December 1998

UILU-ENG-98-2226
DAC-70

University of Illinois at Urbana-Champaign

Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors

Nikolaos Bellas

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

UNCLASSIFIED

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-98-2226 (DAC-70)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Intel
c. ADDRESS (City, State, and ZIP Code) 1308 W Main St Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 5200 NE Elam Young Pkwy Hillsboro, OR 97124
a. NAME OF FUNDING/SPONSORING ORGANIZATION Intel 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code) 5200 NE Elam Young Pkwy Hillsboro, OR 97124	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors

12. PERSONAL AUTHOR(S)
Bellas, Nikolaos

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day) 98 Dec 17	15. PAGE COUNT 126

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Low power design; memory hierarchy; compiler optimizations; cache memories

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The microprocessor industry has started viewing power, along with area and performance, as a decisive design factor in today's microprocessors. The increasing cost of packaging and cooling systems poses stringent requirements on the maximum allowable power dissipation. Most of the research in recent years has focused on the circuit, gate, and register-transfer (RT) levels of the design. In this research, we focus on the software running on a microprocessor and we view the program as a power consumer. Our work concentrates on the role of the compiler in the construction of "power-efficient" code, and especially its interaction with the hardware so that unnecessary processor activity is saved. We propose techniques that use extra hardware features and compiler-driven code transformations that specifically target activity reduction in certain parts of the CPU which are known to be large power and energy consumers.

Design for low power/energy at this level of abstraction entails larger energy gains than in the lower stages of the design hierarchy in which the design team has already made the most important design commitments. The role of the compiler in generating code which exploits the processor organization is also fundamental in energy minimization. Hence, we propose a hardware/software co-design paradigm, and we show what code transformations are necessary by the compiler so that "wasted" power in a modern microprocessor can be trimmed.

More specifically, we propose a technique that uses an additional mini cache located between the instruction cache (I-Cache) and the CPU core; the mini cache buffers instructions that are nested within loops and are continuously fetched from the I-Cache. This mechanism can create very substantial energy savings, since the I-Cache unit is one of the main power consumers in most of today's high-performance microprocessors. Results are reported for the SPEC95 benchmarks in the R-4400 processor which implements the MIPS2 instruction set architecture.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) | 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ARCHITECTURAL AND COMPILER TECHNIQUES FOR ENERGY REDUCTION IN HIGH-PERFORMANCE MICROPROCESSORS

BY

NIKOLAOS BELLAS

Diploma, University of Patras, 1992
M.S., University of Illinois at Urbana-Champaign, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ARCHITECTURAL AND COMPILER TECHNIQUES FOR ENERGY REDUCTION IN HIGH-PERFORMANCE MICROPROCESSORS

Nikolaos Bellas, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1999
Ibrahim Hajj, Advisor

The microprocessor industry has started viewing power, along with area and performance, as a decisive design factor in today's microprocessors. The increasing cost of packaging and cooling systems poses stringent requirements on the maximum allowable power dissipation. Most of the research in recent years has focused on the circuit, gate, and register-transfer (RT) levels of the design. In this research, we focus on the software running on a microprocessor and we view the program as a power consumer. Our work concentrates on the role of the compiler in the construction of "power-efficient" code, and especially its interaction with the hardware so that unnecessary processor activity is saved. We propose techniques that use extra hardware features and compiler-driven code transformations that specifically target activity reduction in certain parts of the CPU which are known to be large power and energy consumers.

Design for low power/energy at this level of abstraction entails larger energy gains than in the lower stages of the design hierarchy in which the design team has already made the most important design commitments. The role of the compiler in generating code which exploits the processor organization is also fundamental in energy minimization. Hence, we propose a hardware/software co-design paradigm, and we show what code transformations are necessary by the compiler so that "wasted" power in a modern microprocessor can be trimmed.

More specifically, we propose a technique that uses an additional mini cache located between the instruction cache (I-Cache) and the CPU core; the mini cache buffers instructions that are nested within loops and are continuously fetched from the I-Cache. This mechanism can create very substantial energy savings, since the I-Cache unit is one of the main power consumers in most of today's high-performance microprocessors. Results are reported for the SPEC95 benchmarks in the R-4400 processor which implements the MIPS2 instruction set architecture.

To my parents, my brother, and Vana

ACKNOWLEDGMENTS

I would like first of all to express my gratitude to my academic advisor, Professor Ibrahim Hajj, for the freedom, and support he gave me to finish my research work, and for his patience and encouragement for all those years that he has served as my advisor. I would also like to thank my co-advisor, Professor Constantine Polychronopoulos, for his invaluable technical and ethical support, and his friendship. Professors Farid Najm, and Naresh Shanbhag served as members of my commitee, and I am indebted to them for their help through comments in the course of this research effort. I would also like to thank Intel Corp. who funded this project, and especially Vivel Tiwari and George Stamoulis for their technical feedback.

A series of people have shaped my life during my stay in Champaign. In particular, I would like to thank Sudhakar Bobba, Howard Chen, David Craig, Antonios Doufas, Manish Goel, Subodh Gupta, Mahadev Nemani, Raj Panda, Sumant Ramprasad, George Stamoulis, Nick Stavrakos, and Andreas Veneris, for their friendship all those years, and for their interesting technical and non-technical discussions we had together. Also to my previous academic advisor, Prof. Dan Saab, and my previous officemates, Ben Mathew and Jalal Wehbeh for their support and friendhip in my first year here in Illinois. I would also like to express my deep gratitude to Professor Dennis Assanis, his wife Eleni and their children, for making Champaign a much better place to live, and helped me adapt to the life in Illinois when I first came in the university.

Lastly, and most importantly, I want to thank my family, who has always supported me in all my endeavors, and has always been there to stand by me in the ups and downs of life. And finally to Vana for her unconditional love and patience with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In recent years, power dissipation has become a major design concern for the microprocessor industry. The shrinking device size, and the large number of devices packed in a chip die coupled with the large operating frequencies, have led to unacceptably high levels of power dissipation.

The problem appears to be more acute in portable systems which operate under the energy constraints of a battery. These systems need a small battery for portability, which should provide enough energy to keep the system running for as long as possible. Hence, the energy dissipation of the portable system should be low so that it does not drain the battery quickly. On the other hand, desktop systems operate in an increasingly "hot" environent. The layout compaction and the high operating frequencies entail high power densities and, thus, high thermal stresses on the chip. If this trend is not controlled by low power techniques, the chip will have reliability problems, such as electromigration, which is due to the high current densities on the metal interconnects. Such low-power techniques also help in keeping the cost of packaging low, thus reducing the cost of the final product.

Optimizing for power is a hard problem because in CMOS and BiCMOS technologies the chip components draw power supply current only during a logic transition. Although this is an attractive characteristic of these circuits, it makes the power consumption dependent on the switching activity inside them. In other words, the same circuit will dissipate a different amount of power under different input vectors.

The power dissipation in a CMOS circuit is the sum of three components:

$$P = P_{dyn} + P_{short} + P_{leakage}. \tag{1.1}$$

Those components are the dynamic power, the short circuit power and the leakage power, respectively. The first term dominates power consumption in CMOS circuit and has attracted the vast majority of research in low-power systems in the last years.

The energy required to make a full transition from '0' to '1' and, then, from '1' to '0' at the output node of a logic gate is the sum of the individual energies E_{lh} and E_{hl}, and is provided by the supply voltage

$$E = E_{lh} + E_{hl} = C_{out} V_{dd}^2. \tag{1.2}$$

The dynamic power P_{dyn} consumed by a CMOS circuit (Fig. 1.1) in a clock cycle is given by the formula

$$P_{dyn} = \frac{1}{2} C_{out} V_{dd}^2 f, \tag{1.3}$$

where V_{dd} is the supply voltage, f is the clock frequency, and C_{out} is the physical capacitance at the output of the circuit, and is the sum of two components: the capacitance between the output node and the supply voltage V_{dd}, and the capacitance between the output node and the ground.

Equation 1.3 assumes that the output of the circuit switches and charges the output capacitance in every clock cycle. This is not always the case, since a change at the input of a circuit does not always propagate to the output.

Therefore, a more accurate formula is

$$P_{dyn} = \frac{1}{2} C_{out} V_{dd}^2 f p, \tag{1.4}$$

where p is the probability that the output of the circuit toggles during a clock cycle (termed transition density in [1]). The term fp expresses the number of output toggles per unit time at the output of the circuit. Therefore, the power consumed in a circuit depends on the toggling of its output; or, more generally, the power consumed by a unit depends on the activity within it.

An even more general equation that considers glitches[1] is

$$P_{dyn} = \frac{A}{2} C_{out} V_{dd}^2 f, \tag{1.5}$$

[1] Not zero-delay transitions.

where A is the average number of transitions in the output of the circuit in one clock cycle. Since A can be larger than one, this formula captures the component of the power dissipation due to glitches.

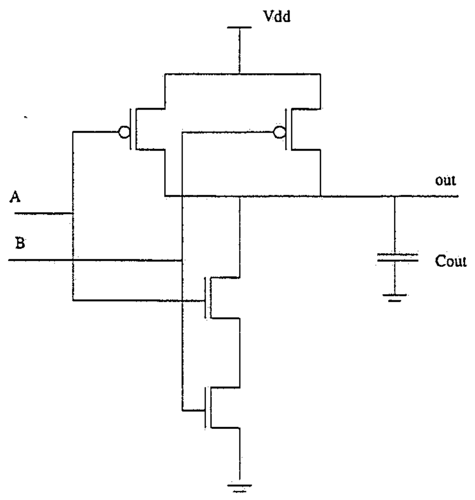


Figure 1.1 CMOS NAND gate.

The short circuit power P_{short} is due to the finite slope of the input waveform. In this case, the pull-up and pull-down transistors are simultaneously on, and a dc current path occurs between the V_{dd} and the ground (Fig. 1.2). The leakage power $P_{leakage}$ is caused by subthreshold currents and by the saturation current of the reverse biased pn junction in an MOSFET transistor. This component will become more important as technology scales down and threshold voltage drops.

Larger and more active units in a microprocessor are expected to consume more power, and should therefore be the target of power minimization. Most of the techniques that have been investigated in the area of low-power design at the logic and the physical level try to minimize the physical capacitance C_{out} and the probability p of switching. Different circuit design techniques, logic and high-level synthesis techniques have been proposed that tackle the problem from the lower levels of the design hierarchy.

An additional problem that designers have to cope with is that low power and high performance are usually two conflicting goals at all levels of the design hierarchy. For example, one common low-power technique for reducing power consumption is to lower the supply voltage. This reduction in supply voltage, however, results in slower circuits and longer clock cycles. Higher frequencies are desirable for high performance, but they increase power consumption.

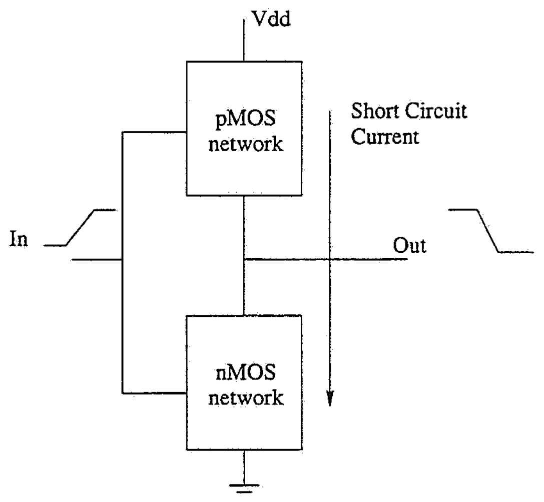


Figure 1.2 Short circuit power component.

Higher activity (and thus utilization) could result in a larger throughput, but also in higher power. The excessive power consumption of today's processors is, in part, the outcome of very high utilization of their components.

The problem of the wasted power caused by unnecessary activity in various parts of the CPU during code execution has traditionally been ignored in code optimization and architecture design. Processor architects and compiler writers are concerned with system performance/throughput and they do little, if anything at all, to eliminate energy/power dissipation at this level. Researchers in the CAD community have started tackling the problem of power minimization through compiler transformations, yet this process is still in its infancy. On the other hand, power dissipation is rapidly becoming the major bottleneck in today's systems integration and reliability. Modern microprocessors are large power consumers: the UltraSPARC-II from Sun consumes 58 W maximum power at 296 MHz, the Pentium Pro Processor consumes 35 W at 200 MHz, and the Alpha 21164PC chip from DEC consumes 32.5 W at 433 MHz.

Table 1.1 shows clearly the power increase for the faster versions of the same processor families. Higher frequencies and larger transistor counts more than offset the lower voltages and smaller devices, and they result in larger power consumption in the newest version in a processor family. This has prompted many manufacturers to design low-power versions of their flagship processors for use in the mobile and multimedia computing industry. Clearly, designing a low-power, high-performance processor is considered an extremely hard problem, which can

Table 1.1 Power trends for current microprocessors.

	DEC 21164	DEC 21164 Higher freq.	Pentium Pro	Pentium II	Ultra SPARCI	Ultra SPARCII
SPECint95	13.3	18.0	8.20	11.9	7.7	12.1
SPECfp95	18.3	27.0	6.21	8.82	11.4	15.5
Average	15.8	22.5	7.21	10.36	9.55	13.8
Freq. (MHz)	433	600	200	300	200	296
λ (μm)	0.35	0.35	0.6	0.35	0.45	0.35
Voltage (V)	2.4	2.4	3.3	2.0	3.3	3.3
Power (W)	32.5	45	28.1	41.4	30 (max)	58 (max)

only be solved if power or energy reduction is a concern from the beginning of the design process and not only an afterthought.

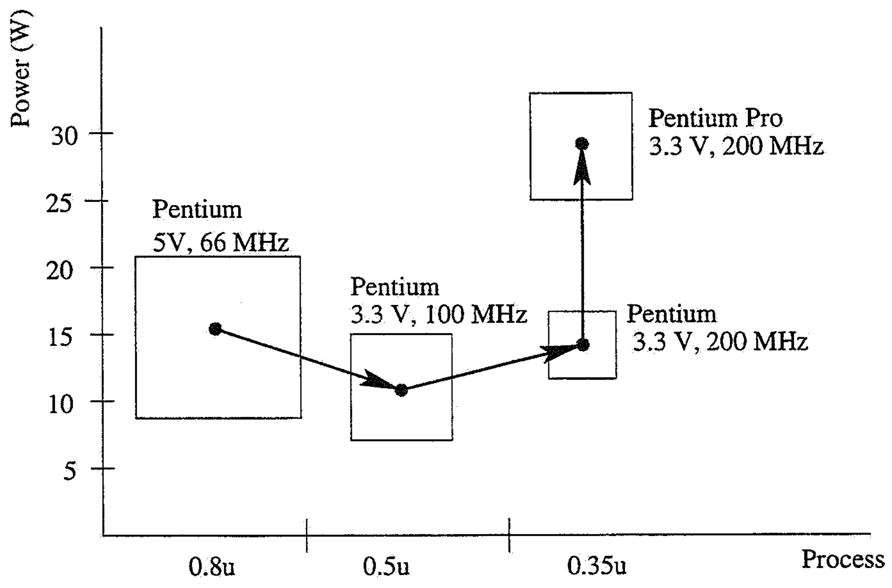


Figure 1.3 Voltage and process scaling do not solve the power problem. The area of each square denotes the relative area of the chip.

Figure 1.3 shows that voltage and process scaling across different processor families can only provide a temporary fix to the power problem [2]. The high-power dissipation is perhaps the first of a number of impeding hurdles that, in the not-too-distant future, will impinge on the rapid growth in uniprocessor performance.

Therefore, there is a demand for low-power design techniques in the whole hierarchy of the design flow. To make these techniques attractive to processor designers, one should not try to optimize for power and neglect the two other important dimensions of the design problem: area and performance. Especially for high-performance microprocessors, the impact on performance should be minimal if a low-power technique is to be deemed acceptable by the design team. It should also be noted that optimizing with respect to only one dimension of the delay, area, and power space usually results in very poorly optimized circuits in the other metrics.

