demonstrate much of a code size decrease to begin with. On average the MediaBench benchmarks show a 64% energy consumption decrease in the –O2 mode, comparable to the decrease demonstrated in configuration II.

The –O3 mode leads to less of an energy increase when compared to the –O2 mode in configuration III, the average being around 2%. This is in direct comparison to the 5% increase of configuration II. It is clear that as our target embedded architecture begins to approach the MIPS R10000, the most aggressive loop nest optimizations of the MIPSPro compiler have less of a negative impact on energy consumption. Again as before, the –O3 mode is less devastating to the energy consumption of the MediaBench benchmarks, where in this configuration there is actually a 1% energy decrease with the –O3 mode.

Figure 4.12 shows the energy breakdown by component for the main optimization modes for configuration III. Since the data side of the memory hierarchy is the same as in configuration II, we

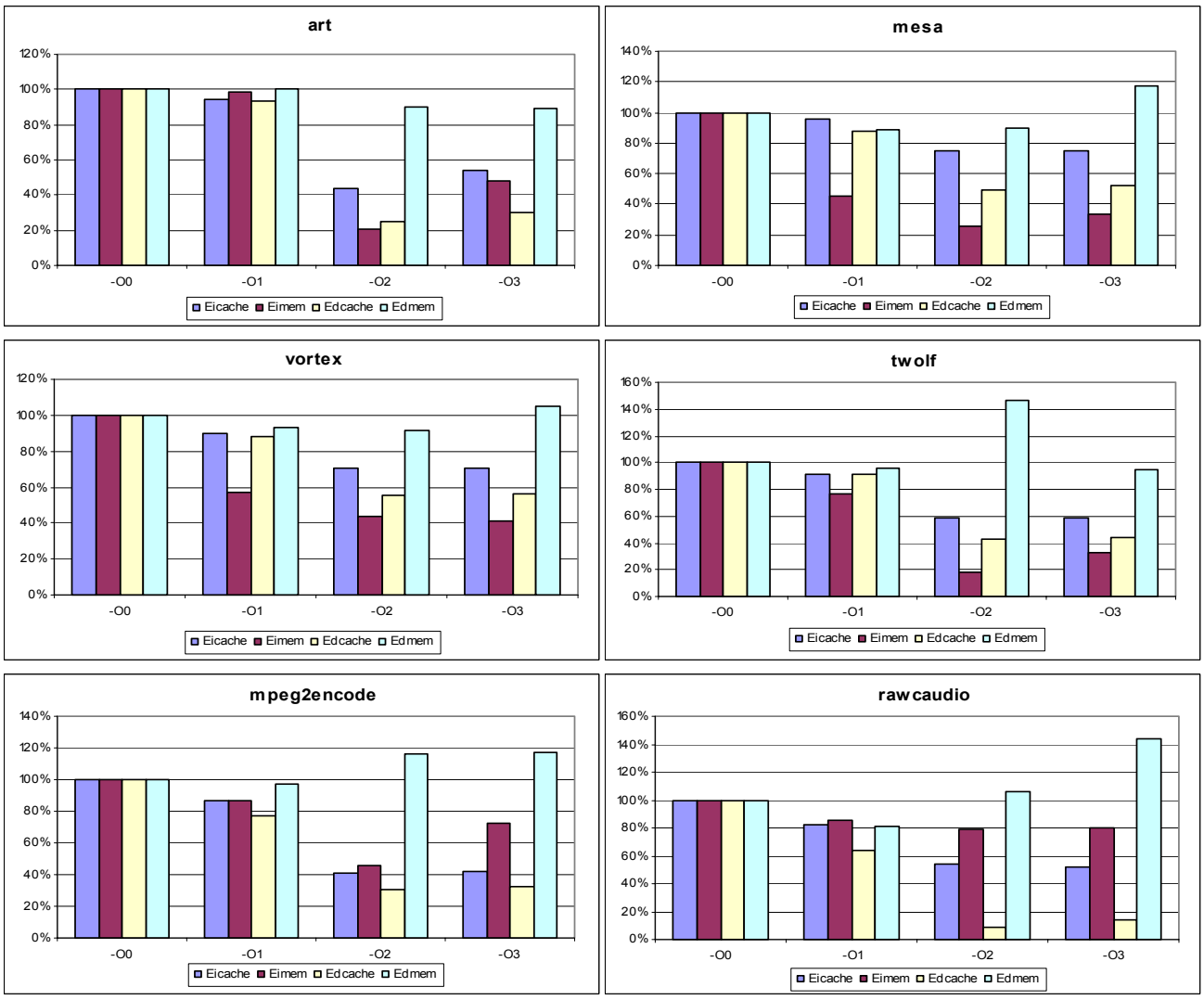

Figure 4.12. Optimization mode energy breakdown by component for configuration III. These results show that the most aggressive loop restructuring optimizations tend to increase code size to the point where the instruction memory energy consumption increases, even when there is an instruction cache involved.

will only analyze the energy effects on the instruction cache and memory. The effect of the optimization modes on instruction cache energy consumption varies wildly from benchmark to benchmark. For the –O2 optimization mode the instruction cache energy consumption is lowered by 49% on average. However, for our three largest benchmarks that number drops to 32%. For the MediaBench benchmarks the –O2 optimization mode is very effective at resizing the repetitive sections of code to fit inside the cache, and consequently for these benchmarks the –O2 mode

demonstrates a 60% instruction cache energy consumption decrease. Also, even though the –O3 mode leads to a 10% code size increase as compared to the –O2 mode, there is only a 1% instruction cache energy consumption increase, and in many of the benchmarks the –O3 mode leads to a decrease in instruction cache energy consumption.

Looking at the instruction memory, we can see that in general, the instruction memory energy consumption follows the trends in code size seen in Figure 4.6, with some exceptions. On average compiling with the –O1 mode leads to an 18% decrease in instruction memory energy consumption. However, for certain benchmarks this decrease is quite larger, an example being the *mesa* benchmark that shows a 54% decrease in instruction memory energy in the –O1 mode. Compiling with the –O2 mode leads to a 62% decrease in instruction memory energy consumption, which is much larger than the decrease seen in the other configurations. The –O3 mode leads to a 13% increase when compared to the –O2 mode, which is much larger than the 7% increase seen when the instruction cache is removed in configurations I and II. Note however that the normalized instruction memory energy of the –O3 mode is about the same between all three configurations, which implies that it is mostly in the –O2 case where the generated code fits better inside the cache.

4.5 Tailoring Unrolling to Energy Requirements

Since overly aggressive loop restructuring optimizations can lead to an undesirable tradeoff between energy consumption and performance, it is of interest to investigate tailoring these optimizations to fit to energy constraints. Earlier in this chapter we identified loop unrolling as an optimization that can be used to improve performance but when used too aggressively can increase code size and potentially energy consumption. In this section we present and analyze a heuristic that provides a

systematic way of choosing a suitable loop unrolling strategy that attempts to improve performance while keeping in mind energy consumption.

4.5.1 Unrolling Heuristic

Figure 4.13 shows our energy-aware unrolling heuristic. Our approach to the problem is as follows. We start with an unoptimized program and a set of loops inside the function that we are interested in unrolling. At each step, we choose the first loop that has not been unrolled yet by a factor of n, and then unroll it by that amount. After the unrolling, we estimate the resultant energy consumption and compare it with the upper bound. If the upper bound has not been reached, we select the next loop to

```
ENERGY_UNROLL( C, E_limit ) {
        C_new = C;
        unroll_factor = 1;
        repeat {
                C_old = C_new;
                l_unroll = Ø;
                L = loop_list( C_old );
                for each loop l ∈ L do {
                        if ( loop_size(l) < unroll_factor) then {
                                l_unroll = l;
                                break;
                        }
                }
                if ( l_unroll ≠ Ø ) then {
                        C_new = perform_unrolling( C_old, l_unroll, unroll_factor );
                        E = estimate_energy( C_new );
                }
                else {
                        unroll_factor = unroll_factor * 2;
                }
        }
        until ( E > E_limit );
        return( C_old );
}
```

unroll. If all the desirable loops have already been unrolled by a factor of *n* or more, we increment *n* and repeat the process. Once the energy consumption after an unrolling becomes larger than the upper bound, we undo the last optimization and return the resulting code as the output. Note that our heuristic is not specific about how the most appropriate loops are chosen. In practice, the SGI profiling tool SpeedShop can be used to identify the functions that contain loops where aggressive optimization would make sense. Also, in our heuristic we multiply the unrolling factor by two each iteration. In practice it might make more sense to just increment the unrolling factor by 1 to obtain more fine-grained results.

4.5.2 Results

For our experiments we selected the *mesa* and *mpeg2decode* benchmarks since they are on opposite ends of the spectrum in terms of unoptimized code size and run times and in Section 4.4 we demonstrated that they are examples of applications with performance and energy tradeoffs under loop transforming optimizations. We chose to run our unrolling heuristic on configuration II, since its lack of instruction cache highlights the energy increasing effects of aggressively optimizing for performance at the expense of code size.

Figure 4.14 shows the performance (in terms of graduated instruction count) for our selected benchmarks when our unrolling heuristic is applied under different memory energy upper bounds. The benchmarks are compiled at the –O2 mode with unrolling turned off as the base case. It can be observed that we achieve a performance improvement of 14% on the average across all energy bounds used. It can also be seen that as we increase our energy upper bound, our loop unrolling strategy produces better-performing code, demonstrating the tradeoff between energy and

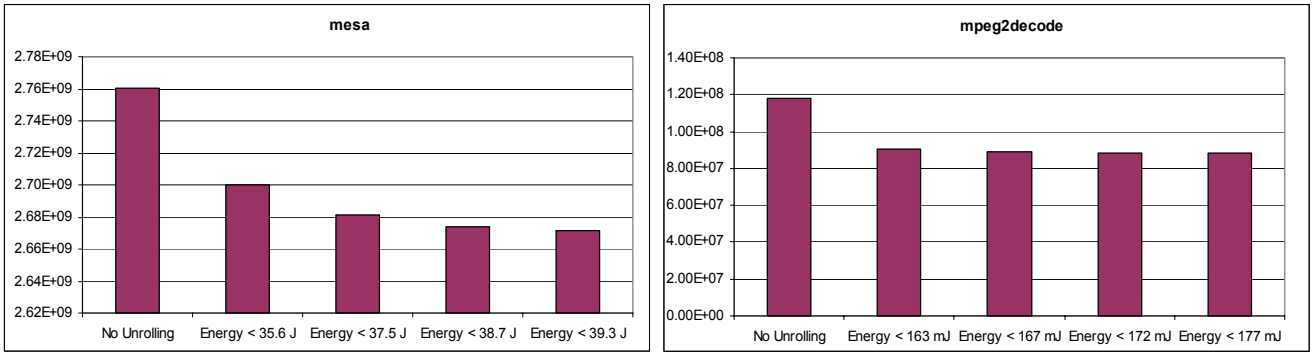

Figure 4.14. Performance in terms of instruction count as a function of the energy upper bound of our unrolling heuristic algorithm. These results show that our heuristic leverages similar performance via unrolling as the more aggressive strategies, while consuming far less energy.

performance. As seen in the second bar in each subgraph of Figure 4.14, in the most restrictive case our heuristic algorithm is able to provide a 10% energy consumption savings while keeping performance to within 2% of the more aggressive unrolling strategies. These results indicate that our heuristic improves performance while keeping the energy consumption below a pre-set limit.

4.6 Considering Leakage Energy

As was mentioned in Chapter 2, the energy consumption in CMOS circuits is divided into both dynamic and static components. The dynamic, or switching energy consumption comes from the bit transitions on individual wires, and is only a factor in an active component. On the contrary, the static, or leakage energy consumption comes from the leakage current inherent in transistor technology and is dissipated independent of the state of a component. That is to say, leakage energy is always consumed in a circuit, while switching energy is only to be considered during active states. Up to this point we have been only analyzing the loop restructuring optimizations in terms of dynamic energy consumption, since in current design technologies it is the dominant part. However, current trends indicate that the contribution of leakage energy to the overall energy budget will

increase exponentially in upcoming circuit generations. Leakage energy is especially important in the large SRAM memories that we are studying, since the leakage consumption is a function of the memory size.