Since both power and performance should be optimized in a processor design, various metrics are needed to quantify the quality of the design. Power (measured in watts) is not a very good metric because it is proportional to the clock frequency. Thus, it can be reduced if we reduce the clock speed. This results in slower computation and not in effective savings. Energy (measured in joules/instruction or SPEC/watts) is better since it is independent of frequency. However, it is propotional to V_{dd}^2, and can be reduced by dropping the supply voltage, and making the processor slower. If we use only this metric, we expect slower processors to be better.

The authors in [3] propose the energy-delay product (measured in joules/SPEC or its inverse $SPEC^2/watts$) as a better metric. To account for changes in the technology scaling factor λ, they improve even better by introducing the $SPEC^2/(Watts * \lambda^2)$ metric, i.e., they include the different feature sizes when they compare between processors. This is because they expect that the energy-delay product will scale according to λ^2. The authors report a somewhat surprising finding: the energy-delay product in a wide spectrum of processors is relatively constant, although energy and delay varied by orders of magnitude. For example, the energy-delay product of the low-power, low-performance R-4600 from MIPS was almost the same as the energy-delay product of the powerful 21164 from DEC.

An increasing number of architecture features have been exposed to the compiler to enhance performance. The advantage of this cooperation is that the compiler can generate code that exploits the characteristics of the machine and avoids expensive stalls. We believe that such schemes can also be applied for power/energy optimization by exposing the memory hierarchy features in the compiler. Our approach, presented in this research, aims at architectural level support for energy reduction, coupled with compiler techniques which take advantage of the new hardware feature.

We are targeting the activity caused by the I-Cache subsystem which is one of the main power consumers in most of today's microprocessors. The on-chip L1 and L2 caches of the 21164 DEC Alpha chip dissipate 25% of the total power of the processor [4]. The bipolar 300-MHz processor in [5] dissipates 50% of its power in the primary caches. In [6], a power analysis of the DLX processor shows that the I-Cache memory and the I-Cache controller are responsible for almost 50% of the total power consumption for some programs. The StrongARM SA-110 processor from DEC, which targets specifically low-power applications, dissipates about 27% power in the I-Cache. Finally, Intel's Pentium Pro, which has a remarkably obscure microarchitecture, dissipates 14% of its power in the Instruction Fetch Unit and the I-Cache.

The justification behind these numbers is that the execution rate of a processor depends critically on the rate at which the instruction stream can be fetched from the I-Cache. The I-Cache should therefore be able to provide the data path of the machine with a continuous stream of instructions, and has therefore very high activity. In addition to that, it has to drive large capacitance wires to the CPU core. What is more, today's caches constitute an ever increasing portion of the die area and the number of transistors of the processor.

1.1 Related Work

The area of power minimization in the architectural and software level is new. A model that views power from the standpoint of the software that executes on a microprocessor and the activity that it causes, rather than from the traditional hardware standpoint has been proposed [7] and tested in different architectures [8], [9], [10]. This methodology attempts to relate the power consumed by a microprocessor to the software that executes on it. This is different from the often used "bottom-up" approach in which power models are built using a layout, gate or RT-level model of each unit, and the power consumption of the whole chip is the sum of the power consumed by each component unit. The authors characterize each instruction of a given microprocessor in terms of the power it dissipates when it is executed. This is the linking bridge between the low-level concept of power dissipation and the high-level concept of software that runs on a microprocessor. It can also provide the means for power minimization through software techniques or through the interaction between software and hardware which has been unexploited thus far.

In [11] and [12] a brief review is presented of compiler techniques that are of interest in the power minimization arena. As expected, standard compiler optimizations, such as loop unrolling, software pipelining, etc., are also beneficial for the reduction of energy since they reduce the running time of the code. The problem of register allocation, which is central in the code generation phase of a compiler, is solved aiming at the minimization of switching activity in [13] and [14]. In [14], the problem of optimizing the energy for the variable allocation in registers and memory is solved using a minimum cost network flow.

In [15], a Gray coding technique for the program counter of a processor is presented which causes less switching in the buses of the CPU. Also, a heuristic for the scheduling of instructions in a dynamic scheduling machine is suggested so that the instruction which causes less switching is selected by the scheduler. An instruction scheduling method that minimizes the number of off-chip accesses for instruction fetch is proposed in [16].

In [17], a new metric for the energy effectiveness of a CPU is introduced: millions of instructions per joule (MIPJ). The authors try to maximize this metric by dynamically varying the system clock through the operating system scheduler. They try to locate periods of idleness of the CPU and reduce the clock frequency during these periods. By using various circuit techniques, such as adiabatic logic or voltage scaling, they can drop the dissipated power when the clock frequency is lower.

More recently, the impact of memory hierarchy in minimizing power consumption, and the exploration of data-reuse so that the power required to read or write data in the memory is reduced is addressed in [18] and [19]. The same authors propose a novel way to organize complex data structures in the memory hierarchy, so that a cost function is minimized. The goal here is to reduce power when complex data structures are manipulated in various ways [20]. Data encoding is another popular technique to reduce the number of switches in large, capacitive buses. In [21], the locality of memory references is exploited to transmit only the offset of a reference with respect to the previous reference and thus reduce the amount of toggling in the buses.

The filter cache [22] tackles the problem of large energy consumption of the L1 caches by adding a small, and thus more energy-efficient cache between the CPU and the L1 caches. Provided that the working set of the program is relatively small, and that the data reuse is large, this "mini" cache can provide the data or instructions of the program and effectively shut down the L1 caches for long periods during program execution. The penalty to be paid is the increased miss rates and, hence, longer average memory access time. Although this might be acceptable for embedded systems for multimedia or mobile applications, it is out of the question for high performance processors. The filter cache delivers an impressive energy reduction of

58% for a 256-byte, direct mapped filter cache, while reducing performance by 21% for a set of multimedia benchmarks [23]. Our approach has a very small performance degradation with respect to the original scheme without the filter cache, and smaller, but still very large, energy gains.

In [24] and [25], a mechanism is described which enables the by-pass of the I-Cache by storing the most frequently accessed instructions in an extra buffer. The solution is limited only in DSP programs, and makes the restrictive (and unrealistic) assumption that all the instructions of a loop will be executed in the first iteration. In [26], the authors propose methods to eliminate the tag comparisons in a cache access when the same cache block is accessed. In [27], the authors demonstrate that some popular hardware techniques that have been proposed to improve the hit rates of caches, such as the victim cache, have a beneficial impact on power as well.

The concept that an intelligent RAM (IRAM) organization [28] is inherently low power is discussed in [29] and [30]. Memory latency improvements have not kept pace with the dramatic performance increase of current microprocessors, and access times are increasingly limiting performance. To attack this problem, researchers have proposed a scheme in which the DRAM is migrated onto the processor itself. This in effect decouples the communication between the CPU and the rest of the system and eliminates the need for time and energy-consuming off-chip memory accesses. However, the power within the CPU increases.

Besides on-chip caches, register files represent a substantial portion of the energy consumption in today's high-performance processors [31]. This trend will probably deteriorate rapidly in the future with wider issue processors and more physical registers. Energy savings can be obtained by replacing a centralized register file with a set of decentralized ones [32]. Those smaller register files have less read and write ports and fewer entries. Tha authors are looking into inter-instruction communication patterns so that communication between instructions of the same group are mostly local. In that case, the scheduler can dispatch every instruction to the cluster where instructions producing its register source files are executed.

Several innovative design techniques that are utilized in ASIC and processor design at IBM are described in [33]. *Pseudo-microcode* is a technique used for decoding instructions, in which only the control signals that change between successive instructions are modified. The idea is that, if a functional path is not required in a cycle, ensure that all control and data paths remain at the previous cycles state to reduce switching. To further reduce the glitches, the authors have designed *transition-once* multiplexers which retain their previous value when not selected and prevent any glitching from propagating from the input to the output when selected.

The unacceptably high levels of power dissipation have prompted the design and manufacturing of a series of processors that target the low-power market. The StrongArm SA-110 processor from DEC dissipates less than 450 mW at a frequency of 160 MHz, with a supply voltage of 1.65 V [34], [35]. The Amulet microprocessor [36] is a self-timed implementation of the ARM architecture and uses specific power-saving features in the cache and the branch prediction mechanism to reduce power. The M*CORE architecture [37] from Motorola has also been designed with low energy consumption in mind and uses techniques like voltage and process scaling, clock gating, and minimization of glitching to achieve this goal.

Almost all manufacturers of high-speed microprocessors use low-power techniques to control excessive power dissipation, without waiving the basic requirement of high performance. Voltage and process scaling are the most successful low-power techniques used so far, contributing about 75% of power reduction in the full chip for Intel's processors [2]. A very useful technique in the microarchitectural level is clock gating, which exploits the idleness caused by system underutilization. Clock gating can eliminate the clock signal propagation and effectively "freeze" a module which is not needed. Aggressive clock gating was extensively used in the StrongARM processor even in the gate level to eliminate spurious signals from propagating in the logic. Other processors, like DEC's Alpha 21264, have a large power dissipation in the clock bus, and they use more sophisticated clocking schemes like the H-tree global clocking scheme [38], along with extra buffers and conditioned clocks. A mere 10-W savings is reported when this clocking scheme is used in 21264.

Dynamic power management techniques at all levels of the design hierarchy are the focus in [39]. Of particular interest is the *OnNow* initiative from Microsoft, which enables OS-directed power management. The operating system (OS) is responsible for handling the global power policy by sensing the status of the device drivers and setting the corresponding devices to *working* or *sleeping.* For that purpose, Microsoft, Toshiba, and Intel have jointly developed the *Advanced Configuration and Power Management Interface (ACPI)*, an OS-independent specification that enables the power management of device drivers and processors through the OS. A monitoring system that is used to collect data from the system resources of a laptop computer and that can be used to perform OS-directed power management is described in [40].

1.2 Thesis Outline and Contribution

In this work we propose and develop a new architecture for high performance processors that targets low-energy execution of programs. We propose various schemes in which the hardware and the compiler cooperate to reduce energy during program execution. This is reminiscent of the methods that are used to enhance the performance of state-of-the-art processors by exposing hardware characteristics to the compiler. Our focus is the memory hierarchy and especially the on-chip caches.

Although there has been an extensive research effort lately to reduce power in the microarchitectural level, there is no work that exploits the flexibility of the compiler and its capacity to produce energy efficient code with collaboration with the hardware. No such work exists in the area of low-power design, even though it is a very mature methodology for high performance design.

Chapter 2 motivates our approach and outlines the solution without going into many details. The addition of an extra cache, dubbed L-Cache from now on, and its use when the thread of control is caught within a loop is demonstrated in the context of a generic RISC architecture.

Chapter 3 details the compiler part of the project, and explains step-by-step the stages of the program transformations done by the compiler. In the first stage, the compiler selects portions of the code that will be stored in the L-Cache using profile data from previous runs. The selection is "static" i.e., it is done during compilation and does not change when the program executes. In later stages, the compiler restructures the program in order to reduce the conflicts in the L-Cache. Finally, it maps the new version of the program in the global address space.

Chapter 4 complements the previous chapter and explains the hardware modifications that are needed for the implementation of our scheme. It is shown that the extra hardware is very simple and straightforward and that the compiler can readily communicate the program reordering information to the hardware.

The energy estimation methodology of the cache is described in Chapter 5. A detailed, transistor-level energy model is developed that considers all the components of an SRAM-based, on-chip cache, and their contribution to the energy dissipation in the cache. This model is used extensively in the experimental evaluation of our design.

Chapter 6 presents the experimental results on energy and performance for a modern processor. Those results also suggest that delay, along with energy, can be reduced if we pursue those optimizations more aggressively.

In Chapter 7 a modified approach is advocated which is more effective for integer benchmarks. It is based on the compiler technology as well, but is simpler than the method described in Chapter 3. Experimental results are given for this method for both energy and performance.

All the previous work utilized compiler technology in conjunction with profile data to guide the L-Cache. In Chapter 8 we investigate "dynamic" techniques in which the selection of the code to be stored in the L-Cache is done at run time, i.e., during program execution. Those techniques do not need compiler intervention, yet they demand extra hardware to keep various statistics for the code execution. Those statistics are then used to guide the placement of the code in the L-Cache. We propose various techniques that use the *branch prediction* hardware of a processor coupled with *confidence estimation* mechanisms.

In Chapter 9 we describe possible extensions of our design, and conclude the thesis.

Chapter 2

MOTIVATION AND APPROACH

From the previous discussion, it follows that the I-Cache system of the processors should be our main focus for drastic power cuts. In order for any such optimization to be attractive, it should not have a large negative impact on performance.

For our discussion, we use the generic RISC model of a scalar processor (Fig. 2.1). The pipeline consists of five stages: the instruction fetch (IF) stage in which the next instruction is read from the I-Cache to the CPU, the instruction decode (ID) stage in which the instruction is decoded and the operands are read from the register file, the execution stage (EX) in which the instructions are executed, the memory stage (MEM) in which the load and store instructions are executed, and finally the write-back stage (WB) in which the results are written to the register file. In case of a miss in any of the caches, the main memory provides the data.

Modern microprocessors have a larger number of stages so that the amount of computation in each stage is smaller. This allows a faster clock and a larger throughput. For example, the R-4000 from MIPS has eight stages because a cache access requires two clock cycles [41]. The high-end, superscalar, superpipelined, processors have typically tens of stages in their pipeline. For illustration purposes, however, our model suffices.

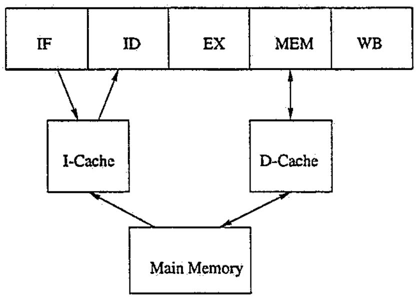


Figure 2.1 Generic pipeline organization.

During a loop execution, the I-Cache unit frequently repeats its previous tasks over and over again: if the thread of control during program execution is caught in a loop, the I-Cache unit fetches the same instructions to the CPU core, and the ID decodes the very same instructions. The problem is that the IF unit does not operate in an efficient way with respect to power consumption, but it only tries to satisfy the demand of the execution units for high throughput, which is achieved through a fast first level (L1) instruction cache and high bandwidth buses between the cache and the CPU core. This approach works for performance, but it unnecessarily performs more work; thus, it dissipates more power than really needed.

To illustrate this point, and also to introduce our modification, let us refer to the following code written in a MIPS-like format:

```
        add   r1,r0,r0      # r1, r2, r3 are registers
        addi  r2,r0,#100    # r0 is always 0
label1: addi  r3,r0,#20
label2: lw    r4,0(r5)      # load a word from memory
        add   r1,r1,r4
        addi  r5,r5,#4
        subi  r3,r3,#1
        bnez  r3,label2     # branch if r3 != 0
        subi  r2,r2,#1
        bnez  r2,label1
```

There are only ten different instructions in this program, but the IF unit fetches $100 * 20 * 5 + 100 * 3 + 2 = 10,302$ instructions. Substantial power gains could be achieved if we could reduce the amount of instructions that the IF unit fetches, and subsequently disable the I-Cache system for all the time that it is not needed. The most usual method for disabling a unit is clock gating, i.e., not allowing the clock ticks to propagate changes to the output of the unit by ANDing them with a control signal. Since a CMOS circuit only consumes power when it switches, clock-gating effectively shuts down the I-Cache.

This is the basic motivation of the architectural support that is proposed in this research. All the instructions that belong to the inner loop can be fetched only the first time the thread of control passes through them. Subsequently, they can be stored in a special internal cache (the L-Cache) which is placed between the I-Cache and the CPU core. Each time the IF unit attempts to fetch an instruction from within the loop, the instruction that resides in this cache can be used instead. In the ideal case, the I-Cache unit can be shut down for the duration of the loop, as it does not need to operate, and its energy dissipation can be saved. Thus, this method exploits the locality in the I-Cache. The new pipeline organization is shown in Fig. 2.2.

A straightforward method to implement this scheme is to place a small level-zero (L0) cache between the CPU and the I-Cache and treat it just as an additional level in the memory hierarchy. This solution deteriorates performance since the smaller extra cache suffers from capacity misses. This is the scheme proposed in [22].

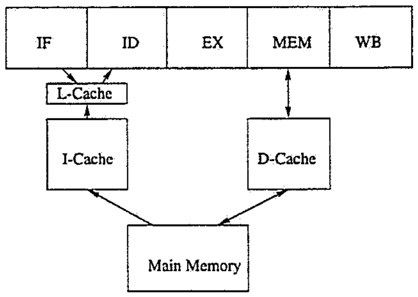


Figure 2.2 Modified pipeline organization.

The only difference between the new architecture and a conventional one is the source that supplies the pipeline with instructions during loop execution, and the subsequent clock gating of the I-Cache. The problem becomes twofold: How can the compiler exploit the new memory hierarchy so that the best candidate loops are cached in the L-Cache? How should the architecture be organized so that the energy savings are maximized. The extra cache also dissipates energy and therefore needs to be much smaller, and more energy efficient than the I-Cache.

The approach advocated in our scheme relies on profile data from previous runs to select the best instructions to be cached. The unit of allocation is the basic block, i.e., an instruction is placed in the L-Cache only if it belongs to a selected basic block.[1] After selection, the compiler lays out the target program so that the selected blocks are placed contiguously before the non placed ones. The main effort of the compiler focuses on placing the selected basic blocks in

[1]A basic block is a sequence of instructions with no transfers in and out except possibly at the beginning or end.

positions so that two blocks that need to be in the cache at the same time do not overlap in the L-Cache.

The compiler maximizes the number of basic blocks that can be placed in the L-Cache by determining their nesting and using their execution profile. The resulting hardware is very simple since most of the task is carried out by the compiler. We eliminate the need for a large L-Cache, thus greatly reducing the power requirements of the extra cache. The experimental results show that the new scheme dissipates 33% of the energy of the original scheme in the I-Cache memory hierarchy subsystem for some of the floating point SPEC95 benchmarks.

2.1 Summary

In this chapter, we explained the motivation behind our research as well as why the I-Cache in modern processors is not used in an energy-efficient way. We proposed the insertion of an extra, smaller cache, the L-Cache, which is managed by the compiler, and which can be used to store frequently executed instructions during code execution. At the same time, the I-Cache can be disabled. In Chapter 3, we describe the compiler techniques needed to exploit the new memory hierarchy.

Chapter 3

COMPILER MODIFICATIONS

In this chapter we explain the algorithm and the data structures involved in selecting and laying out the basic blocks of a program. We should note that the selection of basic blocks to be inserted in the L-Cache is done by the compiler "statically," i.e., during compile time, and not "dynamically" during run time [42]. In a later chapter we discuss "dynamic" schemes for the selection of basic blocks. Those schemes do not need compiler support.

Code placement techniques to reduce cache misses are the subject of the paper by Tomiyama and Yasuura in [43]. The authors use linear programming techniques to minimize the conflicts between basic blocks in the I-Cache, and they give an optimal solution provided that the compiler has no time contsraints to do the placement. Similar techniques for the organization of data in the D-Cache are described in [44]. Both those techniques are only applicable to embedded systems.

The optimization consists of two distinct phases:

- *function inlining*, in which the compiler tries to expose as many basic blocks as possible in frequently executed routines. This step should be done judiciously since function inlining can also create performance and locality problems in the I-Cache.

- *block placement*, the main stage of our method, in which the compiler selects, and then places the selected basic blocks so that the number of blocks that are placed at the same time in the extra cache is maximized. To that effect, it avoids placing two blocks that have been selected to reside at the same time in the L-Cache in the same cache locations.

The reasoning behind our decision to choose a basic block as the basic unit of allocation and not a whole loop can be readily justified by looking in the *control flow graph* (CFG) of a typical loop (Fig. 3.1). In most cases, the loop contains basic blocks which are seldom executed during typical runs. These are blocks that take care of an exception condition or do error handling. If the whole loop was to be allocated in the L-Cache, these basic blocks would occupy space, but they would hardly ever used. They would also disqualify frequently executed blocks from being cached. For example, the inclusion of basic block $B3$ in the L-Cache if the path $B1 \rightarrow B3$ is not executed at all during a typical run will waste allocation space.

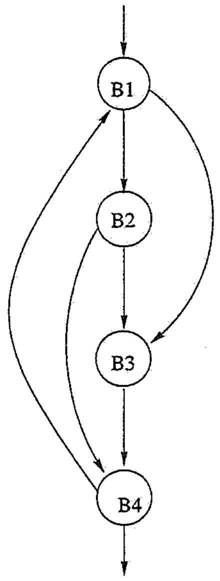


Figure 3.1 CFG of a loop.

3.1 Function Inlining

The first technique we are using is function inlining. Inlining replaces the function call with the body of the called function. It effectively removes the overhead of calling a function, i.e., the call and return instructions as well as the stack manipulation. This is particularly useful for small, frequently called functions. It also enables the effectiveness of a larger set of traditional compiler optimizations like register allocation, copy propagation, common subexpression elimination, constant propagation, etc., which would normally be hindered by function calls.

Function inlining has been studied extensively by researchers in the compiler community. Most of the techniques that are proposed rely on profiling to select the best candidate functions (or function calls) for inlining [45], [46], [47], [48]. Inline expansion can increase the spatial locality and decrease the function call overhead, but it entails some undesirable effects for medium sized caches. The working set might increase with inlining so that it does not fit in the cache anymore, thus reducing the hit rate. In addition, inlining can create register pressure problems and cause the spill of useful variables into memory, if left uncontrolled.

The following code segment illustrates the importance of function inlining in our work:

```
 function A(...)
...
do 100 i=1,n
 B1;          # basic block B1
 call C();    # function call
 B2;          # basic block B2
100 continue;
```

If the code within the loop is executed very frequently, we want to place it within the L-Cache. This is difficult, however, if the linker maps the function C() in a position that overlaps with A() in the L-Cache. This possibility is actually quite large since the L-Cache is small. In that case, our method cannot place the code of function C() in the L-Cache. We will only place B1 and B2 and lose the opportunity for further gains by excluding the body of C(). Actually, our method does not allocate basic blocks in the L-Cache across functions. Hence, each function assumes that, upon entry, the L-Cache is empty. If we inline C(), its body will be placed after B1 and before B2 and the method can be applied.

The net result after inlining is that more frequently executed basic blocks will be exposed within a function. This is important since code placement is only attempted within a function. Code positioning will then become more profitable and the positioning of the functions from the linker will be less important.

We use the weighted *call graph* to describe the representation of the program (Fig. 3.2). A weighted call graph is a directed graph $G = (N, E, init, F_n, F_e)$ where N is a set of nodes, E is a set of edges, *init* is a member of N, F_n is a function $F_n : N \rightarrow [0,1]$, and F_e is a function $F_e : E \rightarrow [0,1]$. Each node n_i in N represents a function in the program, each arc e_{ij} in E represents a static function call from function n_i to function n_j, *init* is the first function in the program, $F_n(n_i)$ is the percentage of clock cycles that is due to the execution of function n_i and all of its successors, and $F_e(e_{ij})$ is the percentage of clock cycles that is due to the execution of

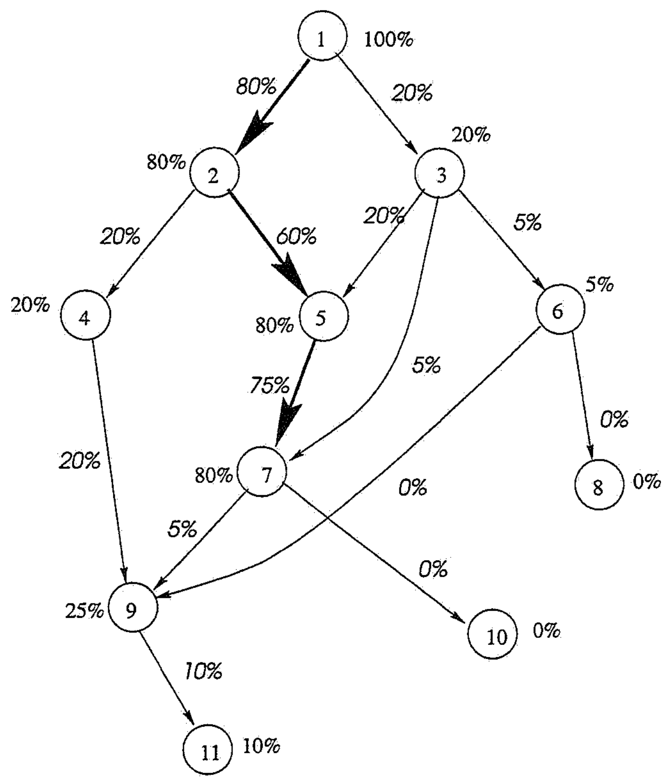


Figure 3.2 Call graph.

n_j and all of its successors when n_j is called from n_i. The following is always true for a node n_i : $F_n(n_i) = \sum_{n_j} F_e(e_{ji})$ for all nodes n_j that are predecessors of n_i in the call graph.

We used the SpeedShop performance tools [49] to obtain profiling data for the time that each function and its descendants took to execute for each function call. The tool inserts additional instrumentation code in the program to capture the execution profile for each function and the call graph is annotated with these data after the run of the program. Using the annotated call graph, our program chooses the appropriate function calls for function inlining and inserts the inlining directives of the MIPSpro compiler in the source code. The compiler does the actual inlining, optimization and code generation.

Our selection heuristic scans the nodes of the call graph and selects the nodes with an execution frequency F_n larger than a threshold T_a. In the example in Fig. 3.2, $T_a = 0.3$. Node n_i needs to have $F_n(n_i) >= 0.3$ to absorb any of its callees. Additionally, the value $F_e(e_{ij})$ should be larger than a threshold T_b for n_j to be absorbed by n_i. The value of T_b is also 0.3 in the example of Fig. 3.2 (i.e., $F_e(e_{ij}) >= 0.3$). Finally, the edge e_{ij} which connects these

Table 3.1 Effect of inlining on execution time.

Benchmark Name	No inline	Inline
102.swim	1	1.19
110.applu	1	0.98
141.apsi	1	1.04
124.m88ksim	1	0.87
129.compress	1	0.83
147.vortex	1	1

two nodes, should have value $F_e(e_{ij})$ at least equal to a threshold T_c when weighed against all the other edges that emanate from n_i. The latter restriction aims at excluding function calls that do not belong to the critical path of the function and are seldom executed. If inlined, they might increase the register pressure and spill useful variables into memory, thus hurting performance. The function calls that correspond to the bold edges in Fig. 3.2 are selected for inlining.

Some experiments are shown in Table 3.1 for some of the SPEC95 benchmarks, in which $T_a = 0.05, T_b = 0.03$, and $T_c = 0.1$, with the additional constraint that only the leaf functions are inlined.[1] The normalized execution time of the inlined code with respect to the original code is shown in the results.

Function inlining is not required for the second part of the compiler optimizations, i.e., for block placement. In some cases, it can improve the effectiveness of block placement by exposing frequently executed code in a function, yet it may create locality problems and hurt performance in other cases.

Function inlining was only applied to the integer benchmarks in our experimenatl evaluation. Those benchmarks have smaller functions which are called frequently, and they favor procedural abstraction. On the other hand, FP benchmarks consist of large functions with a large number of variables which can cause register pressure problems. Integer benchmarks, as opposed to FP benchmarks, consistently performed better when inlining was applied.

[1]Functions that do not contain function calls.

3.2 Block Placement for Maximum L-Cache Utilization

The main step of our algorithm is to position the code in such a way as to maximize the number of basic blocks that are cached in the extra cache. After inlining, the code is laid out so that the most frequently executed basic blocks are placed in the L-Cache. In order to do that, we need to place these blocks contiguously in memory so that they do not overlap. Consider the following code:

```
do 100 i=1, n
  B1;      # basic block
  if (error) then
     error handling;
  B2;      # basic block
100 continue
```

When the code is compiled, the basic blocks B1 and B2 will be separated by the code for the if-statement in the final layout. If the L-Cache size is smaller than the sum of the sizes of B1, B2 and the if-statement, but larger than the sum of the sizes of B1 and B2 only, the blocks B1 and B2 will overlap when stored in the L-Cache. Therefore, we need to place B1 and B2 one after the other and leave the if-statement at the end in the final code layout.

This is usually the case in loops. Blocks that are executed for every iteration are intermingled with blocks that are seldom executed. We identify such cases and move the infrequently executed code away so that the normal flow of control is in a straight-line sequence. This entails the insertion of extra branch and jump instructions to retain the original semantics of the code.

Related optimizations have attracted the attention of the compiler community. In [50] and [51] the authors use *trace scheduling* to position chains of frequently executed paths of the flow graph contiguously in main memory. In addition, they lay out the procedures of the program so that, if one procedure calls another frequently, the two are positioned close to one another in the final code layout. These techniques involve some kind of feedback to the linker which is obtained through profiling.

Hashemi et al. in [52] presented a link-time procedure layout algorithm which uses coloring of the cache blocks to reduce the miss rates in the I-Cache. McFarling [53] considers the structure of the code and avoids placing two procedures that have the same loop nesting in the same cache location.

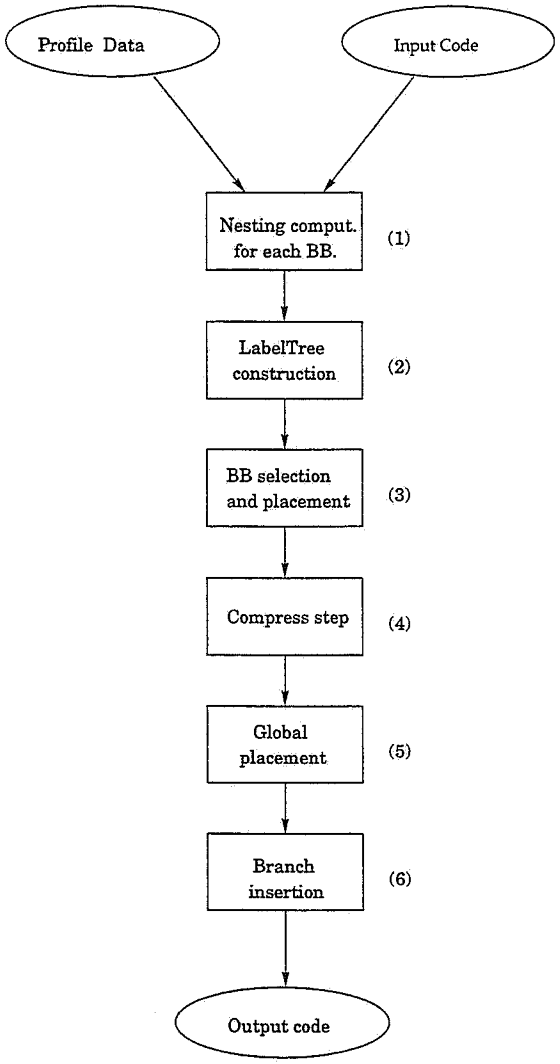


Figure 3.3 Block placement overview.

Our algorithm is outlined in Fig. 3.3. The object code and profile data for the original program are used as input to our tool. The output produced is an equivalent object code in which some of the basic blocks have been reordered and placed in specific memory locations. The following subsections give a detailed description for each of the steps of the method.

3.2.1 First step: Nesting computation

The control flow graph is built for each function of the original program in Step 1. Note that the program can be either the original one or the one that has been created after inlining. A node in the CFG can have none, one, or more than one predecessors, and at most two successors. This is the case when there is a branch instruction at the end of the basic block. We introduce a slight modification in our CFG: although we normally think of a procedure having only one entry, we need to generalize in our case. If there is a function call within a loop of a procedure we need to declare the return from this function call as a new entry to the procedure. The reason for this modification is that we do not want to place basic blocks across procedures. Each procedure, upon entry, will assume that nothing is in the L-Cache from its caller. In other words, basic blocks within a loop which has a function call will not be eligible for caching. This restriction aims at freeing the linker from a possible burden when it maps a function body to the memory space. Some linkers try to map routines that call each other frequently onto contiguous memory addresses to increase the locality of accesses. An inter-function basic block allocation would pose additional constraints to the linker.