In this section, we first show the impact of taking leakage into account on the tradeoff between memory energy and performance of the loop restructuring optimization modes of the MIPSPro compiler. After that, we show how our energy-sensitive compilation approach in the previous section performs when leakage is accounted for. We concentrate on the relative weight of leakage energy to dynamic energy by using the approximation developed earlier, which is to take the per-cycle leakage energy consumption to be a ratio of the per-access dynamic energy consumption. That ratio can be represented as a nonnegative value k where a smaller k value ($0.1 <= k <= 0.2$) represents current fabrication technologies and larger k values ($0.5 < k <= 1.0$) represent a futuristic scenario where the effect of leakage energy will begin to outpace that of switching energy. Note that our approximations for the total energy consumption can return to the values for dynamic energy consumption by just setting $k = 0$.

Figure 4.15 shows our results when we compile six of our benchmarks using the main optimization modes of the MIPSPro compiler. These results detail the actual values for energy consumption for configuration II when we take the leakage energy of the instruction memory into account. These results show an increasingly pronounced tradeoff between performance and energy when compared to the equivalent results of Section 4.4. As an example, when leakage energy is not taken into consideration, the *art* benchmark shows an average energy increase of just under 1.5 (J) when adding the aggressive loop nest optimizations to advance between the –O2 and the –O3 optimization modes of the MIPSPro compiler. As we increase the value of k to represent the increase in the relative weight of the leakage energy to the total energy equation, we see this number

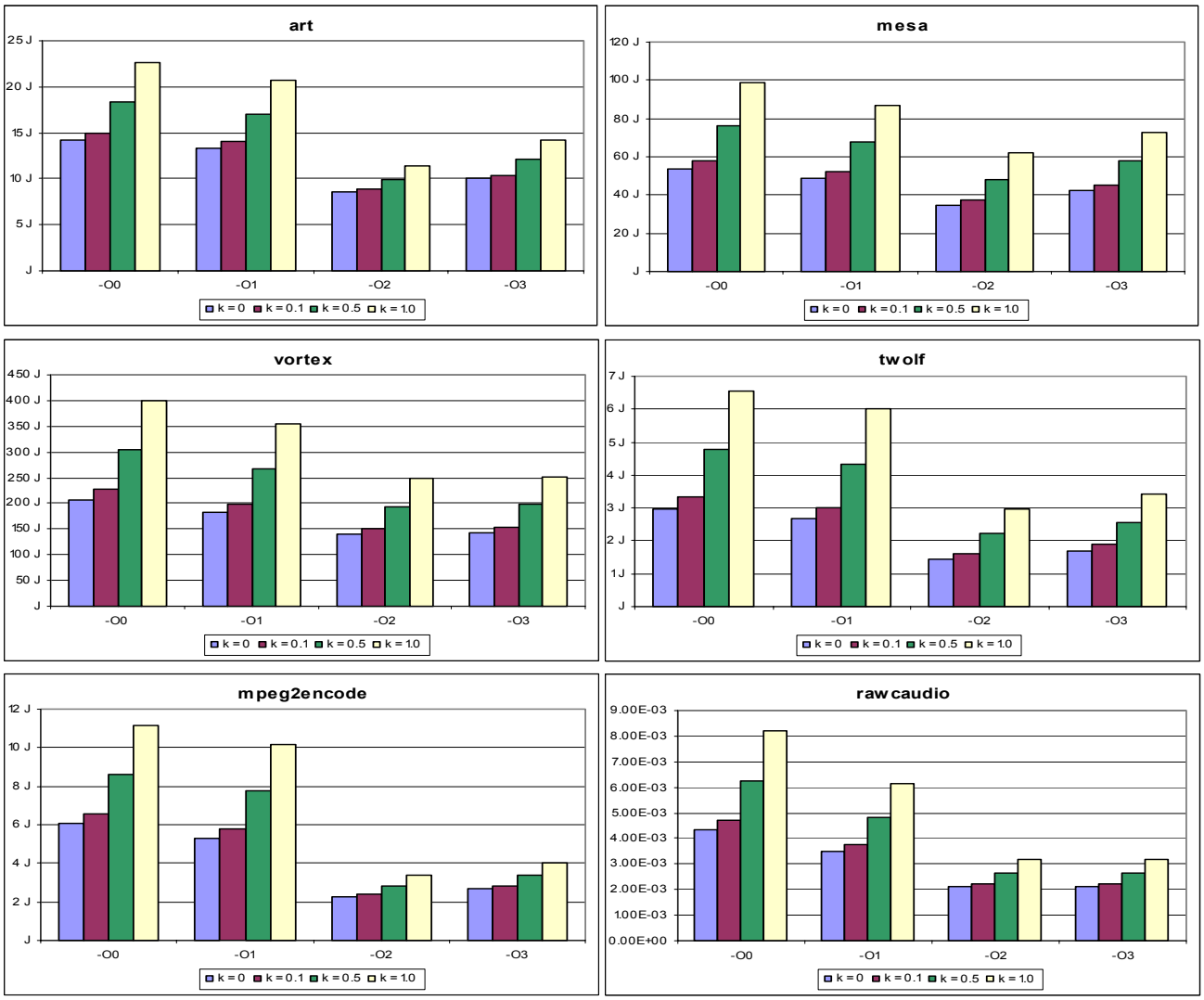

Figure 4.15. Total energy consumption (as the sum of both dynamic and static consumption) for different loop restructuring optimization modes. The value of *k* refers to the relative weight of the per-cycle leakage energy to the per-access dynamic energy. These results show that as we increase *k*, the –O3 mode increases energy consumption when compared to the –O2 mode by an even greater factor.

steadily grow. When $k = 0.1$, our energy increase between –O2 and –O3 is 1.6 (J), when $k = 0.5$ the increase is 1.8 (J), and when we consider the weight of leakage energy to be equal to that of dynamic energy at $k = 1.0$, the energy consumption increase is now 2.1 (J). These results demonstrate that as leakage energy becomes more of an issue in future design methodologies, performance

optimizations that have a side effect of an increased code size will have a greater effect on increasing energy consumption.

Figure 4.16 shows our results when we apply our loop unrolling algorithm to the *mesa* and *mpeg2decode* benchmarks with an energy upper bound that takes leakage energy into account. For this experiment, we plotted the minimum energy upper bound that would be required for certain performance improvements. The results in Figure 4.16 show that as we increase *k*, we require a greater upper energy bound to achieve the same performance via unrolling. As an example, for the *mesa* benchmark, to achieve a performance improvement of 3% (as represented by the middle bar in each subgraph in the top row of Figure 4.16), there is an approximately 23 (J) difference between energy upper bounds when k = 0.1 and k = 1.0. Clearly, as leakage energy becomes more of a factor in future design fabrication methodologies, it will become increasingly important to limit code and energy growth at the expense of performance gains.

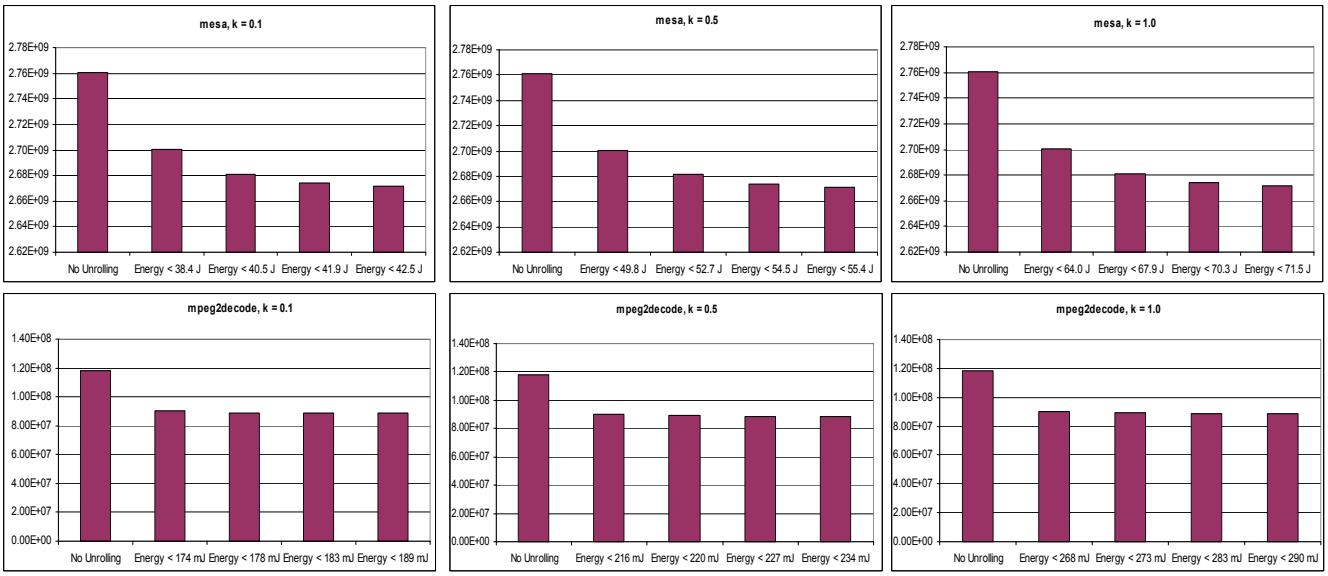

Figure 4.16. Performance versus memory energy upper bound of our unrolling heuristic algorithm when including leakage energy for different values of *k*. These results show that as we increase *k*, we require a greater upper energy bound to achieve the same performance via unrolling.

4.7 Summary

In this chapter we have analyzed the energy consumption characteristics of some standard loop restructuring optimizations. In general, these optimizations can be used effectively to leverage performance gains while also reducing energy consumption. We demonstrated that the –O2 optimization mode of the MIPSPro compiler, which is an example of a suite of non-aggressive loop transformations, can be used to improve performance by an average of 49% while decreasing energy consumption by 40% for memory configuration III. There is certainly a danger of over-applying these optimizations, since the potential for performance improvement often has a strict limit while the potential for energy increases is boundless. As an example, we showed that the most aggressive loop transforming mode of the MIPSPro compiler, the –O3 mode, demonstrates only a negligible 1% performance improvement over the –O2 mode, while at the same time it increases the average memory energy consumption by 13% for configuration III. We also showed that this increase in energy consumption due to code growth will only worsen in the future when leakage energy becomes more dominant.

In order to protect against the energy increasing effects of these most aggressive optimizations, we demonstrated that a simple compiler heuristic can be leveraged to improve performance via loop unrolling while keeping energy consumption under a predetermined bound. Using this heuristic, we were able to improve energy consumption by 10% while keeping the performance within 2% of the more aggressive loop unrolling strategies. We have also shown that very simple run-time metrics can be used be a power-aware compiler as an early estimate of energy consumption, so that low-level equations do not need to be used. It is reasonable to assume that a

compiler that implements loop-transformation heuristics similar to ours will be able to generate more energy-efficient code while achieving similar performance when compared to a standard optimizing compiler such as the MIPSPro.

CHAPTER 5

Interprocedural Optimizations for Low-Energy Software

5.1 Motivation

In the previous chapter we were mainly concerned with *intraprocedural* optimizations, or optimizations that focus on a single function at a time. While extremely useful, intraprocedural optimizations can be severely crippled by certain code organizations, for example a loop containing a procedure call would be optimized without any knowledge of the behavior of the called function.

It is for this reason that we now focus on *interprocedural* optimizations, which can improve application performance while also enhancing the ability of the compiler to perform intraprocedural optimizations. Most of the interprocedural optimizations that have been developed involve moving code between function calls or removing function calls entirely. Like the loop transformation optimizations, interprocedural optimizations can be utilized to improve energy consumption, since they can decrease the code overhead associated with function calls while allowing for global optimizations to be applied. However, similar to the loop nest optimizations, applying

interprocedural optimizations too aggressively can have an adverse affect on energy consumption by increasing code size and consequently requiring the system to have a larger instruction memory.

In this chapter we analyze the effectiveness of interprocedural optimizations on performance and energy consumption for our three target memory hierarchies. We investigate both isolated interprocedural optimizations and suites of interprocedural optimizations using the MIPSPro compiler. Noting that heavily applied interprocedural optimizations such as function inlining can lead to an undesirable tradeoff between improved performance and increased energy consumption, we investigate tailoring the aggressiveness of these optimizations, and present a function inlining heuristic that a compiler could leverage to improve performance while maintaining an energy consumption upper bound. Finally, we demonstrate that our inlining heuristic gains greater importance when leakage energy is considered, especially when simulating future scenarios where leakage energy contributes a substantial portion of the overall energy consumption budget.

5.2 Overview of Interprocedural Optimizations

In this section we provide a brief overview of the functionality of several commonly implemented interprocedural optimizations. We discuss these optimizations in terms of their potential effect on energy and performance, while providing examples at both the C code level and the assembly level. Similar to the loop restructuring optimizations, having a thorough understanding of the low-level characteristics of interprocedural optimizations allows us develop strategies for a compiler to leverage these optimizations for performance gains while staying within specified energy consumption limits.

5.2.1 Function Inlining

Function inlining, also referred to as inline expansion, is a commonly used interprocedural optimization. In most general terms, function inlining is the process of inserting the code for a function in the place of a function call [42]. Figure 5.1 illustrates a relatively simple example of a C code transformation after the application of function inlining.

The most immediate advantage of function inlining is the removal of the overhead involved in making a function call, which includes the saving and restoring of registers and the passing of parameters. Even more importantly, inlining exposes the function code to the calling environment, enabling subsequent code and data optimizations. An inlined function can be optimized differently for each call site into which it is integrated, making global code optimizations possible. These performance-increasing effects of function inlining would at first glance imply that energy consumption would also be decreased.

However, this is not necessarily the case for all inlined functions. As function inlining involves copying the instructions for a procedure call (potentially) several times, it is often the case that aggressive inlining can lead to an overall code size increase. For systems where power is not a

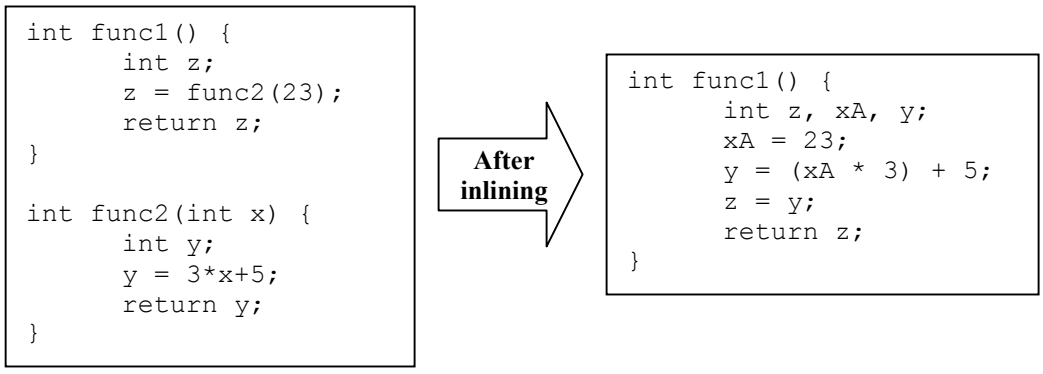

Figure 5.1. Effect of function inlining on sample C code

limiting design constraint, this side effect of aggressive inlining would not be a problematic issue. For the embedded systems we are examining, however, this increase in code size would lead to a larger instruction memory module, with the added possibility that the overall energy consumption will actually increase after optimizations are applied. For this reason and also due to the fact that certain procedure calls offer little or no performance gain after inlining, heuristics are often put in place by compilers to determine a suitable function inlining strategy [29]. Later in this chapter we present our own function inlining heuristic that attempts to improve performance while keeping energy consumption in mind.

Figure 5.2 shows the MIPS IV assembly code that corresponds to the function inlining example of Figure 5.1. From this example we can clearly see the potential for improvement in instructions executed and code size due to procedure call overhead removal (from 28 to 14 instructions). Note, however, that if in our original example function *func2* was called five times or more instead of only once, inlining would lead to a code size increase.

5.2.2 Interprocedural Constant Propagation

During constant propagation analysis, the compiler determines if a given variable is assigned a value only once during execution, and whether that value is obtainable at compile-time. If both are the case, the compiler can choose to replace all instances of that variable usage by a constant. This type of optimization is an extremely effective way of increasing performance while lowering energy consumption.

Interprocedural constant propagation takes this technique and extends it further, by analyzing variables across procedure boundaries. If a variable is not modified when passed as a parameter in a

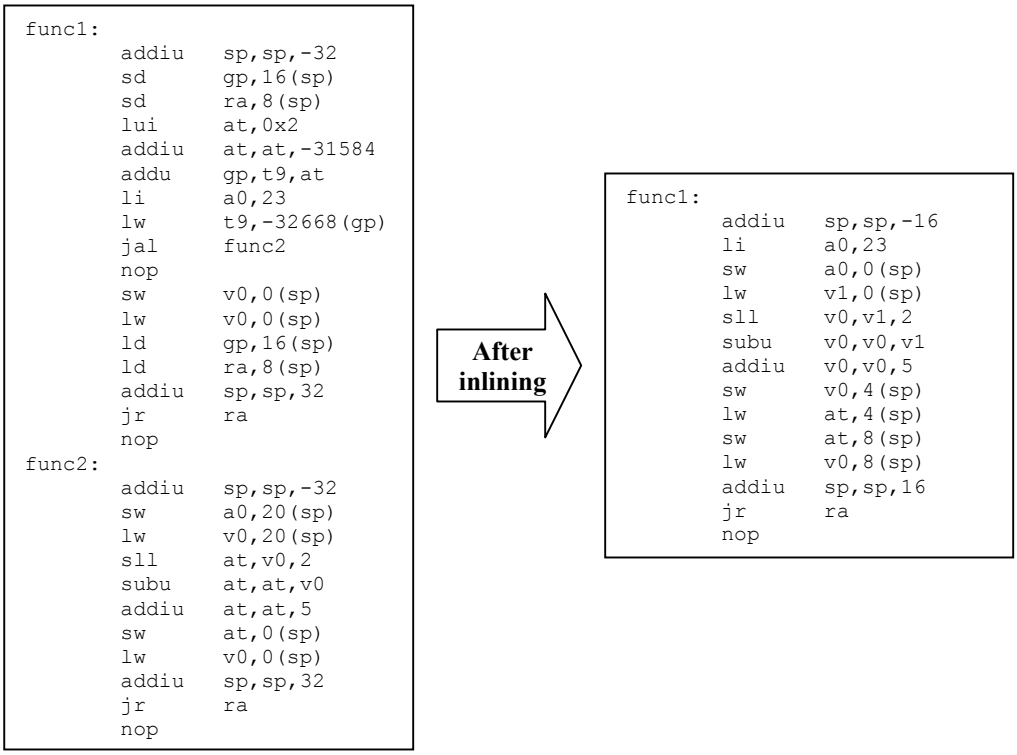

Figure 5.2. Effect of function inlining on equivalent MIPS IV assembly code

function call, its constant value can be propagated both into the function call and later in the calling function. Figure 5.3 shows an example C code segment where interprocedural constant propagation would lead to significant speedups. In this example the values of variables *a* and *b* will be considered constants within function *func2* only if the compiler implements interprocedural analysis.

The only potential negative impact of aggressive interprocedural constant propagation is an increased compile time. This makes little difference in our target system, since an application designer would in general be able to tolerate long compile times to improve energy consumption. Consequently, unlike function inlining, applying interprocedural constant propagation does not lead to a tradeoff between performance and energy.

```
int func1() {
  int a, b, c, d;
  a = 5;
  b = 6;
  c = func2(a, b);
  d = a * c;
  return d;
}

int func2(int e, int f) {
  int g, h;
  g = e + f;
  h = g*e;
  return h;
}
```

Figure 5.3. C code where interprocedural constant propagation would improve performance

5.2.3 Other Optimizations

There are many other interprocedural compiler optimizations that have interesting performance and energy tradeoffs but are not supported by the MIPSPro compiler, so we will not delve into them deeply. Instead, we present an overview of some of the more interesting yet not commonly implemented interprocedural optimizations:

- *Procedure cloning* is an interprocedural optimization where the compiler creates specialized copies of procedure bodies [10]. The compiler then partitions the function calls among the specialized clones in such a way that each call allows for better optimization. Cloning increases the ability of other optimizations to improve performance, but can often lead to excessive code growth if not bounded by some heuristic.

- *Loop embedding* is utilized in situations where a function call is found inside a loop [42]. Instead of utilizing inlining, a compiler can choose to take the code for a loop and embed it into the called function. This will also lead to significant code growth since for most

situations, a special version of the function call code will need to be created, the original being kept for calls that do not occur within loops.

- *Loop extraction* is the inverse operation of loop embedding [42]. Loop extraction is the process of taking an outer loop out of the function call and placing it in the call site. This allows for more loop transformation based optimizations to be leveraged at the call site. Loop extraction has a similar danger of code growth as loop embedding, since for both cases a special version of the function call would be required.

5.3 Analyzing Interprocedural Optimization Modes

In this section we analyze the Interprocedural Analyzer (IPA) mode of the MIPSPro compiler. As discussed earlier, the IPA mode optimizes code by utilizing function inlining, interprocedural constant propagation, and dead function elimination, along with other interprocedural optimizations. The IPA mode attempts to optimize for performance with little restrictions on the growth of generated executables. For this reason it is of interest for us to examine the tradeoffs between code size and performance demonstrated by these interprocedural optimizations, and to analyze the effect on energy consumption for our three target memory hierarchies.

5.3.1 Code Size/Performance Tradeoffs

As mentioned previously, the instruction memory size of our target embedded system is tailored to the custom applications are run on it. Consequently, the code (executable) size of a compiled benchmark is an important metric for estimating the per-access instruction memory energy

consumption. In a system with no instruction caching, the number of instruction memory accesses is set by the number of instructions executed. Therefore it is of interest for us to analyze the graduated instruction count (the MIPS R10000 term for instructions executed) as both a metric for performance and, along with the code size, as a predictor of instruction memory energy consumption.