Next, in the same step, the tool finds the loops and the nesting for every basic block [54]. Basic blocks within loops that contain function calls are not considered for placement (Fig. 3.4). A $LabelSet(B)$ for every basic block B is the set of loops to which B belongs. If B is not nested, $LabelSet(B) = \{\}$. If B is enclosed in loops L_1, L_2 and L_3, then $LabelSet(B) = \{L_1, L_2, L_3\}$. These are the same sets used in [53]. In Fig. 3.5, an example is given to describe the data structures used and the information produced during the first step of the algorithm. A loop nesting is shown in 3.5(a), the corresponding CFG in 3.5(b), and the LabelSets in 3.5(c).

3.2.2 Second step: LabelTree construction

In Step 2, we construct a directed acyclic graph (DAG) using the LabelSets as follows: the nodes are the different LabelSets found in the previous step. There is an arc between two such LabelSets $< l_1, l_2 >$ if l_1 is a proper subset of l_2 (Fig. 3.6a). Our data structure, dubbed *LabelTree*, is similar, yet it differs from what is used in [53]. First, the nodes refer to basic blocks and not procedures. Second, as the name implies, *LabelTree* is a tree in our case rather than a DAG (see Appendix A for a proof).

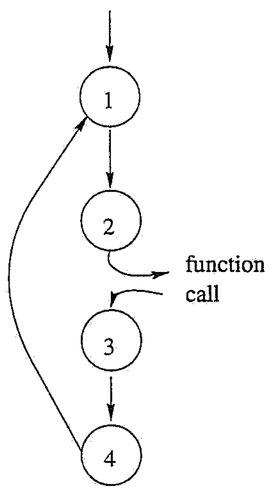


Figure 3.4 Function call within a loop.

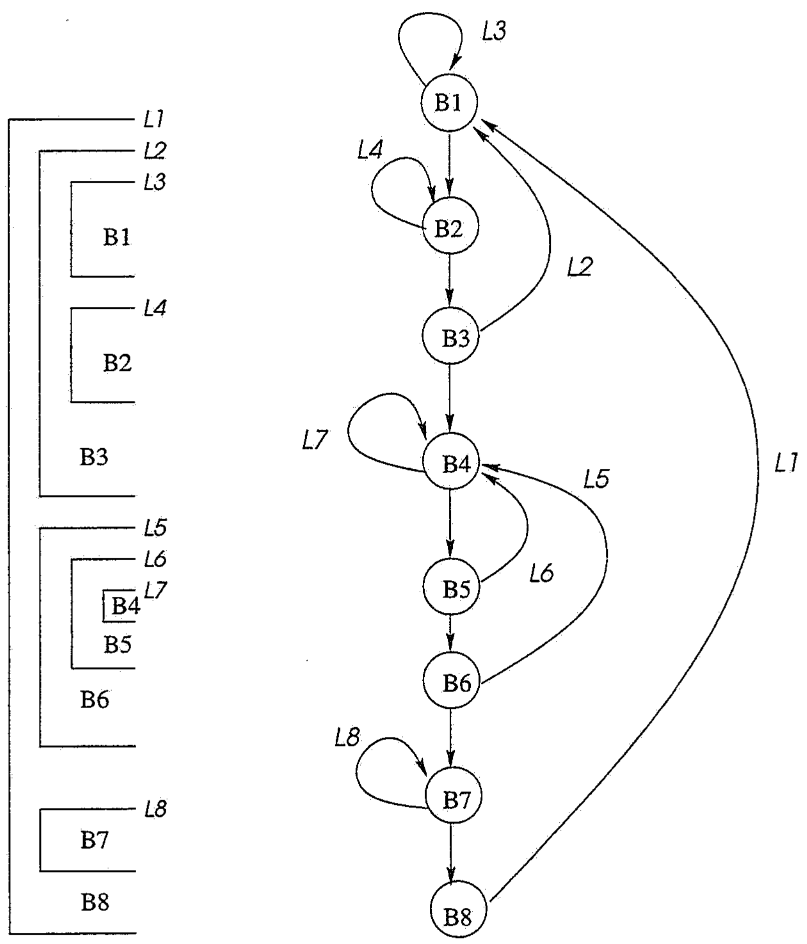
The LabelTree describes the nesting relationship between basic blocks. Basic blocks in the same path in the LabeleTree belong to the same nesting, although in different depths. Basic blocks that are near the leaves of the LabelTree are deeply nested, whereas basic blocks that are near the root are not.

3.2.3 Third step: Basic block selection and placement

Step 3 takes over the main part of our allocation algorithm (Fig. 3.9). A well-known NP-complete problem is that of placing objects with a given value and a weight into a knapsack so that the total value of the placed objects is maximized under the constraint that their weight does not exceed the capacity of the knapsack. We only expect to find a good heuristic which will place the most frequently executed basic blocks in the L-cache provided that their size is smaller than the size of the L-Cache.

The algorithm scans the basic blocks in descending order of execution frequency. Hence, the most important blocks are the first to be considered and have a greater chance to be placed in the L-Cache. For every node in the *LabelTree* we designate a size, which denotes the position in the L-Cache where a basic block of the node should be placed in every step of the algorithm. The size should always be less than or equal to the cache size; otherwise the current basic block cannot be placed in the L-Cache.

The first step is to propagate the effect of the size of the basic block under consideration towards the leaves of the tree rooted at node N (DOWN_TRAV()). Suppose, for example, that

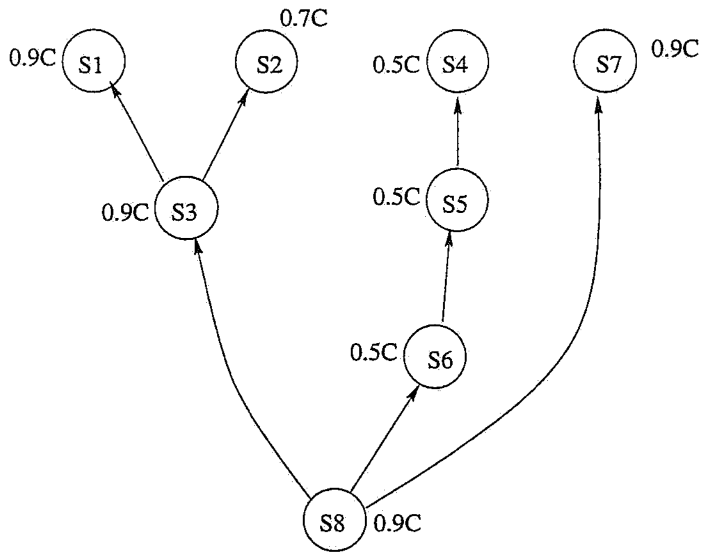


B1: {L1, L2, L3}=S1
B2 : {L1, L2, L4}=S2
B3 : {L1, L2}=S3
B4: {L1, L5, L6, L7} = S4
B5: {L1, L5, L6} = S5
B6: {L1, L5} = S6
B7 : {L1, L8}=S7
B8 : {L1}=S8

(a) (b) (c)

Figure 3.5 First step of block placement.

the current basic block is B_3 in Fig. 3.6a. Both nodes B_1 and B_2 have already been considered and placed in the L-Cache. The size of B_3 added to the $max(size(B_1), size(B_2))$ should not exceed the cache size C. If this is the case, B_3 is placed in the L-Cache. In other words, B_3 will remain in the L-Cache while B_1 and B_2 are executed, and it will not be replaced. This step aims at placing B_3 in a different cache position from both B_1 and B_2. If B_3 overlapped with them, it would have to be fetched from the I-Cache instead, since it would be replaced by B_1 after being executed. This technique maximizes the number of basic blocks that are placed in the cache and avoids conflicts between them.



√	B4 : 0.3C	(0)
√	B1 : 0.6C	(0)
√	B2 : 0.5C	(0)
√	B5 : 0.1C	(0.3C)
√	B7 : 0.8C	(0)
√	B3 : 0.2C	(0.6C)
	B6 : 0.7C	
√	B8 : 0.1C	(0.8C)

(a) (b)

Figure 3.6 LabelTree.

If $max(size(B_1), size(B_2)) + size(B_3) > C$, the placement of B_3 is not possible, and the algorithm continues with the next basic block.

Subsequently, the algorithm calls UP_TRAV() which propagates the effect of the new placement to the outer blocks. This, in effect, reduces the chance of the outer blocks to be placed in the L-Cache, which is not bothering at all, since we are mostly interested for the inner, most frequently executed blocks. In Fig. 3.6a , the annotated *LabelTree* for the example in Fig. 3.5 is given with the final placement of the basic blocks in 3.6b. All the blocks except B_6 are placed in the L-Cache (the positions are in the parentheses and are with respect to the beginning of the L-Cache).

The algorithm is greedy because it tries to accumulate as many important basic blocks as possible in the L-Cache. In the case where the most frequently executed basic blocks are the most deeply nested, the algorithm will succeed in putting all of them in the L-Cache provided that the size of each one is smaller than the cache size.

In practice, we only consider a fraction of the basic blocks of the program, i.e., the ones with a substantial contribution to the execution time. This will speed up the algorithm significantly. We rule out any basic block with execution time less than a user-defined threshold. The complexity of the algorithm is O((number of BBs)$*h$), where h is the height of the *LabelTree.* The maximum *LabelTree* height is 2^d, where d is the depth of the deepest nested loop (usually a small integer).

A basic block will not be selected for placement in algorithm *Allocate()* if any of the following is true:

- It belongs to a library and not to a user function. We follow the convention that only user functions are candidates for placement since they have the tightest and deepest loops.
- The algorithm finds that the basic block was too large to fit in the L-Cache. This can be either because the size of the block is larger than the cache size, or because it cannot fit at the same time with other, more important, basic blocks. The algorithm described in this subsection is used to implement this criterion.
- Its execution frequency is smaller than a threshold, and is thus deemed unimportant.
- It is not nested in a loop. There is no point in placing such a basic block in the L-Cache since it will be executed only once for each invocation of its function.
- Even if its execution is large, its *execution density* might be small. For example, a basic block that is located in a function which is invoked many times might have a large execution frequency, but it might only be executed a few times for every function invocation. We define the execution density of a basic block as the ratio of the number of times it is executed to the number of times that the function in which it belongs is invoked.
- Finally, a very small basic block is not placed in the L-Cache even if it passes all the other requirements. The extra branch instructions that might be needed to link it to its successor basic blocks will be an important overhead in this case.

A basic block is placed in the L-Cache only if it is expected to stay there for a long period of time without getting replaced. This in effect decouples the communication between the I-Cache and the L-Cache and reduces the traffic between them.

3.2.3.1 Example

We refer to Fig. 3.6 to show how the algorithm works. We consider the basic block with the largest contribution in the execution time first, in that case B_4, which belongs to LabelSet

S_4. Basic block B_4 is the first to be considered and can be placed in the L-Cache without any conflict. We set the variable *size* of S_4 equal to the size of B_4, i.e., $0.3C$. We also set the *size* of all the LabelSets between S_4 and the root to $0.3C$.

We continue with B_1, which belongs to S_1. Basic block B_1 can also be placed in the L-Cache since no conflict arises. The *size* variable of S_1, S_3 and S_8 are set to $0.6C$. Next, B_2 is placed in the L-Cache, but the *size* of S_3 and S_8 do not change.

Basic block B_5 is not in a leaf; therefore, we need to use the $DOWN\_TRAV()$ function to propagate the effects of the inclusion of B_5 on its descendants. Since $size(S_4) + size(B_5) = 0.4C < C$, we can place B_5 in the L-Cache. We also set $size(S_5) = size(S_6) = 0.3C + 0.1C = 0.4C$. We continue in this manner, and we only select a basic block if it does not create a conflict with a block that has already been selected. We notice that B_6 cannot be placed in the L-Cache because $size(B_6) + size(S_5) > C$.

3.2.4 Fourth step: Compression

In some cases algorithm *Allocate()* can be improved. In Fig. 3.7, the three basic blocks B_1, B_2, and B_3 which correspond to the three labelsets S_1, S_2, and S_3, respectively, can all be placed contiguously in the L-Cache without any conflict. These basic blocks belong to three different loops, and the sum of their sizes is smaller than C. Step 4 tries to reorder some of the basic blocks in different memory locations so that they are placed more densely in the memory address space. Of course, this reordering should not create conflicts between the basic blocks.

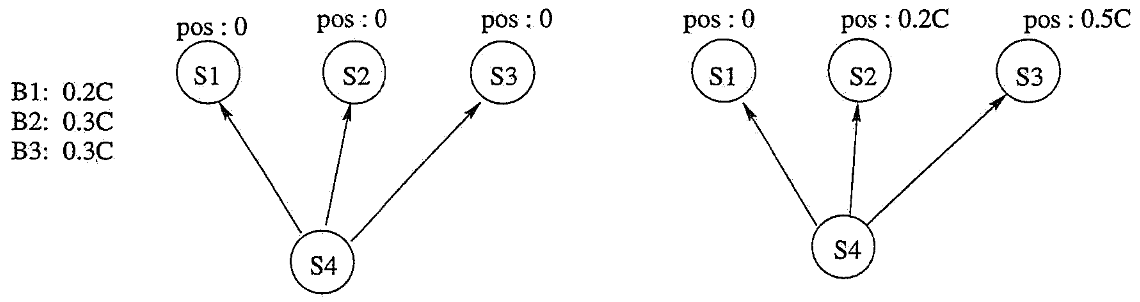


Figure 3.7 Compress step.

The algorithm considers the leaves of a LabelTree and seeks to reposition them so that the space they occupy is minimized. It checks whether these basic blocks can be placed contiguously in the memory address space without any conflict.

3.2.5 Fifth step: Global placement

Step (5) in our methodology is the placement of the basic blocks in the global address space. The algorithm takes as input the placement of the basic blocks with respect to the L-Cache and tries to minimize the necessary space as much as possible.

To implement this algorithm, we assign a set, dubbed *pos_set*, for every line of the L-Cache and we initialize it to the empty set. The set for line i will designate which positions are not available for placement during the course of the algorithm. For example, assume that $pos\_set(12) = \{0, 1, 4\}$. That means that the positions 0,1 and 4 are not available for the L-cache line 12. In other words, a new basic block cannot occupy any of the memory locations $12, 12 + C, 12 + 4C$, because another block already occupies this slot.

Since a basic block will typically occupy more than one cache lines we need to consider more than one such set when we place a basic block. If basic block BB has a size of N instructions and is placed in position P of the L-Cache, we need to consider all the lines: P to $P + N - 1$, i.e., $pos\_set(P) \ldots pos\_set(P + N - 1)$.

For example, assume that we want to place basic block BB in the memory space, given that its position P in the L-Cache is $P = 2$ and that it has three instructions ($N = 3$). Assume also that the L-Cache has six lines with the following position sets:

$pos\_set(0) = \{0, 1, 2, 3, 4\}$
$pos\_set(1) = \{0, 1, 2, 4\}$
$pos\_set(2) = \{0, 4\}$
$pos\_set(3) = \{0, 4\}$
$pos\_set(4) = \{4\}$
$pos\_set(5) = \{\}$

The block will affect the sets $pos\_set(2), pos\_set(3)$, and $pos\_set(4)$. We cannot place the new block in positions 0 and 4; therefore, we need to find a position which is unoccupied in all the three lines. These positions are 1, 2, 5, 6, etc. In order to position the blocks as tightly as possible, we select the smaller of each value each time. Therefore, we choose to place the basic block in position 1, and we modify the position sets as follows: $pos\_set(2) = \{0, 1, 4\}$,

$pos\_set(2) = \{0,1,4\}$, and $pos\_set(2) = \{1,4\}$. The actual new position of the new basic block is: $1 * C + 2$.