Figure 5.4 shows the normalized code size and graduated instruction count for six of our benchmarks when compiled with the major loop transforming and interprocedural analyzing modes

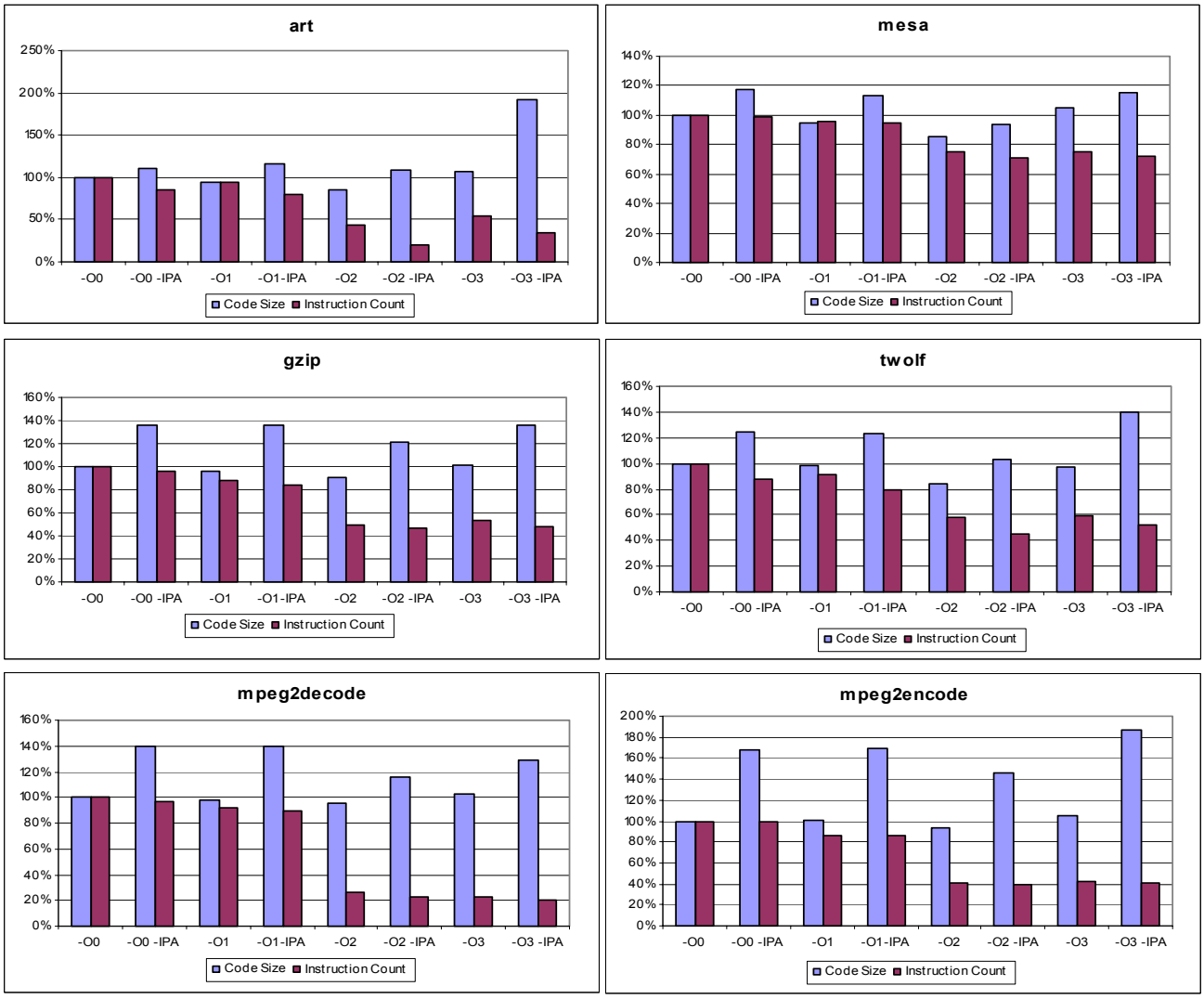

Figure 5.4. Code size and performance tradeoffs of the MIPSPro IPA mode. These results show that the IPA mode leads to a 7% performance improvement along with a 19% code size increase.

of the MIPSPro compiler. As was expected, adding the IPA mode leads to a code size increase; on average compiling with the –Ox –IPA flags instead of the –Ox flag leads to a 19% code size increase across all benchmarks used. The benchmarks from the MediaBench suite, with their relatively low unoptimized code sizes, show the greatest increase in code size as a result of the interprocedural optimizations. As an example the *mpeg2encode* benchmark shows an average code size increase of 67% with the IPA mode. Conversely the larger benchmarks show less of an increase in code size when these types of optimizations are applied. The *mesa* benchmark shows a relatively small 13% code size increase on average, while the other large benchmark *vortex* (not pictured) shows a 16% code size decrease on average. These differences can be attributed to benchmark organization, where benchmarks with several different calls of one procedure can realize large code size gains after inlining.

Along with the increase in code size, our benchmarks also demonstrate the expected performance improvement when the IPA mode is applied. On average, compiling with the –IPA flag leads to a 7% performance improvement as compared to the same level of optimization without interprocedural analysis across all benchmarks used. The benchmarks that have a relatively small unoptimized instruction count show less of a performance improvement however, an example being the *mpeg2decode* benchmark which only demonstrates a 1% performance increase. As a contrast, the two largest running benchmarks (*art* and *vortex*) show an average performance boost of 19% when the IPA mode is turned on.

With an average code size increase of 19% and an average performance improvement of 7%, we can use our instruction memory energy estimation metric to show, that in a system with no instruction caching, compiling with the IPA mode of the MIPSPro compiler would lead to a 11%

instruction memory energy increase. These results clearly demonstrate the tradeoff between code size and performance inherent in interprocedural optimizations.

5.3.2 Energy Consumption – Configuration I

Because memory configuration I contains no caches, both the loop restructuring and interprocedural optimizations would be expected to have a direct effect on data and instruction memory energy consumption. Figure 5.5 confirms this hypothesis. When compiling with the interprocedural optimizations, there is on average a 5% energy decrease when compared to the modes without IPA across all benchmarks used. This number is misleading however, in the sense that several of our benchmarks demonstrate large energy decreases while others show the direct opposite. As an example, when compiling with the IPA mode the *art* benchmark shows a 21% energy decrease on average, while the *mpeg2encode* benchmark has an increase in energy consumption of 10%.

These contrasting results can be explained by examining the energy breakdown by

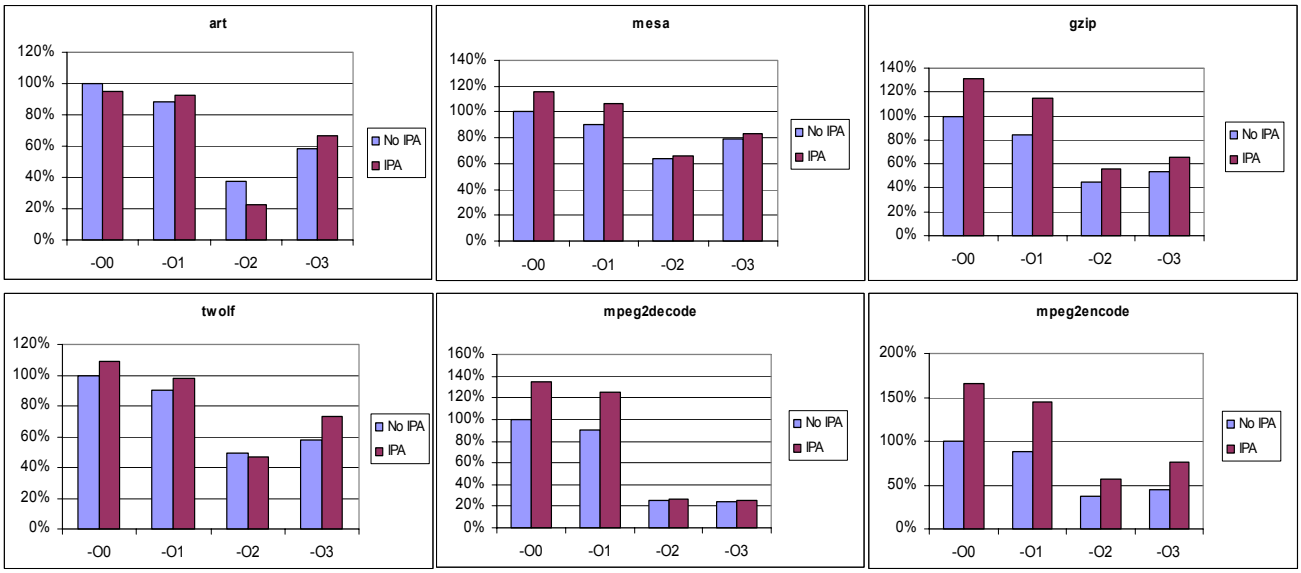

Figure 5.5. Total energy consumption of the MIPSPro IPA mode for configuration I. These results show that the interprocedural-optimizing mode leads to a 5% energy consumption decrease on average.

component for configuration I in Figure 5.6. Due to the code size increase inherent in most interprocedural optimizations, instruction memory energy is increased across almost every benchmark and optimization mode, there being on average a 5% increase when the IPA mode is used. As is to be expected, the benchmarks that have the smallest unoptimized code size show the largest gains in instruction memory energy consumption. For example, the relatively small *gzip* benchmark shows a 21% instruction memory energy consumption increase while on the other side of

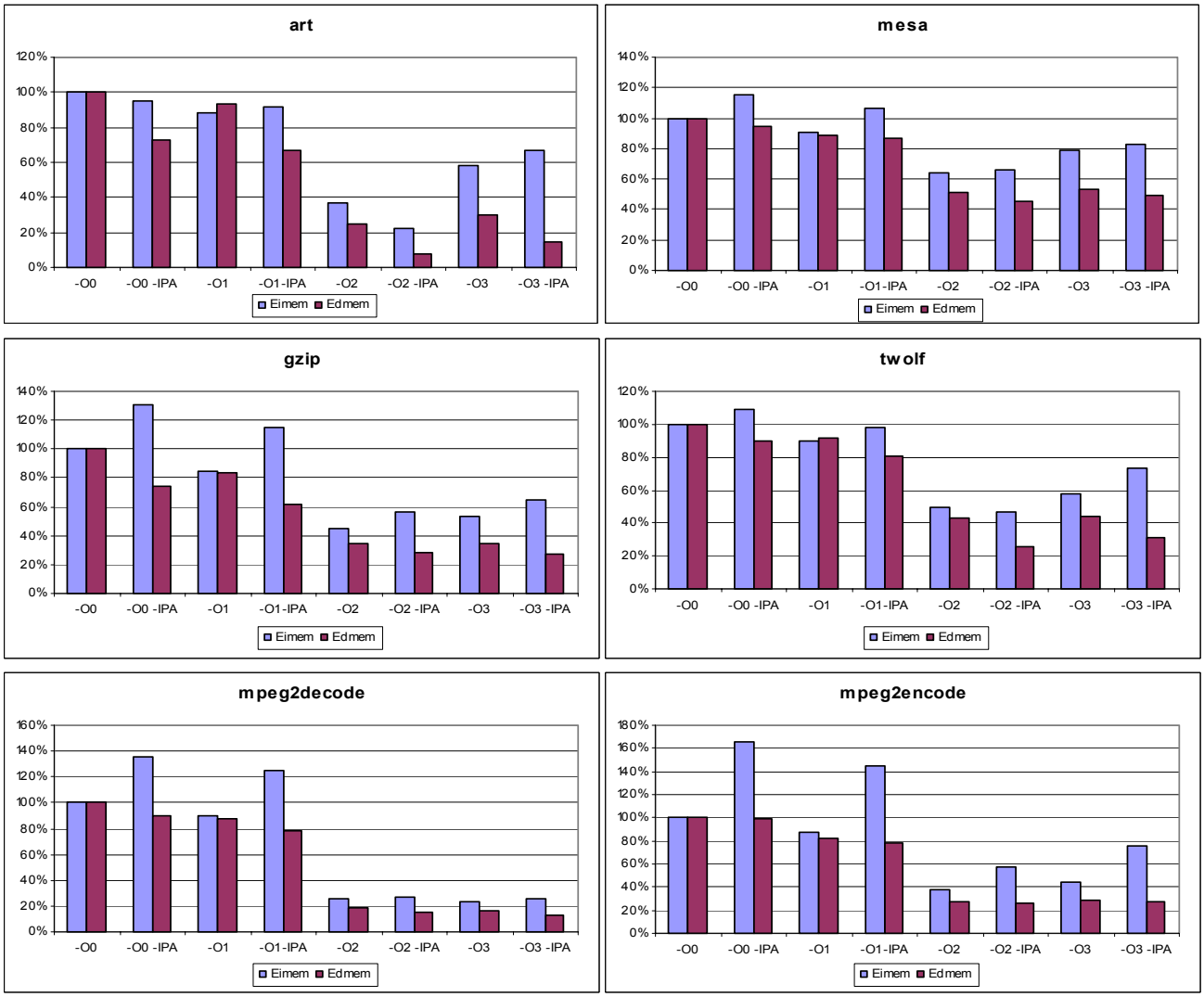

Figure 5.6. IPA mode energy breakdown by component for configuration I. These results show that the growth in code sizes due to interprocedural optimizations lead to instruction memory energy increases.

the spectrum, the relatively large *mesa* benchmark shows only a 10% increase.

For the data memory we see that using the interprocedural optimizations, by providing more global code information to the compiler, allow for more aggressive loop restructuring optimizations. Due to this fact the average data memory energy consumption for configuration I after application of the IPA mode is decreased by 10% across all benchmarks used. Similar to what was shown in Chapter 4, there are discrepancies in the effects of these optimization modes on data memory energy consumption that are dependent on the loop structure of the benchmarks. As an example, the *art* benchmark contains several regular nested loops structures, and shows a relatively large average data memory energy consumption decrease of 22%. Conversely, the *rawcaudio* (not pictured) benchmark spends less time inside loops and shows a relatively small average data memory energy consumption decrease of 2%.

5.3.3 Energy Consumption – Configuration II

The instruction memory side of the memory hierarchy of configuration II is the same as in configuration I. Consequently the same results and analysis apply. The difference is in the data memory hierarchy, were we now consider the effect of the IPA mode on a data memory with caching.

Figure 5.7 shows the total energy consumption of our benchmarks for the interprocedural analyzing mode of the MIPSPro compiler for configuration II. As can be seen from the figure, adding a data cache appears to magnify the effect of the IPA mode on energy consumption. What is really happening is that adding the data cache significantly lowers the overall data memory energy consumption, and consequently the loop transforming operations have less of an effect. When

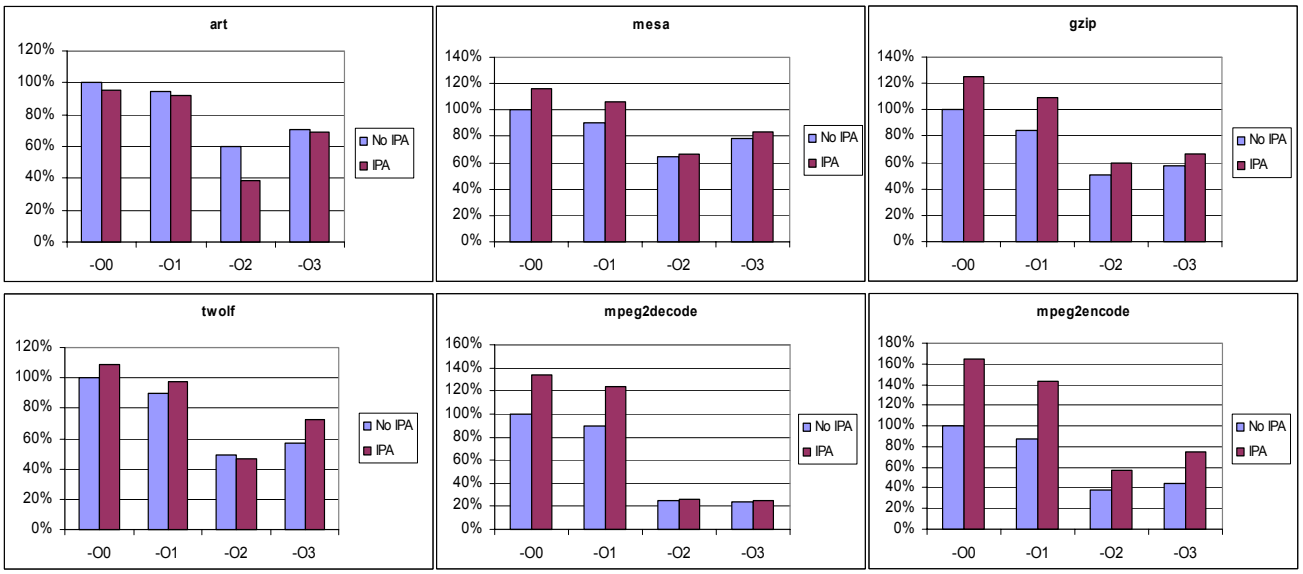

Figure 5.7. Total energy consumption of the MIPSPro IPA mode for configuration II. These results show that as we add the data cache for this configuration, the effect of the interprocedural optimizations on increasing energy consumption is magnified.

compiling with the interprocedural optimizations, there is an average 6% energy increase when compared to the modes without IPA across all benchmarks used. This is in stark contrast to the 5% energy decrease we saw in configuration I. Again as before, the smaller benchmarks tend to show a relatively larger energy increase, an example being the *mpeg2encode* benchmark that demonstrates a 43% energy increase.

Figure 5.8 shows the energy breakdown by component for configuration II. The IPA mode allows the compiler to optimize function calls with knowledge of the calling environment, and the result is that there are far fewer data cache misses. Consequently, the data cache energy consumption is lowered for every one of our benchmarks, the average being a 12% energy decrease across all benchmarks used. An interesting benchmark to look at in this configuration is the *twolf* benchmark. The *twolf* benchmark was customized by SPEC in order to have non-uniform data access patterns, so that the data cache miss rate would be increased to test the performance of caching strategies. The MIPSPro interprocedural analyzer appears to effectively allow the loop

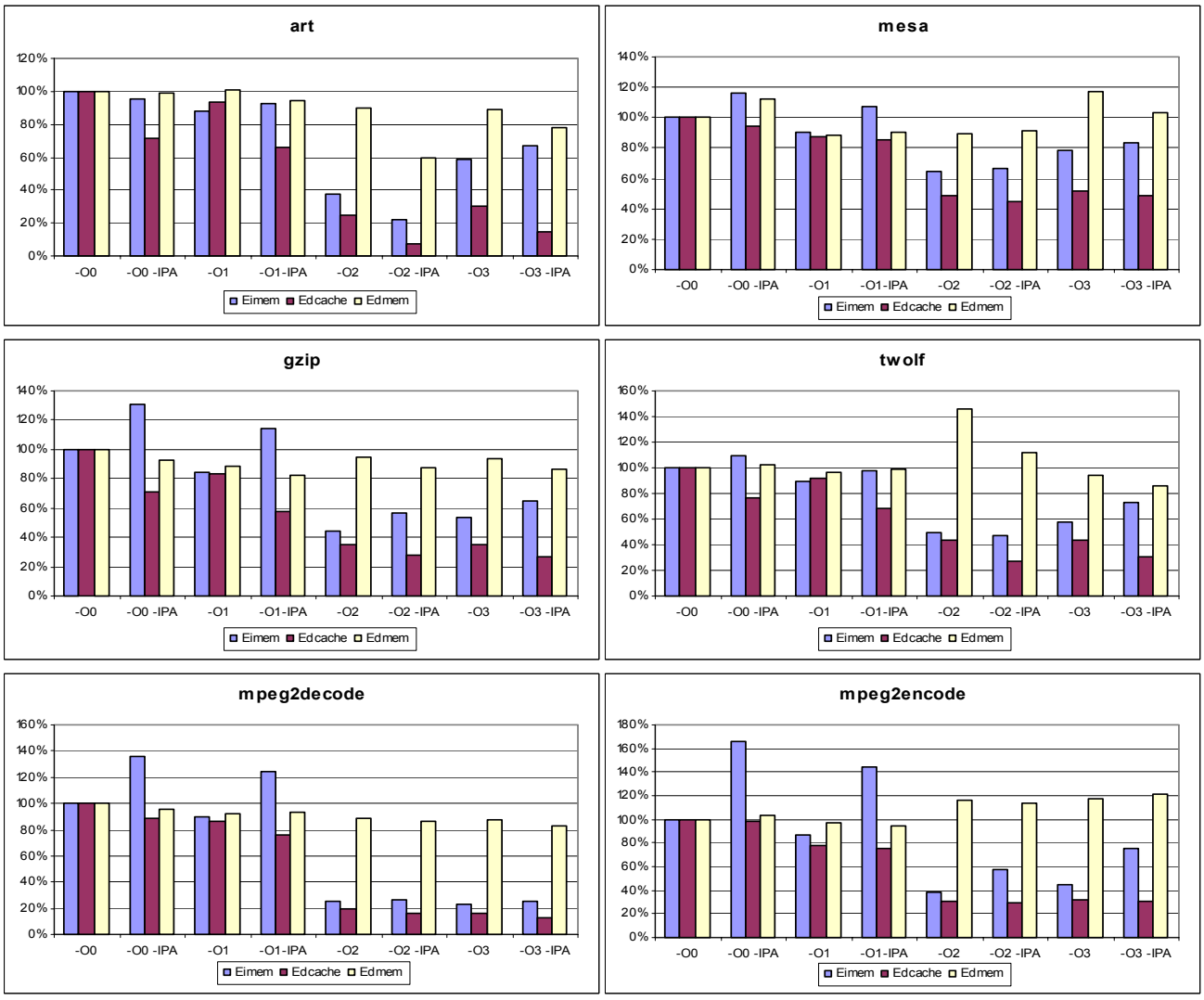

Figure 5.8. IPA mode energy breakdown by component for configuration II. These results show that the interprocedural optimizations allow for more effective loop restructuring optimizations, improving the overall data cache energy consumption.

restructuring optimizations to correct the data access patterns, the result being a decrease in data cache energy consumption of 19%.

Since the overall data memory energy consumption is lowered with the introduction of the cache in configuration II, the effect of the interprocedural optimizations is greatly lessened. Across all benchmarks used, the data memory energy consumption is lowered by 2% on average, as opposed to 10% for configuration I. This result is tempered, however, by the *rawcaudio* benchmark,

which shows a 35% increase in data memory energy. This could be due in part to the fact that the *rawcaudio* benchmark has the smallest number of unoptimized data cache misses.

5.3.4 Energy Consumption – Configuration III

The data memory side of the memory hierarchy of configuration III is the same as in configuration II. Consequently the same results and analysis apply. The difference is in the instruction memory hierarchy, were we now consider the effect of the IPA mode on an instruction memory with caching.