The following algorithm implements the global placement heuristic that we described. The input is the position P of every selected BB with respect to the beginning of the L-Cache, and the L-Cache size C. The output is the position of each basic block BB in the address space.

```
/* Position $P$ is an array with the positions with respect to the L-Cache for every $BB$ */
void GLOBAL_PLACE (Position $pos$, CacheSize $C$) {
    initialize the $pos\_set$ to {} for every line in the L-Cache;
    /* place all the basic blocks of the L-Cache first */
    for every basic block $BB$ that is placed in the L-Cache {
            Let $P = pos(BB)$;
            Let $N = size(BB)$;
            Consider all the cache lines from $P$ to $P + N - 1$ and compute:
            $X = min(\cap_{i=P}^{P+N-1}(\overline{pos\_set(i)})$
            $X$ is the new position of $BB$
            insert $X$ to all $pos\_set(i)$ :pos_set(i) = $pos\_set(i) \cup \{X\}, i = P \ldots P + N - 1$
            global position of $BB$ is : $X * C + P$;
    }
    append the rest BB at the end;
}
```

3.2.6 Sixth step: Final layout and branch insertion

Finally, in Step 6, the tool performs the actual layout of the code and restructures the CFG. It starts by placing the basic blocks that were selected in Step 3. Those blocks are placed at the beginning, in the memory locations assigned to them in Step 5. Then, all the other basic blocks are placed contiguously. This arrangement greatly simplifies the hardware of the L-Cache, as we will see in the next section. If needed, branches are placed at the end of the blocks to sustain the functionality of the code. These branches introduce a performance overhead with respect to the initial code. As we will see in the experimental evaluation section, the overhead is usually very small.

In Fig. 3.8, a complete example of the original and the restructured CFG is shown for the code of Fig. 3.5. Blocks B_1 and B_2 will overlap in the L-Cache since B_2 will be executed only when the loop of B_1 exits. On the other hand, if B_3 overlapped with B_2 or B_1, it would miss in every execution of the $B_3 \rightarrow B_1$ loop.

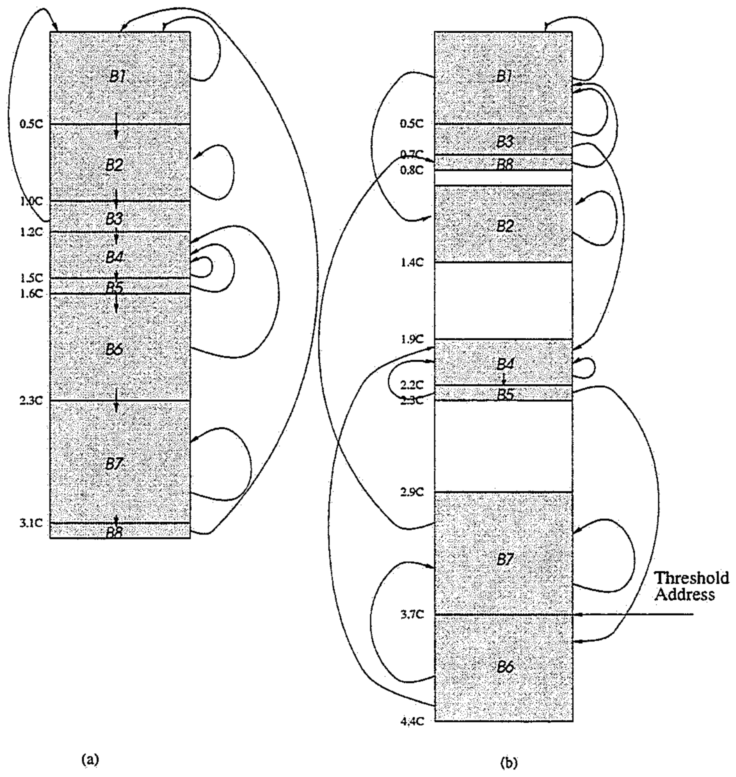


Figure 3.8 CFG restructuring example.

3.3 Summary

In this chapter, we detailed the code transformations done by the compiler to better exploit the L-Cache subsystem. The transformations are applied in three stages: the basic blocks that are going to be stored in the L-Cache are selected using profile data from previous runs of the code. The most frequently executed basic blocks that are nested in a loop are the best candidates in this selection process. Then, our tool seperates the selected code from the non selected one. In the third stage, the selected basic blocks are repositioned so that the conflicts between them are minimized. To do that, we use their nesting relationship as well as profile information. We also explained how function inlining can be used to enhance the effectiveness of the compiler in selecting a larger number of basic blocks. The output of our tool is an equivalent and restructured code which, in cooperation with the hardware, can better exploit the new scheme.

In Chapter 4 we describe the hardware modifications needed to implement the new memory hierarchy scheme.

```
void Allocate(LabelTree T, CacheSize C) /* T root of the LabelTree, C size of L-Cache */
   for every node N in T set size(N) = 0;
   for every basic block BB in the program in descending order of number of times executed
      let N be the node in T that BB corresponds to;  /* N is unique */
      if N is the root next;  /* we don't place BB which are not nested */
      old_size(N) = size(N);
      fit = DOWN_TRAV(N, BB, C);
      if (fit == FALSE) then continue;   /* next basic block */
      UP_TRAV(N);
      put the basic block BB in the L-Cache in position old_size(N);
   end for;
end Allocate;

boolean DOWN_TRAV(TreeNode N, BasicBlock BB, CacheSize C)
   if (DOWN_TRAV_FIRST(N, BB, C) then
      DOWN_TRAV_SEC(N);
      return true;
   else
      return false;
end DOWN_TRAV;

flag = true;
boolean DOWN_TRAV_FIRST(TreeNode N, BasicBlock BB, CacheSize C)
   if (N != NULL and flag) then
      if (size(N) + size(BB) > C) then
         flag = FALSE;
      else
         temp_size(N) = size(N) + size(BB);
      end if;
      for all children of N do
         DOWNTRAV_FIRST(N->child);
   end if;
end DOWN_TRAV_FIRST;
void DOWN_TRAV_SEC(TreeNode N)
   if (N != NULL) then
      size(N) = temp_size(N);
      for all children of N do
         DOWN_TRAV_SEC(N->child);
   end if;
end DOWN_TRAV_SEC;

procedure UP_TRAV(TreeNode )
   while (N->parent) do
      N = N->parent;
      size(N) = max(size(N->child)) among all children;
   end while;
end UP_TRAV;
```

Figure 3.9 Placement algorithm.

Chapter 4

HARDWARE ENHANCEMENTS

In addition to the compiler enhancement, our scheme needs extra hardware for the implementation of the L-Cache scheme. This is shown in Figs. 4.1 and 4.2. The organization and the management of the L-Cache is much simpler than the organization of the I-Cache or the D-Cache.

The organization of the L-Cache itself is depicted in Fig. 4.1. It only needs the data and the tag part, an extra 32-bit comparator for the tag comparison, and a 32-bit multiplexer which drives the data from the L-Cache or the I-Cache to the data path pipeline.

The *PC* is presented to the L-Cache tag at the beginning of the clock cycle. The L-Cache tag will only output its tag if the *blocked_part* signal is on. This signal is generated by the instruction fetch unit (IFU), and its meaning is explained later. In that case, the comparator checks for a match, and if it finds one, it instructs the multiplexer to drive the contents of the L-Cache in the data path. At the same time, the data portion of the L-Cache asserts its output and sends the new instruction to the data path. The I-Cache is disabled for this clock cycle, since the signal *blocked_part* is on.

In the case of an L-Cache miss (*LCache_Hit* is off), the I-Cache controller activates the I-Cache in the next clock cycle and gets the referenced instruction from there. At the same time, this instruction is transfered to the L-Cache. Note that the L-Cache and I-Cache are only accessed sequentially and never in parallel. If *blocked_part* = off, the I-Cache controller activates the I-Cache without waiting for the *LCache_Hit* signal. In this way, the L-Cache can be bypassed without a delay penalty.

Figure 4.2 presents the portion of the IFU in our implementation that generates the signal *blocked_part*. Recall that the compiler has already laid out the code so that the basic blocks

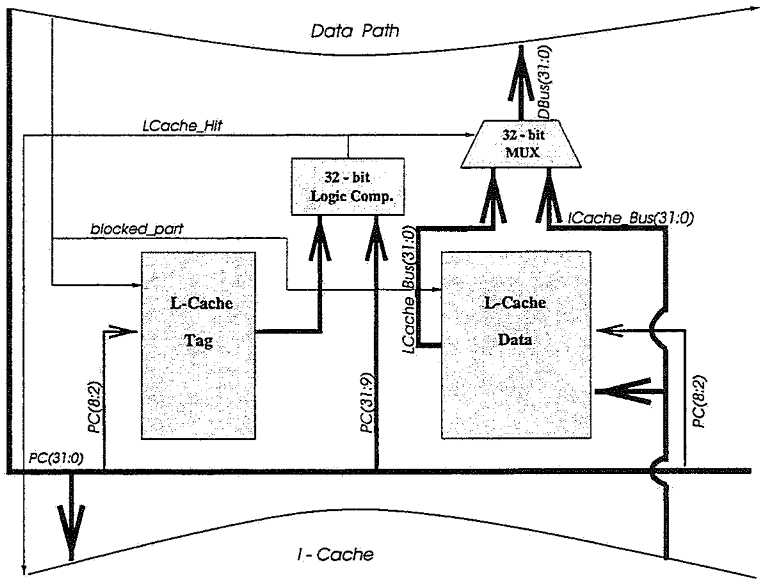


Figure 4.1 L-Cache organization.

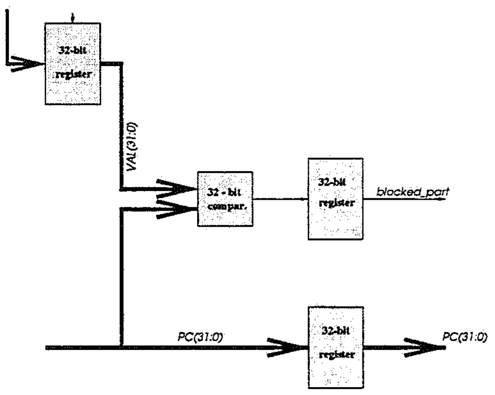


Figure 4.2 Extra hardware in the IFU.

that are destined for the L-Cache are placed before the others. A 32-bit register is used to hold the address of the first non placed block in the main memory layout. If the *PC* has a value less than that address, the 32-bit comparator will set *blocked_part* = on, else this signal will be set to off. This way, the machine can figure out which portion of the code executes with only an extra comparison. Moreover, the comparison is done at the end of the previous clock cycle, so that it does not belong to the critical path of the L-Cache.

This simplification is only possible because of the way that the code has been restructured in the compilation phase. Notice also that if *blocked_part* = on, the L-Cache can still miss: this will happen, for example, when the basic block to be placed in the L-Cache has not been executed before, i.e., the first time the thread of control passes through it. Therefore, the tag portion of the L-Cache is still needed.

Finally, we extend the instruction set architecture (ISA), and we add an instruction called *alloc* which has a J-type MIPS format. It marks the boundary between the selected and the non selected code. This extra instructions is used to store the address of the first non-placed instruction in the 32-bit register, as described in the previous section, and it is the first instruction to be executed upon entry in a procedure. The ID stage of the pipeline will decode the instruction and place the address in the register. There is only one such instruction per function, and its effect on performance is negligible.

In addition, the function call instructions (*jal* and *jalr* in MIPS) are modified to reset this register. This is important since not all functions have basic blocks that are placed in the L-Cache, and the register needs always to have valid contents. If the content of the register is zero, the I-Cache will always be accessed and the L-Cache will be bypassed.

This design is a typical example of how hardware and software can cooperate to achieve a more energy-efficient execution in a high-performance machine. The compiler lays out the code as explained in Chapter 3, and communicates the threshold address to the hardware. The hardware selects the I-Cache or the L-Cache depending on the output code that the compiler generated.

4.1 Summary

In this chapter, we described the additional hardware that is needed to implement the compiler-assisted placement of basic blocks in the L-Cache. The hardware is very simple, since the compiler takes over most of the work in selecting and repositioning the basic blocks. Besides the extra cache, our approach needs a register to store the address of the first non-selected basic block. The PC can determine if the thread of control is in a basic block that has been selected for placement in the L-Cache by comparing that address with the address of the current instruction.

The L-Cache and the I-Cache are only accessed serially and never in parallel. This creates some performance problems, but it is beneficial for energy. In the next chapter we will present the energy models for the SRAM-based caches that were used to estimate the energy dissipation in the memory hierarchy subsystem.

Chapter 5

ENERGY ESTIMATION OF THE CACHES

The most widespread technique for high-level power and energy estimation is *library macro-modelling*, i.e., the characterization of library cells using an analytical expression that captures the important features of the cells with respect to power. A separate model is supplied for each module in the library: adders, multipliers, memories, controllers and so on. These models reflect how the complexity of a particular module affects its power [55] [56].

Accurate energy models are deemed necessary as a prerequisite for design for low-energy caches. Related work has been done in [57], [58], and [59], among others. Most of these proposed techniques utilize models that consider the structure of the cache and use information about the utilization statistics of the cache. Our model is more detailed, and it considers internal banking of the cache so that both the access time and the energy are reduced. The model equations are also very flexible since they allow the estimation of the energy of the cache under different parameters of cache complexity. Moreover, they allow the accurate estimation of energy in a very small amount of time, since the time complexity of the models does not depend on the number of input vectors or the number of accesses in the cache [60].

In this research, we present a detailed, transistor-level energy model of on-chip caches that use SRAM technology. The energy estimation is based on the work by Wilson and Jouppi [61], in which they propose a timing analysis model for SRAM-based caches. Our model uses run-time information of the cache utilization (number of accesses, number of hits, misses, input statistics, etc.) gathered during simulation, as well as complexity and internal cache organization parameters (cache size, block size, associativity, banking, etc.). The utilization parameters

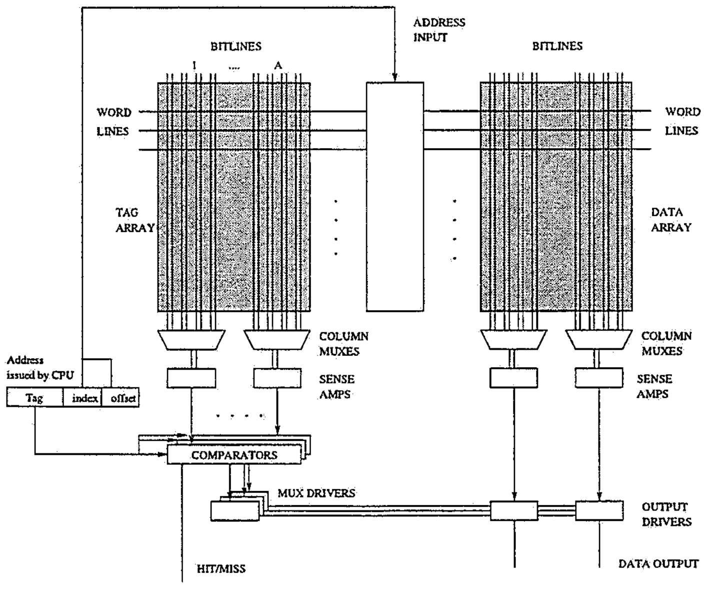


Figure 5.1 Cache model.

are available from the simulation of the R-4400. The cache layout parameters, such as transistor and interconnect physical capacitances, can be obtained from previous layouts, from libraries, or from the final layout of the cache itself.

5.1 Internal Cache Organization

The internal cache organization is shown in Fig. 5.1 [61]. This is a general model of an A-way set associative cache, with size of C bytes and a block size of B bytes. The operation of the cache is now briefly described.

The cache is organized as a collection of $S = \frac{C}{B \times A}$ sets, so that one set contains A blocks, or $B \times A$ bytes. The CPU issues an address to the cache consisting of three parts: the tag, the index and the offset. The index part has length $\log_2(S)$ bits and is used to index the set from which the data will be retrieved. The offset part has length $\log_2(B)$ bits and is used to select the appropriate word within a block to return to the CPU. Finally, the tag part is used to check whether there is a hit or a miss in the cache.

The cache consists of two arrays used to store the tag and the actual data. Each array is organized as a series of rows and columns so that there is one CMOS Static RAM cell at the intersection of a row and column. In Fig. 5.1 we assume that one row in the data array stores a single set. The decoder first selects a row from the tag and data array using the index and offset bits of the CPU address. Each bitline is first precharged high. When the decoder makes the selection, each memory cell in that row pulls down one of its two bitlines, depending on the value of the cell.

A set of sense amplifiers monitors small changes in the bitlines and transforms them into legitimate voltage values. Usually, a sense amplifier is shared among several pairs of bitlines. Extra column multiplexers are used in both arrays to implement this sharing.

The information read from the tag array is compared to the tag bits of the address issued by the CPU. There are A such comparators for an A-associative cache. The result of the comparison is used to drive the output bus with the data that have been read, in the meantime, from the data array.

In most of today's caches, the tag and data arrays are broken row-wise and column-wise so that the time to access the data is reduced. Three new parameters are defined for that purpose for each of the two arrays. The parameter N_{dwl} shows how many times the data array is split vertically resulting in more and shorter wordlines. The parameter N_{dbl} shows how many times the data array is split horizontally resulting in more and shorter bitlines. Finally the parameter N_{spd} indicates how many sets are mapped into a single row. The tag array can be broken independently according to the parameters N_{twl}, N_{tbl}, and N_{tspd}.

Using these organizational parameters, each data subarray has $\frac{8 \times B \times A \times N_{spd}}{N_{dwl}}$ columns and $\frac{C}{B \times A \times N_{dbl} \times N_{spd}}$ rows. The total number of data subarrays is $N_{dbl} \times N_{dwl}$. Similarly, each tag subarray has $\frac{A \times (T+st) \times N_{tspd}}{N_{twl}}$ columns and $\frac{C}{B \times A \times N_{tbl} \times N_{tspd}}$ rows.

5.2 Energy Models for the Cache

Each of the components of the cache is now analyzed in detail with respect to energy dissipation. The gate capacitance of a device or gate x is denoted as $C_{g,x}$ and its junction capacitance as $C_{j,x}$.

5.2.1 Energy dissipation in the decoder

Figure 5.2 shows the decoder architecture. The address from the CPU is partitioned to sets of three bits and is used to drive a set of 3-to-8 decoders. Each of these decoders asserts one out of eight outputs. The NOR gates have as many inputs as the number of 3-to-8 decoders. The outputs of the NOR gates are used to drive the selected wordlines through a wordline buffer. The 3-to-8 decoder can be implemented using eight, three-input NAND gates.

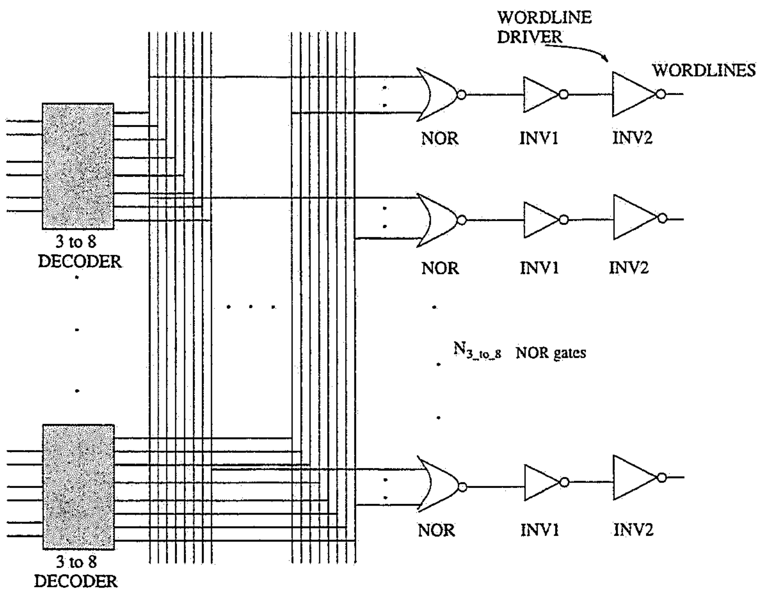


Figure 5.2 Address decoder.

In the actual implementation, the 3-to-8 decoders are actually shared among four subarrays as shown on Fig. 5.3 and are collectively called predecoders. The energy dissipation in the input of the predecoder is given by

$$E_{pred\_input} = \frac{1}{2} \times V_{dd}^2 \times N_{d,inp} \times 2 \times [4 \times \lceil \frac{N_{dwl}N_{dbl}}{4} \rceil \times C_{g,pred\_inp} + 2 \times B \times A \times N_{dbl} \times N_{spd} \times C_{wordmetal}] + \frac{1}{2} \times V_{dd}^2 \times N_{d,inp} \times 2 \times [4 \times \lceil \frac{N_{twl}N_{tbl}}{4} \rceil \times C_{g,pred\_inp} + 2 \times B \times A \times N_{tbl} \times N_{tspd} \times C_{wordmetal}].$$

The value $N_{d,inp}$ is the total number of transitions in the input address (computed during simulation). Note that there are $\lceil \frac{N_{dwl}N_{dbl}}{4} \rceil$ predecoders in the data subarray, and each address input bit or its complement is connected to all eight NAND gates.

For completion, we also give the energy dissipation due to the interconnect capacitance of the metal wires that transfer the address bits to the predecoder inputs. The wire length can be approximated by noting that the total edge length of the data array is approximately $8 \times B \times A \times N_{dbl} \times N_{spd}$. Using Fig. 5.3, we see that the length of the interconnect wire is about one quarter of this length.

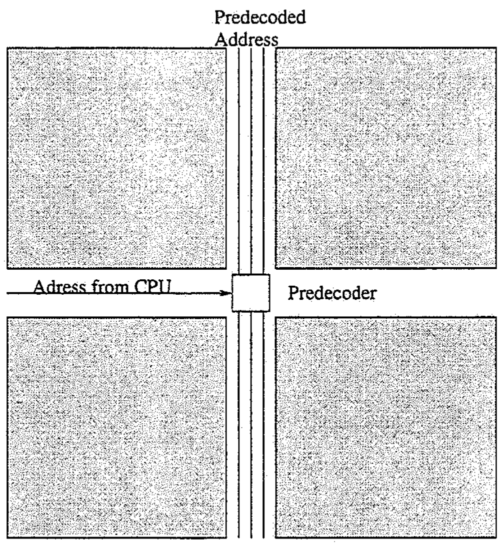


Figure 5.3 Banking of the cache.

The second contribution in the energy dissipation of the decoder comes from the junction capacitances of the NAND gates and the gate capacitances of the NOR gates (Fig. 5.2). It is given by

$$E_{row\_dec} = \frac{1}{2} \times V_{dd}^2 \times N_{acc} \times \{2 \times \lceil \frac{N_{dwl} N_{dbl}}{4} \rceil \times N_{3\_to\_8,data} \times [C_{j,NAND} + \frac{C}{8 \times B \times A \times N_{dbl} \times N_{spd}} \times C_{g,NOR} + \frac{C}{2 \times B \times A \times N_{dbl} \times N_{spd}} \times C_{bitmetal}]\} + \frac{1}{2} \times V_{dd}^2 \times N_{acc} \times 2 \times C_{j,NOR} + \frac{1}{2} \times V_{dd}^2 \times N_{acc} \times \{2 \times \lceil \frac{N_{twl} N_{tbl}}{4} \rceil \times N_{3\_to\_8,tag} \times [C_{j,NAND} + \frac{C}{8 \times B \times A \times N_{tbl} \times N_{tspd}} \times C_{g,NOR} + \frac{C}{2 \times B \times A \times N_{tbl} \times N_{tspd}} \times C_{bitmetal}]\} + \frac{1}{2} \times V_{dd}^2 \times N_{acc} \times 2 \times C_{j,NOR}.$$

The value N_{acc} is the number of accesses in the cache, $N_{3\_to\_8,data}$ is the number of the 3-to-8 decoders for each data subarray, and $C_{bitmetal}$ is the capacitance per bit of the interconnect between the NAND and NOR gates.

The factor $\frac{C}{8 \times B \times A \times N_{dbl} \times N_{spd}}$, which is equal to (Number of rows in the subarray)/8, is used because each NAND gate in the predecoder drives that many NOR gates. Note that the interconnect capacitance is also taken into consideration.

Each of the 3-to-8 decoders modifies only two of its outputs for every access. In addition, only one NOR gate in each array evaluates its output to one, since only one wordline can be selected for each cache access (the term that contains the $C_{j,NOR}$ takes care of that effect). We also multiply by two since one wordline switches from one to zero and another from zero to one in every cache access.

5.2.2 Energy dissipation in the word lines

In every clock cycle, a wordline will be charged and another will be discharged (Fig. 5.4). The energy dissipation in the word lines is given by

$E_{wordline} = V_{dd}^2 \times N_{acc} \times N_{dwl} \times (C_{j,INV1} + C_{g,INV2}) + V_{dd}^2 \times N_{acc} \times (8 \times B \times A \times N_{spd} \times 2 \times C_{g,PASS} + C_{j,INV2} + 8 \times B \times A \times N_{spd} \times C_{wordmetal}) + V_{dd}^2 \times N_{acc} \times N_{twl} \times (C_{j,INV1} + C_{g,INV2}) + V_{dd}^2 \times N_{acc} \times [(T + St) \times N_{tspd} \times 2 \times C_{g,PASS} + C_{j,INV2} + (T + St) \times N_{tspd} \times C_{wordmetal}]$.

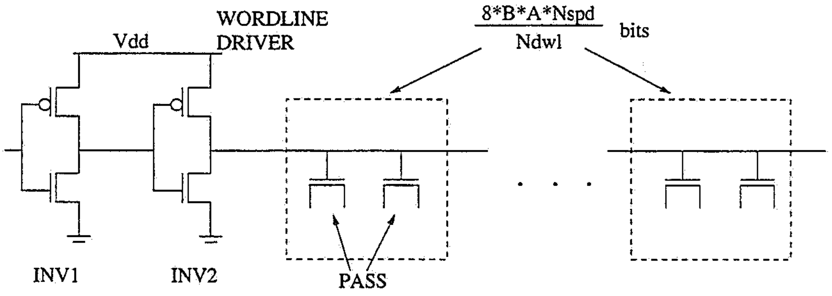


Figure 5.4 Word line architecture.

The length of the tag is T bits, and there are also St bits for the status in each block. For example, the *valid* and the *dirty* bit are status bits. Each data subarray has $2 \times 8 \times B \times A \times N_{spd}$ pass transistors, and each tag subarray has $2 \times (T + St)$ pass transistors.

5.2.3 Energy dissipation in the bit lines

The energy dissipated in the bit lines is due to the precharge phase, as well as to the read or write phase during which one of the two bitlines for every cell is discharged to a logic low value. We assume that the logic swing of the bitline is $\frac{1}{2}V_{dd}$. Figures 5.5 and 5.6 show the precharge and the cell circuit, respectively. The contribution of this component to the energy dissipation is given by

$E_{bitline} = \frac{1}{2}(\frac{1}{2}V_{dd})^2 \times (N_{dbit,pr} \times C_{dbit,pr} + N_{dbit,r} \times C_{dbit,r} + N_{dbit,w} \times C_{dbit,w} + N_{tbit,pr} \times C_{tbit,pr} + N_{tbit,r} \times C_{tbit,r} + N_{tbit,w} \times C_{tbit,w}) + \frac{1}{2} \times (N_{prech\_log} \times N_{acc} \times C_{prech})$.

The following notations have been used:

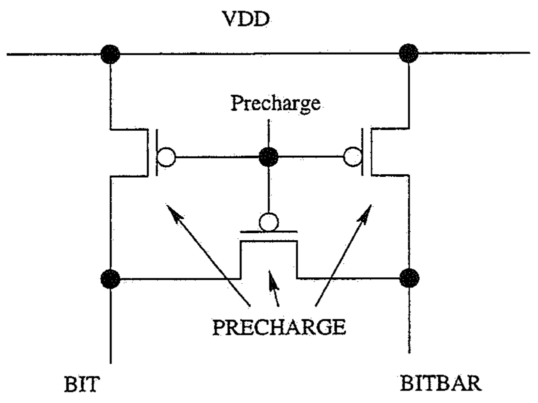


Figure 5.5 Bit line precharge circuit.

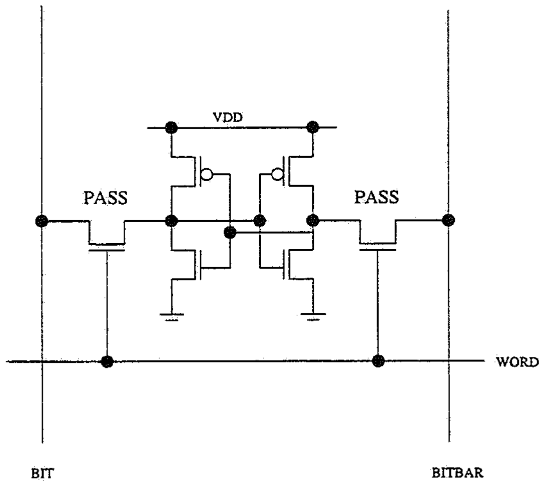


Figure 5.6 Memory cell.

- $N_{dbit,pr} = N_{acc} \times 2 \times 8 \times B \times A \times N_{spd} \times \frac{1}{2}$ is the total number of precharges in all the bitlines in the data array. We assume that half of them switch to a logic high during precharge, since only that half has switced to logic low in the previous clock cycle.

- $N_{dbit,r} = N_{acc} \times 2 \times 8 \times B \times A \times N_{spd} \times \frac{1}{2}$ is the total number of read transitions in the bit lines.

- $N_{dbit,w} = N_{whit} \times (St + W_{averg,data\_write}) + \frac{1}{2} \times N_{rmiss} \times (2 \times 8 \times B \times A \times N_{spd})$ is the total number of write transitions in the bit lines. N_{whit} is the total number of write hits in the cache, and N_{rmiss} the total number of read misses.

- $N_{tbit,pr}, N_{tbit,r}, and N_{tbit,w}$ are defined similarly for the tag array.

- $N_{prech\_log} = 8 \times B \times A \times N_{spd} \times N_{dbl} + A \times (T + St) \times N_{tspd} \times N_{tbl}$ is the number of bitline precharge circuits in the cache.

- $C_{dbit,pr} = C_{dbit,r} = C_{dbit,w} = \frac{C}{B \times A \times N_{dbl} \times N_{spd}} \times (\frac{1}{2} \times C_{j,PASS} + C_{bitmetal}) + 2 \times C_{j,PRECH} + C_{j,MUX}$ is the switched capacitance for each precharge (or read or write) transition. The junction capacitance of the pass transistor is multiplied by two since it is shared between two vertically adjacent cells. The $C_{j,MUX}$ is the junction capacitance of the column multiplexer at the other end of the bitline.

5.2.4 Energy dissipation in the comparators

Each one of the A comparators for an A-associative cache is designed using dynamic logic (Fig. 5.7), and it has a width of T bits.