Figure 5.9 shows the total energy consumption of our benchmarks for the interprocedural analyzing mode of the MIPSPro compiler for configuration III. The instruction cache limits the number of accesses to the instruction memory, and consequently the energy improvements of the loop transforming optimizations outweigh the effects of the IPA mode. Across all benchmarks used applying the IPA mode demonstrates a 7% energy consumption decrease. This result now means

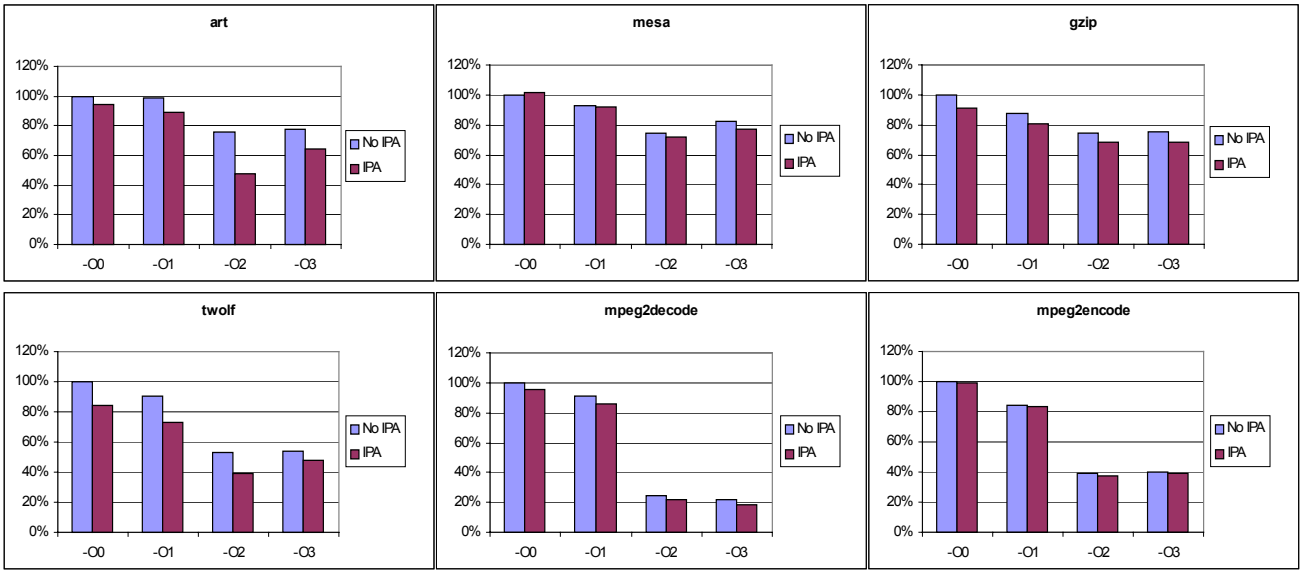

Figure 5.9. Total energy consumption of the MIPSPro IPA mode for configuration III. These results show that adding an instruction cache limits the number of accesses to the instruction memory, and consequently the IPA mode increases the total energy consumption by less of a factor.

that the IPA mode has demonstrated energy improvements in configurations I and III, with the only increase in energy being seen in configuration II. A possible explanation of this occurrence is that when the caching strategies on both sides of the memory hierarchy are equally complex, the effect of the loop restructuring optimizations on decreasing data memory energy consumption outweighs the energy increases inherent in the interprocedural optimizations. Only when the energy consumption of the data memory hierarchy is made relatively small with caching, as in configuration II, do the code size increases of the IPA mode lead to an increased overall memory energy consumption.

Figure 5.10 shows the energy breakdown by component for configuration III. Interprocedural optimizations such as function inlining increase the locality of instructions, and consequently the number of instruction cache misses should be lowered. This was the case for our experiments, as the instruction cache energy consumption was lowered for each of our benchmarks, the average being about 7% when the IPA mode was turned on. The smaller benchmarks showed less of an improvement in instruction cache energy consumption, and example being the *mpeg2encode* benchmark that only shows a 1% decrease.

Since improving the cache performance also means less memory accesses, the introduction of interprocedural analysis should have a positive effect on instruction memory energy consumption. Instead in Figure 5.10 what we see are wildly varying differences in instruction memory energy consumption for the different benchmarks and optimization mode. On average there is a 17% increase in instruction memory energy consumption when the IPA mode is turned on across all benchmarks used, with the maximum increase being found in the *mesa* and *gzip* benchmarks which both show a 43% increase. These results can be explained by the fact that the instruction memory energy consumption in configuration III is relatively small; the instruction cache consumes several

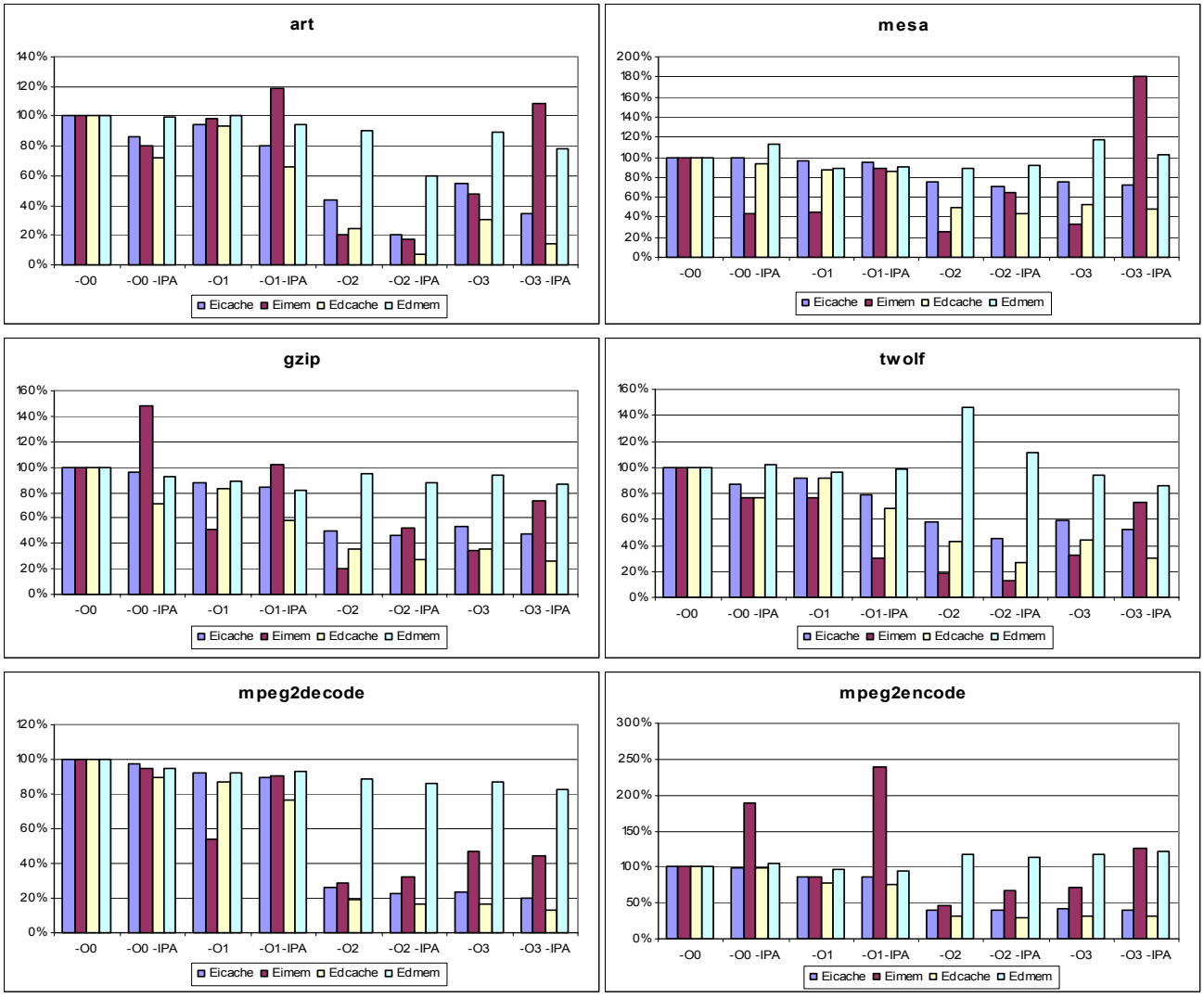

Figure 5.10. IPA mode energy breakdown by component for configuration III. These results show that the instruction cache energy consumption is decreased on average due to the fact that interprocedural optimizations often increase instruction locality.

times the energy as does the instruction memory. Therefore what we are seeing as large increases are actually relatively small when compared to the rest of the memory hierarchy.

5.4 Tailoring Inlining to Energy Requirements

Since overly aggressive interprocedural optimizations can lead to an undesirable tradeoff between energy consumption and performance, it is of interest to investigate tailoring these optimizations to fit to energy constraints. In this section we present and analyze a heuristic that provides a systematic way of choosing a suitable function inlining strategy that attempts to improve performance while keeping in mind energy consumption. Note that a similar approach to ours for limiting the effects of function inlining has been proposed by Leupers and Marwedel in [22]. Our work is different from theirs in that we focus on energy consumption.

5.4.1 Inlining Heuristic

Figure 5.11 shows our energy-aware inlining heuristic. Our approach to the problem is as follows. We start with the unoptimized program and, at each step, try to select the most appropriate (function, call-site) pair and perform inlining. After an inlining is performed, we estimate or measure the resulting energy consumption and compare it with the energy upper bound. If we are still under the upper bond, we select the next (function, call-site) pair, and so on. The process stops when the energy consumption after the inlining becomes larger than the upper bound. When this occurs, we undo the last inlining, and return the resulting code as the output. This algorithm does not attempt to find an optimal function inlining strategy in terms of energy consumption; it just guarantees that a specified upper-limit is not exceeded.

Note that our heuristic is not specific about how the most appropriate (function, call-site) pairs are chosen, and in practice, there are several methods that can be used in order to select these pairs. The simplest approach is by brute force. An optimizing compiler can iterate through every possible function call to choose the one that gives the most performance benefit when inlined. This

```
ENERGY_INLINE( C, E_limit ) {
    C_new = C;
    repeat {
        C_old = C_new;
        F = function_list( C_old );
        if ( |F| < 2) then {
            return ( C_old );
        }
        else {
            [f_inline, cs_inline] = best inlinable candidate ∈ F
            C_new = perform_inlining( C_old, f_inline, cs_inline );
        }
        E = estimate_energy( C_new );
    }
    until ( E > E_limit );
    return( C_old );
}
```

Figure 5.11. Energy-aware function inlining heuristic

approach, while not very efficient, works well for applications with relatively small numbers of (function, call-site) pairs.

A more advanced approach involves using SpeedShop to generate a basic-block profiling run, where the exact number of times that a function is called from a particular call-site can be exposed to the compiler. With this information, we can choose functions to inline according to how many times they are called. In practice, functions can be of varying size, so that a function that is called less might not necessarily have a smaller impact on energy when inlined. Consequently, for our experiments we used the SpeedShop tool to generate lists of function calls where inlining was potentially beneficial, and then we applied the brute-force approach to select the most appropriate (function, call-site) pairs.

5.4.2 Results

For our experiments we chose to look at the energy consumption of our selected benchmarks for configuration II, since this configuration highlights the energy increasing effects of interprocedural optimizations. We selected the *equake*, *mesa*, and *vortex* benchmarks since they have a multitude of functions to inline, and from Section 5.3 we saw that they are examples of applications that demonstrate code size and performance tradeoffs when applying interprocedural optimizations.

Figure 5.12 shows the performance (in terms of graduated instruction count) for our selected unoptimized benchmarks when our inlining heuristic is applied under different memory energy upper bounds. It can be observed that we achieve a performance improvement of 3% on the average across all energy bounds used. It can also be seen that as we increase our energy upper bound (i.e., allow more energy consumption in instruction memory), our function inlining strategy produces better-performing code, demonstrating the tradeoff between energy and performance. On average, our heuristic algorithm provides a 2.5% energy consumption savings while keeping performance within 1.5% of the most aggressive inlining strategies. It is also to be noted that there is little benefit in our example to inlining every (function, call-site) pair, both in terms of performance and the memory energy expended. As an example, for the *mesa* benchmark, inlining at every possible

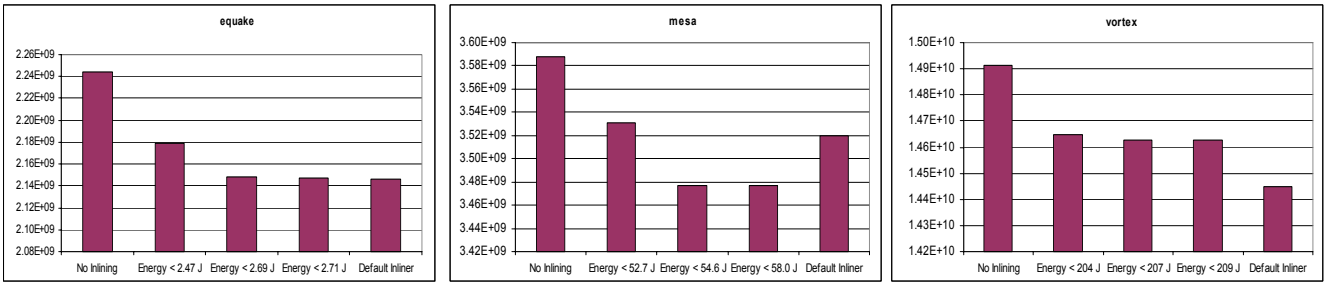

Figure 5.12. Performance in terms of instruction count as a function of the energy upper bound of our inlining heuristic algorithm. These results show that our heuristic demonstrates energy savings while keeping performance within a few percent of the aggressive default inliner of the MIPSPro compiler.

choice leads to a code that dissipates over 76 (J) while our heuristic chooses an inlining strategy that dissipates just under 55 (J) and performs slightly better. In general, inlining every possible function call is not a good idea since for most applications, there will be a significant number of (function, call-site) pairs where the overhead reduction from inlining is overwhelmed by the size of the procedure being inlining. The results given in Figure 5.12 indicate that our heuristic improves performance while keeping the energy consumption below a pre-set limit.