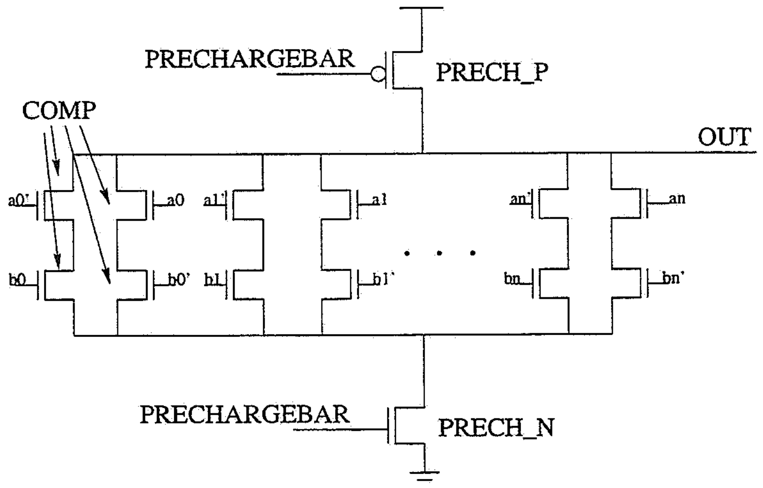


Figure 5.7 Comparator.

The energy dissipation in the A comparators is given by

$E_{comp} = \frac{1}{2} \times V_{dd}^2 \times N_{acc} \times A \times (C_{j,PRECH\_P} + C_{g,MUX\_DRV}) + \frac{1}{2} \times V_{dd}^2 \times N_{misses} \times A \times [T \times 2 \times C_{j,COMP} + C_{j,PRECH\_N} + T \times C_{j,COMP} + C_{g,MUX\_DRV}] + \frac{1}{2} \times V_{dd}^2 \times N_{hits} \times (A - 1) \times (T \times 2 \times C_{j,COMP} + C_{j,PRECH\_N} + T \times C_{j,COMP} + C_{g,MUX\_DRV}] + \frac{1}{2} \times V_{dd}^2 \times N_{hits} \times [T \times C_{j,COMP} + T \times 2 \times C_{j,COMP}] + \frac{1}{2} \times V_{dd}^2 \times N_{tag,inp} \times A \times 2 \times T \times C_{g,COMP}$.

The value $N_{tag,inp}$ is the number of switches in the input of the evaluation transistors. We assume that in case of a miss, half of the paths to the ground will be on and half will be off.

When a path is on, we have to multiply the junction capacitance of a transistor by two to approximate the capacitance switched. In case of a hit, only one out of the A comparators evaluates to one.

5.2.5 Energy dissipation in the multiplexer drivers

Fig. 5.8 shows the context of the multiplexor drivers in the cache systems. Each multiplexor is responsible for controlling the multiplexing of the $8 \times B$ bits from each cache block onto the data bus that reads out b_0 bits towards the CPU or the main memory. There is one such multiplexer driver for each of the A comparators in an A-way set associative cache. The implementation of a multiplexer is shown in Fig. 5.9

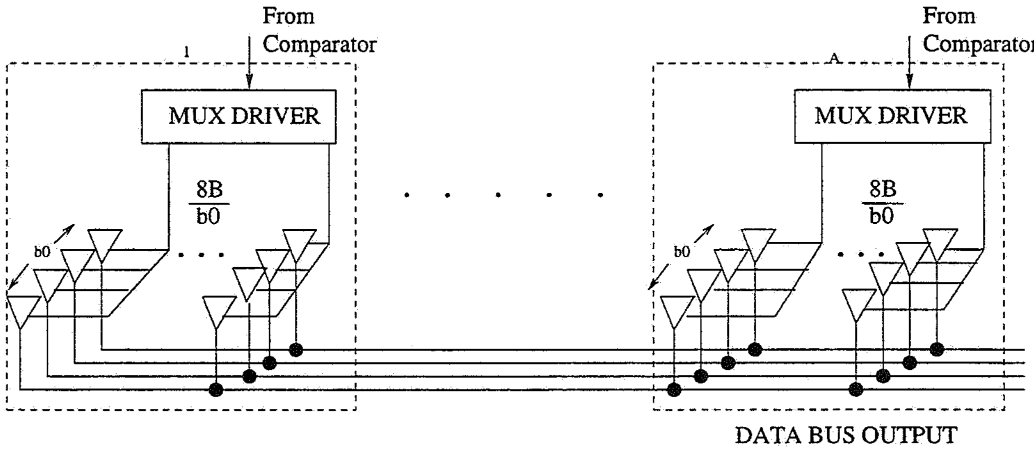


Figure 5.8 Data bus output for multiplexer drivers.

The energy dissipation in the multiplexer drivers is given by

$$E_{mux\_driver} = \frac{1}{2} \times V_{dd}^2 \times N_{hits} \times (\frac{8 \times B}{b_0} \times C_{g,NOR} + C_{j,INV}) + \frac{1}{2} \times V_{dd}^2 \times N_{hits} \times (C_{g,DRV\_INV} + C_{j,NOR}) + \frac{1}{2} \times V_{dd}^2 \times N_{hits} \times (b_0 \times C_{g,OUTDRV}, + C_{j,DRV\_INV} + 4 \times B \times A \times N_{spd} \times N_{dbl} \times C_{wordmetal}).$$

5.2.6 Energy dissipation in the output bus

The energy dissipation in the output busses towards the CPU and the main memory is given by

$$E_{output} = \frac{1}{2} \times V_{dd}^2 (N_{out,a2m} \times C_{out,a2m} + N_{out,d2m} \times C_{out,d2m} + N_{out,d2c} \times C_{out,d2c}).$$

The following notation has been used:

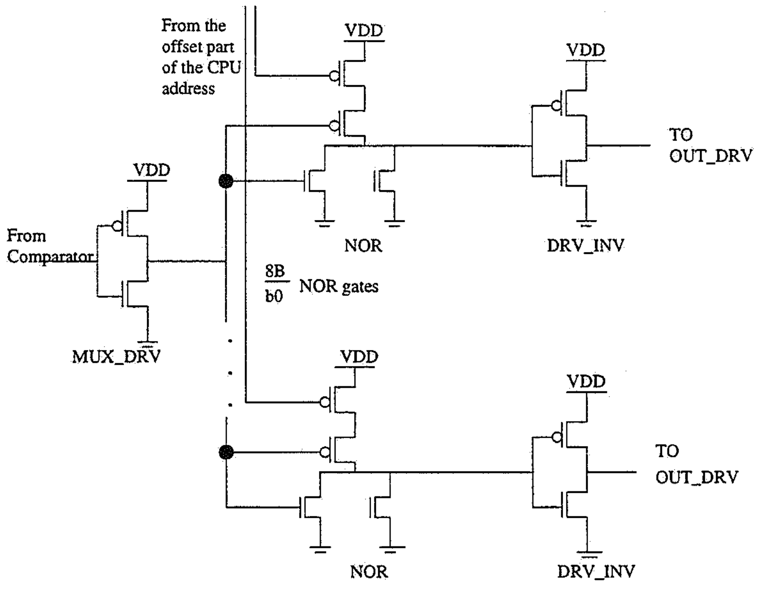


Figure 5.9 One of the A multiplexer drivers.

- $N_{out,a2m} = \frac{1}{2} \times (N_{rmiss} + N_{whit} + N_{wmiss}) \times W_{addr\_bus}$, is approximately the total number of zero-to-one transitions in the address bus from the cache to the next level of memory (L2 cache or main memory). The address bus switches in case of a read miss, in case of a write hit (to write the data to the main memory), and in case of a write miss (to get the data from the main memory). The address bus has W_{addr_bus} bits length.

- $N_{out,d2m} = \frac{1}{2} \times (N_{whit} + N_{wmiss}) \times W_{data\_bus}$, is approximately the total number of zero-to-one transitions in the data bus from the cache to the next level of memory. The data bus switches each time that data needs to be written in the memory.

- $N_{out,d2c} = \frac{1}{2} \times (N_{reads} \times W_{avg\_read})$ is approximately the total number of zero-to-one transitions in the data bus from the cache to the CPU. The factor W_{avg_read} is the average number of bits read from the cache.

- The capacitances for off-chip transfers are set equal to 20 pF, while the capacitances for on-chip transfers are set equal to 0.5 pF [61].

5.3 Energy Distribution in the Cache

The models presented in the previous section were used in conjunction with a simulator of the MIPS2 architecture to obtain realistic estimation of the energy consumption in on-chip caches. The gate and junction capacitances were obtained from [61] for a $0.8-\mu$m process.

The number of accesses in the cache, number of hits, misses, as well as the switching activity in the CPU address were computed by simulation of a set of SPEC95 benchmarks. The following benchmarks were simulated: *101.tomcatv, 102.swim, 103.su2cor, 104.hydro2d, 129.compress, 130.li,* and *134.perl* (described in Table 5.1). Figs. 5.10 and 5.11 show the distribution of energy consumption on the various components of our cache model for two cases.

Table 5.1 Statistics for the 32-kbyte I-Cache utilization.

Benchmark	*Accesses in billions*	*Hit rate*	*Energy in joules*
tomcatv	4.79	99.95%	24.98
swim	0.39	99.99%	2.017
su2cor	10.13	99.97%	52.85
hydro2d	4.5	99.90%	23.50
compress	0.037	99.99%	0.20
li	0.21	99.99%	1.08
perl	2.56	98.76%	13.45

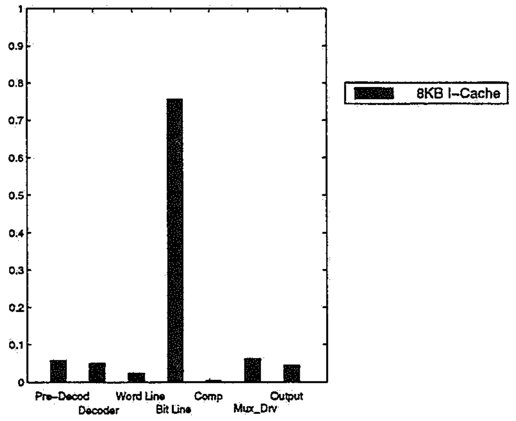


Figure 5.10 Distribution of energy dissipation in a direct mapped, 8-kbyte I-Cache with a block size of 16 bytes.

The bit lines are by far the most energy consuming part of the cache. This is mainly due to the very large capacitance of the precharge transistors, as well as to the length of the bitlines.

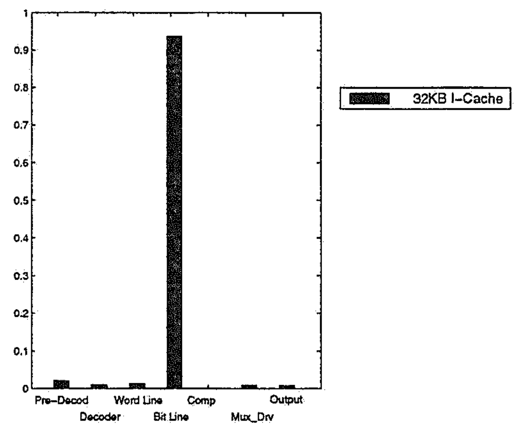


Figure 5.11 Distribution of energy dissipation in a direct mapped, 32-kbyte I-Cache with a block size of 32 bytes.

In every clock cycle, half of the bitlines will precharge, and then half will discharge to logic zero.

Several circuit techniques have been proposed and implemented to reduce the energy dissipated in an SRAM-based cache, especially in the bitlines. Low swing bitlines and bitline isolation are standard techniques that attempt to restrict the switching of the bitlines to a small voltage range. With this arrangement, a small differential voltage on the bitlines is sufficient to trigger the full transition on the sense amplifier outputs [62]. A voltage swing of a few hundreds of millivolts is usually sufficient to trigger the sense amplifiers.

The energy dissipation of the internal cell, the column decoders and sense amplifiers was negligible and it was not taken into consideration.

5.4 Summary

In this chapter, we presented a detailed, transistor-level model for the estimation of energy in SRAM-based caches. These models will be used in the next chapter to evaluate the energy gains in our scheme. We presented a detailed model for each of the components of the caches: the address decoder, wordline, bitline, comparators, multiplexer drivers, and output buses. Energy consumption is a function of the cache complexity (size, blocks size, associativity), its internal organization (banking), its utilization statistics (number of accesses, input statistics, etc.), and the capacitance of the interconneccts and the transistors. Experimental evaluation showed that the bitline and the precharge circuit dominate the energy dissipation of the on-chip caches.

In Chapter 6, we present extensive experimental evidence that proves the feasibility of the L-Cache scheme in the context of a modern high-performance processor.

Chapter 6

EXPERIMENTAL EVALUATION

In this section we describe the experimental results on energy savings and performance effects of applying the proposed techniques to the SPEC95, and we compare our method with the filter cache method (i.e., only a L0 cache without any compiler support) [63]. We first outline the experimental setup.

6.1 Simulator Environment

We evaluated the effectiveness of our software/hardware enhancements by examining the energy savings on a set of SPEC95 benchmarks (Table 6.1). We determined whether the bechmark programs are amenable to these modifications and what is the potential for energy savings in them. The benchmarks were compiled with the MIPSpro compiler using the -O2 optimization flag. Hence, we enabled all the traditional optimizations but we disabled any interprocedural analysis and inlining. That was necessary in order to test our own inlining heuristic.

To gauge the effect of our L-Cache in the context of a realistic processor operation, we simulated the MIPS2 ISA using the *MINT* [64] and the *SpeedShop* [49] tool suites. MINT is a software package for instrumenting and simulating binaries on a MIPS machine. We built a MIPS2 simulator on top of MINT which accurately reflects the execution profile of the R-4400 processor. Table 6.2 shows the latencies (in clock cycles) of the functional units of our simulator based on [41], and Table 6.3 describes the memory subsystem configuration as (cache size / block size / associativity / cycle time / latency to L2 cache in clock cycles / transfer bandwidth in bytes per clock cycles from the L2 Cache). Both I-Cache and D-Cache are banked both row-wise and column-wise to reduce the access time and the energy per access [61]. We use the

Table 6.1 SPEC95 benchmarks description.

Benchmark	*Description*
tomcatv	Mesh generation program
swim	Shallow water model with 1024 $\times$ 1024 grid
su2cor	Computation of the masses of elementary particles in quantum physics
hydro2d	Solution of hydrodynamical Navier-Stokes equations
mgrid	Computation of a 3-D potential field in electromagnetics
applu	Solution of five coupled parabolic/elliptic PDEs
turb3d	Simulation of isotropic, homogeneous turbulence in a cube
apsi	Computation of temperature variations
wave5	Solution of Maxwell's equation
go	Computer game that uses artificial intelligence
m88ksim	Simulator of the 88100 microprocessor
compress	File compression using the Lempel-Ziv coding
li	Lisp interpreter
perl	Perl interpreter
vortex	Object-oriented DB transaction program

tool *cacti*, described in [61], to estimate the access time of the on-chip caches, as well as the optimal banking that minimizes the access time. The tool provides the optimal values for the parameters N_{dwl}, N_{dbl}, and N_{spd} for the data array, and for the corresponding parameters for the tag array.

We considered a filter cache of 256 and 512 bytes, and block size that varied between 8 and 32 bytes. The L-Cache was also 256 and 512 bytes, and always had a block size of 4 bytes, i.e. the size of a MIPS instruction. A larger block size does not significantly increase the hit rate of the L-Cache, whereas it negatively affects the dissipated energy per access. The L2 unified cache is off-chip and its energy dissipation is not modeled.

We also experimented with different scenarios for the user-given thresholds that guide the basic block selection and placement in the L-Cache (Table 6.4). A more aggressive scenario results in larger energy gains at the expense of larger performance degradation. A frequency threshold of 0.01%, for example, will force the tool to mark for placement only basic blocks that have an execution time of at least 0.01% of the total execution time of the program. A size threshold of 10 instructions will force the tool to mark only the basic blocks that have at least 10 instructions, and so on.

Table 6.2 Functional units latency.

Resource	Latency
Integer ALU	1
Integer MULT	12
Integer DIV	76
FP ADD/SUB	4
FP MULT (single)	7
FP MULT (double)	8
FP DIV (single)	23
FP DIV (double)	36
FP SQRT (single)	54
FP SQRT (double)	112
FP CONVERT	2–6

Table 6.3 Memory subsystem configuration in the base machine.