5.5 Considering Leakage Energy

Up to this point we have been only analyzing interprocedural optimizations in terms of dynamic (switching) energy consumption. While in CMOS circuits dynamic energy is the dominant part, current trends indicate that the contribution of static (leakage) energy to the overall energy budget will increase exponentially in upcoming circuit generations [7]. The leakage energy consumption of large SRAM memories is expected to be particularly problematic due to the fact that it increases with the size of memory.

In this section, we first show the impact of taking leakage into account on the tradeoff between memory energy and performance of interprocedural optimizations. After that, we show how our energy-sensitive compilation approach in the previous section performs when leakage is accounted for. We concentrate on the relative weight of leakage energy to dynamic energy by using the approximation developed in Chapter 2, which is to take the per-cycle leakage energy consumption to be a ratio of the per-access dynamic energy consumption. We represent that ratio as a nonnegative value k where a smaller k value ($0.1 <= k <= 0.2$) represents current fabrication

technologies and larger k values ($0.5 < k <= 1.0$) represent a futuristic scenario where the effect of leakage energy will begin to outpace that of switching energy.

Figure 5.13 shows our results when we compile six of our benchmarks using the IPA modes of the MIPSPro compiler. These results detail the actual values for energy consumption for configuration II when we take the leakage energy of the instruction memory into account. These

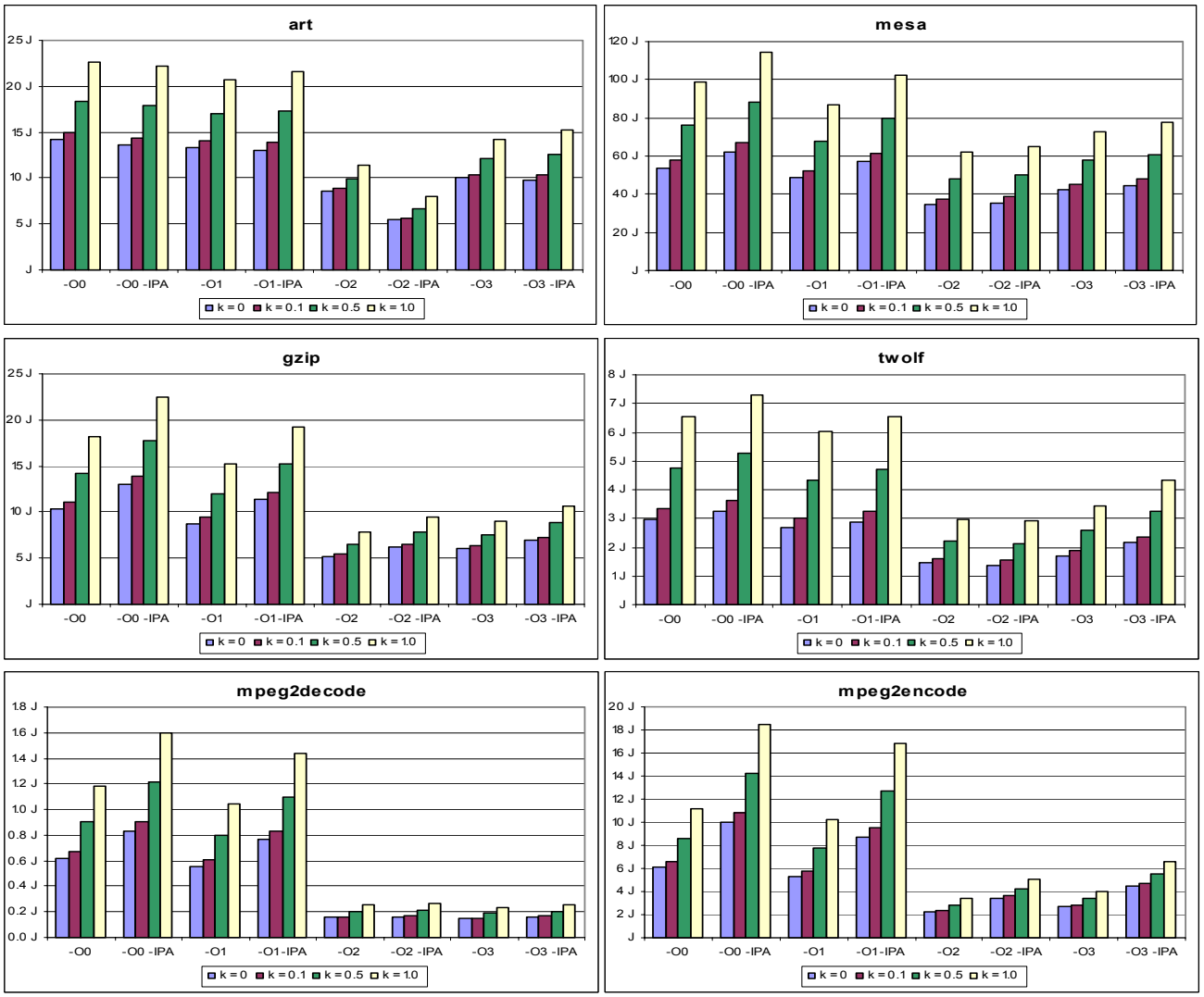

Figure 5.13. Total energy consumption (as the sum of both dynamic and static consumption) for different interprocedural optimization modes. The value of *k* refers to the relative weight of the per-cycle leakage energy to the per-access dynamic energy. These results show that as we increase the weight of leakage energy, optimizations that tend to expand the code size lead to more pronounced energy increases.

results show an increasingly pronounced tradeoff between performance and energy when compared to the equivalent results of Section 5.3. As an example, when leakage energy is not taken into consideration, the *mesa* benchmark shows an average energy increase of just over 5 (J) when adding interprocedural optimizations to the standard optimization modes of the MIPSPro compiler. As we increase the relative weight of the leakage energy to the total energy equation, we see this number steadily grow. When k = 0.1, our average energy increase is 5.5 (J), when k = 0.5 the average increase is 7.3 (J), and when we consider the weight of leakage energy to be equal to that of dynamic energy at k = 1.0, the average energy consumption increase is 9.6 (J). These results demonstrate that as leakage energy becomes more of an issue in future design methodologies, performance optimizations that have a side effect of an increased code size will have a greater effect on increasing energy consumption.

Figure 5.14 shows our results when we apply our function inlining algorithm to the *equake*, *mesa*, and *vortex* benchmarks with an energy upper bound that takes leakage energy into account. For this experiment, we plotted the minimum energy upper bound that would be required for certain performance improvements. The results in Figure 5.14 show that as we increase k, we require a greater upper energy bound to achieve the same performance via inlining. As an example, for the *vortex* benchmark, to achieve a performance improvement of 2% (as represented by the middle bar in each subgraph in the bottom row of Figure 5.14), there is an approximately 170 (J) difference between energy upper bounds when k = 0.1 and k = 1.0. Clearly, as leakage energy becomes more of a factor in future design fabrication methodologies, it will become increasingly important to limit code and energy growth at the expense of performance gains.

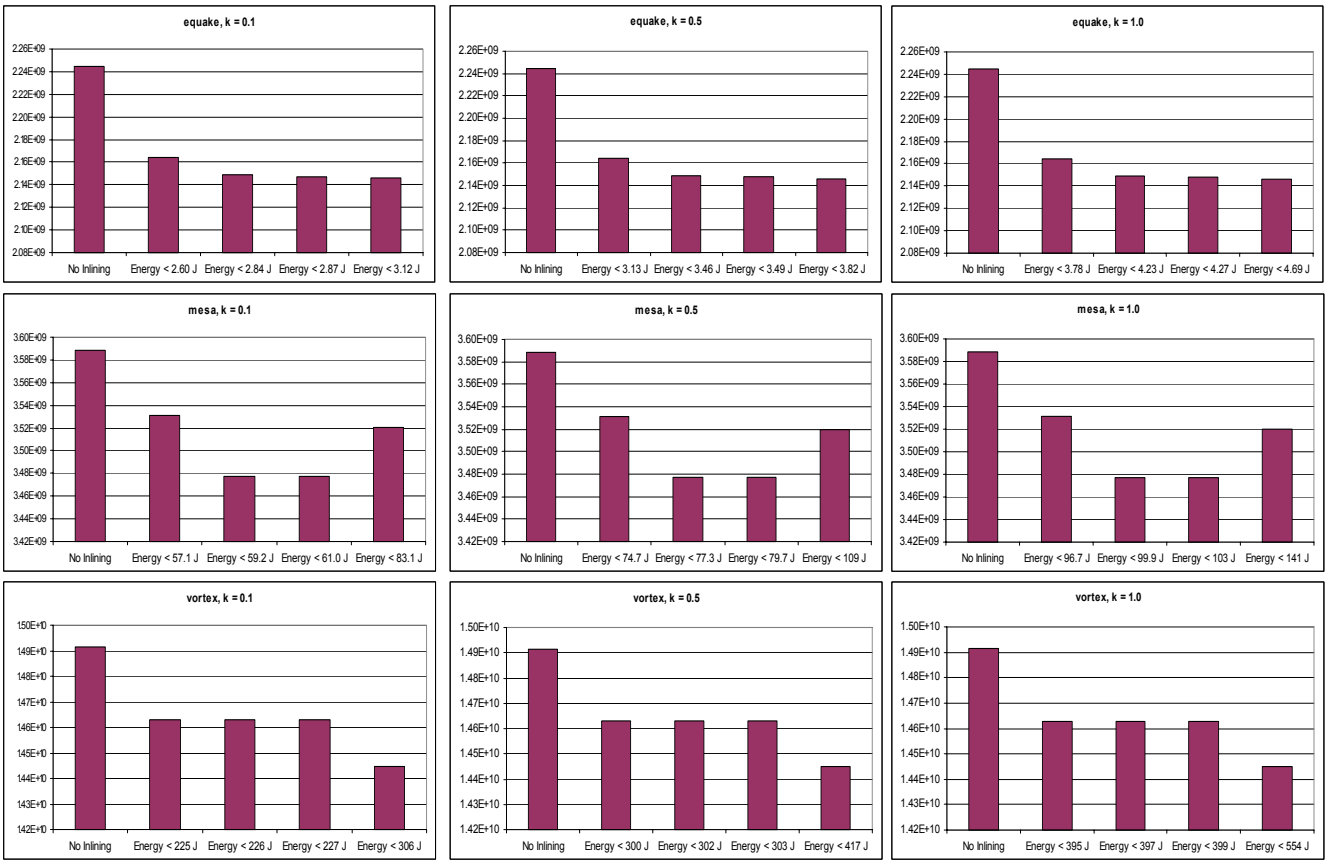

Figure 5.14. Performance versus memory energy upper bound of our inlining heuristic algorithm when including leakage energy for different values of *k*. These results demonstrate that our heuristic is even more energy-efficient when leakage energy becomes more of a factor.

5.6 Summary

In this chapter we have analyzed the energy consumption characteristics of some standard interprocedural optimizations. In general, these optimizations have the potential to increase the overall memory energy consumption. As an example, compiling with the MIPSPro –IPA mode leads to an average 7% performance improvement for configuration II, which is offset by a 6% energy consumption increase. These energy increases are almost entirely due to an expanding code size inherent in these optimizations, requiring a larger instruction memory.

In order to minimize the energy increasing effects of these interprocedural optimizations, we demonstrated that a simple compiler heuristic can be leveraged to improve performance via function inlining while keeping energy consumption under a predetermined bound. Using this heuristic, we were able to improve energy consumption by 2.5% while keeping the performance within 1.5% of the more aggressive loop unrolling strategies. It should be noted that usefulness of these types of optimizations is extremely limited when used in isolation, and in general interprocedural optimizations such as function inlining would only be performed when future loop restructuring optimizations are expected to be later applied. Consequently, a compiler that optimizes for energy consumption might consider skipping interprocedural optimizations entirely.

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

Power consumption is an important design focus for embedded and portable systems. In this work, we have provided a systematic methodology for analyzing the effect of compiler optimizations on memory energy consumption. In doing this, we have derived detailed analytical energy consumption models for on-chip memory structures that would be found in embedded systems, and have applied these models towards memory hierarchies such as would be found in System-on-Chip (SoC) devices. We also discussed how to leverage software on a general-purpose processor in order to profile hardware counters that can be used to provide the run-time data for these energy models.

Using our energy estimation methodology, we have analyzed both loop restructuring optimizations and interprocedural optimizations, and have shown data regarding the energy consuming properties of both isolated optimizations and optimization suites. With this data we have been able to clearly demonstrate the energy and performance tradeoffs inherent in the more aggressive of these optimizations. Realizing that many performance-oriented optimizations lead to

an undesirable growth in code sizes, we have shown that the product of instruction count and code size is a fairly accurate energy estimate. We have also presented heuristics that attempt to tailor the aggressiveness of many of these optimizations, and have analyzed their effectives. Finally, we have clearly demonstrated that our techniques will gain greater importance in future design generations, where leakage energy will constitute a larger percentage of the overall memory energy budget, and consequently optimizations that increase code size will increase instruction memory energy consumption by a greater factor.

The results presented in this work can be exploited in several ways. First, since embedded systems can tolerate much larger compilation times than their general purpose counterparts (as many of them run a single application for which a large number of processor cycles can be spent in compilation), we can run different optimized versions of the code and select the one with the best energy efficiency. Our results presented in this paper indicate that the compiler should attempt to minimize the product of the code size and dynamic instruction count since it is a very good first level approximation for instruction memory energy consumption. Second, systems with strict energy requirements can utilize our function inlining heuristic to improve performance without violating design constraints. We have shown that the interprocedural optimizations such as function inlining can often lead to dramatic energy consumption increases, while the loop transforming optimizations, if applied intelligently, can lead to energy decreases alongside performance gains. Since interprocedural optimizations are often applied before their loop transforming counterparts, in order to maximize performance it would make sense to allow for the interprocedural optimizations to increase energy a certain percentage past the desired maximum, as the application of the loop transformations will be able to bring the energy consumption level back down to the limit.

6.2 Future Work

In the future the results presented in this thesis can be applied towards other more advanced low-power research projects. An excellent example is the PACT (Power aware Architecture and Compilation Techniques) project at Northwestern University. The objective of the PACT project is to develop power-aware architectural techniques with associated compiler and CAD tool support. The goal is to take Department of Defense (DOD) applications written in C and generate power and performance-efficient code for systems utilizing the architectural power-aware techniques developed. The current structure of the PACT project is summarized in Figure 6.1.

Our work could best be applied in the compiler side of the PACT project, where the current design goal is to take a design description in C and automatically generate System-on-Chip (SoC) designs that incorporate ARM cores, FPGAs, and ASICs, along with numerous memory devices. Figure 6.2 shows a sample SoC that utilizes the Advanced Microcontroller Bus Architecture

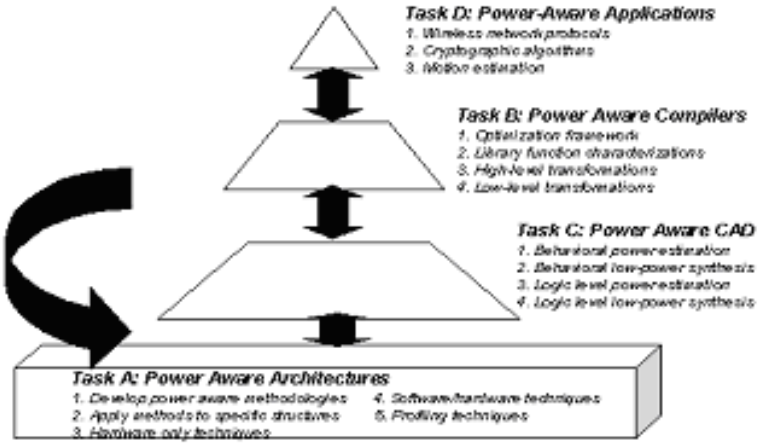

Figure 6.1. Overview of the PACT project

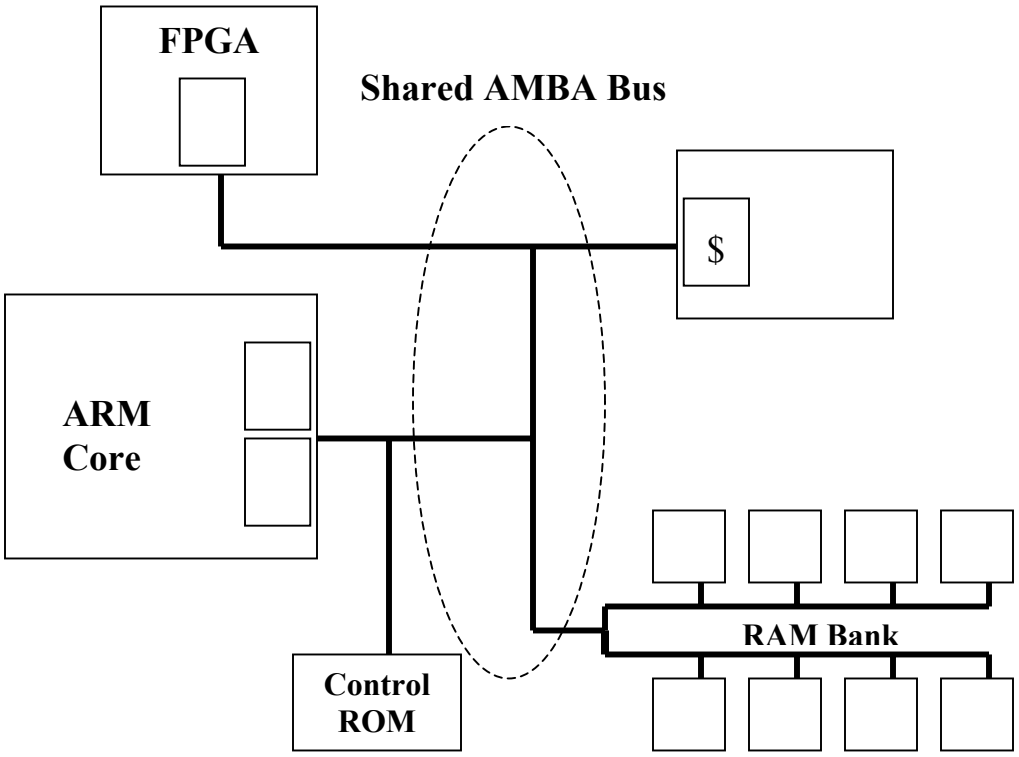

Figure 6.2. Sample System-on-Chip using AMBA

(AMBA) standard [1] developed at ARM as its interconnection framework. The AMBA bus attempts to handle all the details of bus arbitration and address decoding in a low-power manner. Optimizations that are meant to improve cache performance gain greater importance in an AMBA configuration, as a cache miss from the ARM processor can lead to lengthy latencies due to contention on the bus from the other on-chip resources. Consequently our future work will include adapting our energy model to reflect the AMBA bus and analyzing how our loop restructuring and interprocedural optimizations effect the energy consumption of different SoC configurations.

References

[1] *AMBA Specification Rev 2.0*. ARM Limited, 1999.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. Of the 33rd Int'l Symposium on Microarchitecture*, 2000.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Int'l Symposium on Computer Architecture*, 2000.

[4] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342. University of Wisconsin-Madison, June 1997.

[5] J. Butts and G. Sohi. A static power model for architects. In *Proc. of the 33rd Int'l Symposium on Microarchitecture*, 2000.

[6] 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.

[7] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[8] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded java environment. In *Proc. of the 8th Int'l Symposium on High Performance Computer Architecture*, 2002.

[9] Z. Chen, M. Johnson, L. Wei, and K. Roy. Estimation of standby leakage power in CMOS circuits considering accurate modeling of transistor stacks. In *Proc. of Int'l Symposium on Low-Power Electronics and Design*, 1998.

[10] K. Cooper, M. W. Hall, K. Kennedy. Procedure cloning. In *Proc. of the 1992 IEEE Int'l Conference on Computer Language*, 1992.

[11] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proc. Of the 7th Int'l Symposium on High Performance Computer Architecture*, 2001.

[12] J. Edmondon et. al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119-135, 1995.

[13] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley, 2000.

[14] N. B. I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proc. Of Int'l Symposium of Low-Power Electronics and Design*, 1998.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[16] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. In *Proc. of the Int'l Symposium on Low-Power Electronics and Design*, 2001.

[17] M. B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proc. of Int'l Symposium on Low-Power Electronics and Design*, 1997.

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. Of the 28th Int'l Symposium on Computer Architecture*, 2001.

[19] J. Laudon and D. Lenoski. System Overview of the SGI Origin 200/2000 Product Line. In *Proc. of IEEE COMPCON '97*, 1997.

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, 1997.

[21] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[22] R. Leupers, P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proc. of the Int'l Conference on Computer Aided Design*, 1999.

[23] M. Martonosi, D. Brooks, and P. Bose. Modeling and analyzing CPU power and performance: metrics, methods, and abstractions. Held in conjunction with *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2001.

[24] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proc. of the 25th Int'l Symposium on Computer Architecture*, 1998.

[25] *MIPSpro Compiling and Performance Tuning Guide*. Silicon Graphics, Inc., 1999.

[26] *MIPS R10000 Microprocessor User's Manual*. MIPS Technologies, Inc., 1996.

[27] J. Montanaro et. Al. A 160-MHz, 32-b, 0.5W CMOS RISC microprocessor. *Digital Technical Journal*, 9(2):49-62, 1996.

[28] R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[29] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[30] Northwestern University - Center for Parallel and Distributed Computing. http://www.ece.northwestern.edu/cpdc.

[31] *Origin2000 and Onyx2 Performance Tuning and Optimization Guide*. Silicon Graphics, Inc., 1997.

[32] M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. of Int'l Symposium of Low-Power Electronics and Design*, 2001.

[33] Jan Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall Publishing, 1996.

[34] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In *Proc. of the 38th Design Automation Conference*, 2001.

[35] Silicon Graphics, Inc. http://www.sgi.com.

[36] D. Singh and V. Tiwari. Power challenges in the Internet world. In *Cool Chips Tutorial: An Industrial Perspective on Low Power Processor Design* (held in conjunction with *The 32nd Int'l Symposium on Microarchitecture*), 1999.

[37] A. Sinha, A. Chandrakasan. JouleTrack – a web based tool for Software energy profiling. In *Proc. of the 38th Design Automation Conference*, 2001

[38] The Standard Performance Evaluation Corporation. http://www.spec.org, 2000.

[39] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. of the 27th Int'l Symposium on Computer Architecture*, 2000.

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

[41] M. Wolf, D. Mayden, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the 29rd Int'l Symposium on Microarchitecture*, 1997.

[42] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[43] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[44] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high performance circuits. In *IEEE Symposium on VLSI Circuits*, 1998.