Parameter	Configuration
L1 I-Cache	32KB/32/1/1/4/8
L1 D-Cache	32KB/32/2/1/4/8

Our experimental methodology was as follows. First, we ran the benchmarks to collect the profile data. The data were used to drive the inline and the block placement heuristics. The tool, along with the restructuring of the body of the program, selected various statistics regarding the quality of the generated code. SpeedShop was used for profiling and the MIPSpro compiler was used for compilation and code optimization. The actual simulation was done using MINT. Function inlining was used only for the SPECint95 benchmarks. Through experimentation, we found out that inlining is more beneficial when only leaf functions are absorbed; hence, we limit our inlining procedure to consider only leaf functions.

The next section delineates the energy and delay results for the filter and the L-Cache under the different scenaria described earlier. It also looks into potential performance gains using a faster clock, equal to the access time of a smaller I-Cache. We show that using our method, energy as well as delay can be simultaneously reduced when the compiler assumes the role of statically allocating instructions to the L-Cache.

Table 6.4 User-given thresholds in the L-Cache experiments.

Experiments	*Frequency Thres.*		*Size Thres.*		*Exec. Density Thres.*	
	FP	INT	FP	INT	FP	INT
Aggressive (a)	0.01%	0.01%	5	5	5	5
Less Aggressive (b)	0.5%	0.5%	10	5	10	5
Moderate (c)	1%	1%	20	5	20	5

Table 6.5 Percentage of basic blocks placed in the L-Cache (only user-level code).

Benchmark	*L-Cache size*					
	32 instr.	64 instr.	128 instr.	256 instr.	512 instr.	1024 instr.
tomcatv	6.08%	9.94%	14.92%	15.47%	16.02%	16.02%
swim	1.03%	3.77%	5.48%	5.82%	5.82%	5.82%
su2cor	1.15%	2.43%	3.58%	3.83%	3.90%	3.90%
hydro2d	3.21%	6.76%	9.34%	10.02&	10.07%	10.07%
mgrid	4.61%	7.24%	9.43%	10.31%	10.31%	10.31%
applu	2.48%	4.20%	10.12%	12.42%	13.47%	13.47%
turb3d	0.27%	0.81%	1.44%	1.71%	1.98%	1.98%
apsi	0.48%	1.12%	1.79%	2.00%	2.10%	2.10%
wave5	1.65%	2.97%	4.09%	4.38%	4.42%	4.42%
m88ksim	0.15%	0.15%	0.15%	0.15%	0.15%	0.15%
li	0.06%	0.06%	0.06%	0.06%	0.06%	0.06%
compress	1.11%	1.11%	1.11%	1.11%	1.11%	1.11%

6.2 Results

Using the configuration of Tables 6.2 and 6.3, we performed an analysis across different organizations of the filter cache and the L-Cache. Table 6.5 shows the percentage of static basic blocks that are selected to be cached in the L-Cache in the course of program execution. On average, less than 7% of all the basic blocks in the user-level code are selected in the SPECfp95 bechmarks.

The percentage of dynamic instructions that cause the machine to access the L-Cache in the course of program execution is shown in Table 6.6. This access may result in a hit or a miss. For the FP benchmarks, most of the basic blocks have a negligible execution frequency or are too large to fit in the cache (for smaller caches only). Integer benchmarks have a large number of basic blocks with small execution frequency and, in addition, many blocks that are

Table 6.6 L-Cache utilization statistics: percentage of instructions that cause an access to the L-Cache.

Benchmark	*L-Cache size*					
	32 instr.	64 instr.	128 instr.	256 instr.	512 instr.	1024 instr.
tomcatv	15.25%	40.17%	90.05%	99.84%	99.94%	99.94%
swim	0.10%	72.71%	91.68%	99.93%	99.93%	99.93%
su2cor	10.05%	53.28%	66.92%	98.17%	99.22%	99.22%
hydro2d	22.61%	39.36%	39.36%	94.48%	94.49%	94.49%
mgrid	5.39%	7.31%	97.79%	99.38%	99.38%	99.38%
applu	12.62%	43.12%	66.87%	73.01%	73.35%	84.28%
turb3d	3.89%	50.90%	74.44%	74.87%	77.90%	77.90%
apsi	10.50%	34.83%	79.65%	83.84%	86.76%	86.76%
wave5	25.30%	44.54%	70.35%	94.11%	96.16%	96.16%
m88ksim	46.16%	46.16%	46.16%	46.16%	46.16%	46.16%
compress	15.49%	15.49%	15.49%	15.49%	15.49%	15.49%
li	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%

not nested in a loop. Blocks with small execution density are also a problem for some integer benchmarks.

This percentage is high for all the SPECfp95 benchmarks, reflecting the efficacy of our approach for these programs. As expected, a larger L-Cache is more succesful in storing basic blocks and therefore in disabling the I-Cache for a larger period of time. In some cases, even a small L-Cache is capable of effectively shutting-down the I-Cache for the duration of the program execution. The law of diminishing returns applies here as well, since a very large L-Cache (1024 instructions) is usually as succesful as smaller ones. In most cases, a 256 instruction L-Cache approximates the performance of an infinite size L-Cache.

On the other hand, most integer benchmarks do not have a large number of basic blocks that can be cached in the L-Cache. They are also insensitive to the cache size variation, which is to be expected since the basic blocks of integer programs are generally small. Most of the basic blocks of the SPECint95 benchmarks are not nested, or they are nested within a loop that contains a function call; hence, they cannot be included in the L-Cache. Integer programs with complex control flow graphs, such as interpreters, compilers and so on, have a large number of different paths in the CFG. These benchmarks have the worst behavior. Benchmarks with a more regular structure (compression programs, simulators, etc.) are better suited to our algorithm.

Tables 6.7 and 6.8 show the energy gains in the I-Cache subsystem for the three different L-Cache and filter cache configurations. The numbers are normalized with respect to the energy dissipation of the original scheme. The energy in the modified configurations is due to both the I-Cache and L-Cache (or filter cache for the three last columns). A result less than one is desirable since it denotes an improvement in energy or delay with respect to the original scheme. Even a small delay penalty can be acceptable as long as the energy reduction is substantial.

The clock period was set equal to the access time of the D-Cache, which is the critical path in many high performance processors. A 32-kbyte, two-way set associative D-Cache, with a 32-byte block size has an access time of 11.4 ns using a 0.8 μm feature size (as found using *cacti*).

Table 6.7 Normalized energy relative to the base machine for 256-byte extra cache.

Benchmark	*256 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.565	0.622	0.622	0.16	0.154	0.193
swim	0.312	0.318	0.347	0.131	0.135	0.189
su2cor	0.498	0.511	0.511	0.192	0.169	0.205
hydro2d	0.418	0.438	0.493	0.122	0.134	0.187
go	0.952	0.974	0.986	0.466	0.324	0.300
compress95	0.873	0.875	0.875	0.448	0.328	0.307
li	1	1	1	0.409	0.315	0.321
perl	0.934	0.94	0.949	0.474	0.355	0.335

For the SPECfp95 benchmarks, a 0.5-kbyte L-Cache is almost as successful as the filter cache in reducing the energy of the I-Cache subsystem, especially when an aggressive scenario is followed. Filter caches with a 32 bytes clock size have large energy consumption per memory access, and they need a 32-byte large bus to connect them with the I-Cache. The filter caches capture all the instructions, no matter what their nesting is or how often they execute.

The performance overhead of these cache configurations with respect to the original execution time is given in Tables 6.9 and 6.10. This is a full chip simulation that takes into consideration the latency in the memory hierarchy, the structural hazards in the FPU, and the data dependency hazards in both the integer unit and the FPU.

Table 6.8 Normalized energy relative to the base machine for 512-byte extra cache.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.141	0.198	0.198	0.084	0.104	0.156
swim	0.139	0.145	0.173	0.092	0.114	0.174
su2cor	0.373	0.389	0.389	0.110	0.124	0.173
hydro2d	0.260	0.261	0.261	0.088	0.112	0.172
go	0.951	0.974	0.986	0.428	0.302	0.287
compress95	0.873	0.875	0.875	0.310	0.248	0.271
li	1	1	1	0.359	0.377	0.280
perl	0.934	0.94	0.949	0.421	0.32	0.308

Table 6.9 Normalized delay relative to the base machine for 256-byte extra cache.

Benchmark	*256 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	1	1	1	1.05	1.032	1.023
swim	1	1	1	1.032	1.018	1.013
su2cor	1	1	1	1.181	1.155	1.141
hydro2d	1	1	1	1.020	1.013	1.009
go	1.012	1.012	1.012	1.204	1.114	1.070
compress95	0.980	0.979	0.979	1.220	1.117	1.06
li	1	1	1	1.207	1.172	1.094
perl	1	1	1	1.244	1.153	1.102

The most important advantage of the L-Cache with respect to the filter cache is the small performance overhead, which is vital for high performance machines. The performance overhead is due to the miss rates in the L-Caches and the extra jump instructions that are inserted by the compiler as discussed previously. Filter cache configurations suffer from a much larger miss rate.

An optimal L-Cache has a size of 128 instructions (i.e., 0.5-kbytes) for the FP benchmarks. Small caches are not very succesful in disabling the I-Cache. Larger caches, on the other hand, have larger energy dissipation per access, yet not a much better hit rate than average sized caches. The energy dissipation drops as the size increases, but it goes up again for the larger caches. On the average, the new scheme dissipates only 55% of the energy of the original scheme for the FP benchmarks, when a 128-instruction L-Cache is included. Notice also that the new scheme never dissipates more energy than the original one.

Table 6.10 Normalized delay relative to the base machine for 512-byte extra cache.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	1	1	1	1.011	1.006	1.04
swim	1	1	1	1.011	1.006	1.004
su2cor	1	1	1	1.141	1.133	1.128
hydro2d	1	1	1	1.007	1.004	1.002
go	1.017	1.013	1.013	1.184	1.103	1.062
compress95	0.979	0.979	0.979	1.126	1.063	1.035
li	1	1	1	1.175	1.103	1.068
perl	1	1	1	1.211	1.131	1.084

6.3 Performance Improvements

The previous tables identify the opportunity for performance gains if the designer exploits the smaller access time of the extra caches. By reducing the clock period, delay along with energy can be simultaneously reduced. This concept is particularly attractive for our compiler-driven scheme, since it can benefit from the very high hit rate in the L-Cache.

We set the clock period equal to the access time of the I-Cache and we modify the size of the I-Cache so that it becomes the critical path in the CPU. We present results for a direct-mapped I-Cache of 8-kbyte, with a block size of 16 bytes. The new clock period is 7.91 ns. In this case, the access time for the D-Cache becomes two clock cycles. If the L-Cache can satisfy most of the requests from the pipeline, then the smaller I-Cache will not severely affect the hit rate. What is more, the smaller I-Cache consumes less energy. Therefore, the energy dissipation of the CPU is smaller, although the energy of the system will probably increase because the number of accesses outside the CPU will go up.

Again, we denote the execution time of the original configuration that uses no extra caches as unity, and we normalize everything else with respect to that. The extra cache size varies again between 256 and 512 bytes. We should also note that other such ideas can be readily applicable for enhancing performance by reducing the clock period. For example, the D-Cache, as opposed to the I-Cache, can be made smaller. In this case, we need two clock cycles to access the larger I-Cache.

By applying this framework to our simulator, we observed that energy as well as delay can be reduced. Tables 6.11 and 6.12 show the normalized energy dissipation of the new scheme

Table 6.11 Normalized energy relative to the base machine for a 256-byte extra cache, and a direct-mapped, 8-kbyte I-Cache with block size of 16 bytes.

Benchmark	*256 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.146	0.155	0.155	0.094	0.117	0.177
swim	0.095	0.096	0.100	0.082	0.109	0.17
su2cor	0.125	0.125	0.107	0.096	0.117	0.174
hydro2d	0.143	0.149	0.150	0.081	0.108	0.169
go	0.195	0.200	0.198	0.153	0.149	0.193
compress95	0.182	0.182	0.182	0.148	0.149	0.195
li	0.181	0.182	0.190	0.141	0.147	0.198
perl	0.191	0.192	0.193	0.155	0.155	0.210

Table 6.12 Normalized energy relative to the base machine for a 512-byte extra cache, and a direct-mapped, 8-kbyte I-Cache with block size of 16 bytes.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.078	0.086	0.086	0.077	0.106	0.169
swim	0.073	0.074	0.078	0.077	0.106	0.169
su2cor	0.108	0.110	0.110	0.081	0.109	0.171
hydro2d	0.145	0.152	0.152	0.075	0.105	0.168
go	0.195	0.198	0.200	0.147	0.145	0.192
compress95	0.183	0.183	0.183	0.121	0.133	0.189
li	0.181	0.183	0.190	0.132	0.139	0.191
perl	0.192	0.192	0.194	0.145	0.149	0.198

with respect to the original scheme. Since the I-Cache is only 8-kbyte in the new scheme, the on-chip energy consumption is much lower than in the original architecture, but the off-chip energy will probably go up, because more references have to access the L2 cache and the main memory. The original architecture is the one described in Table 6.3.

We notice from Tables 6.13 and 6.14 that performance can be significantly improved. This is especially true for the L-Cache, since the compiler can guide the hardware to access either the L-Cache or the I-Cache, and avoid the unacceptably high miss rates of the filter cache for some programs.

In SPECfp95 benchmarks, the delay when the L-Cache is included drops by 20% when the L-Cache is 256 bytes or 512 bytes. For a filter cache with a block size of 16 bytes, the delay

Table 6.13 Normalized delay relative to the base machine for a 256-byte extra cache, and a direct-mapped, 8-kbyte I-Cache with block size of 16 bytes.

Benchmark	*256 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.818	0.811	0.811	0.847	0.834	0.823
swim	0.823	0.823	0.823	0.845	0.835	0.83
su2cor	0.810	0.810	0.810	0.856	0.837	0.824
hydro2d	0.785	0.784	0.784	0.798	0.793	0.787
go	0.959	0.956	0.960	1.084	1.023	0.909
compress95	0.860	0.860	0.860	1.025	0.954	0.915
li	0.907	0.907	0.907	1.061	1.008	0.973
perl	0.989	0.989	0.989	1.141	1.079	0.999

Table 6.14 Normalized delay relative to the base machine for a 512-byte extra cache, and a direct-mapped, 8-kbyte I-Cache with block size of 16 bytes.

Benchmark	*512 bytes extra cache*					
	L-Cache			Filter Cache		
	(a)	(b)	(c)	8B	16B	32B
tomcatv	0.818	0.812	0.812	0.820	0.816	0.810
swim	0.823	0.823	0.823	0.833	0.829	0.824
su2cor	0.784	0.784	0.784	0.799	0.794	0.789
hydro2d	0.785	0.784	0.784	0.789	0.787	0.783
go	0.958	0.960	0.956	1.070	1.014	0.903
compress95	0.860	0.860	0.860	0.961	0.917	0.898
li	0.907	0.907	0.907	1.039	0.990	0.955
perl	0.989	0.989	0.989	1.119	1.064	0.984

drops by 17.5% when the filter cache is 256 bytes large, and by 20% when the filter cache is 512 bytes large.

The results favor the L-Cache approach when it comes to the integer benchmarks. The delay in the L-Cache approach drops by 9.5% in the SPECint95 benchmarks for either size. On the other hand, the delay increases by 1.6% for a 256 bytes filter cache with 16 bytes block size, and it remains the same for a 512 bytes filter cache with the same block size.

6.4 Summary

In this chapter we presented a detailed experimental evaluation for our approach, and we compared it to the filter cache, i.e., the method which uses an extra cache without compiler support. We showed that our method is preferable for high-performance processors because it only poses a small performance burden on the machine. Results from four FP and four integer benchmarks show that the energy dissipation in the on-chip memory hierarchy drops by 77% and 6%, respectively, for a 512-bytes L-Cache, and by 90% and 62%, respectively, for a 512-bytes filter cache with a block size of 8 bytes. However, the filter cache suffers from a 4.25% and 17.4% delay increase for the FP and integer benchmarks, respectively. On the other hand, the L-Cache has a negligible delay penalty. The filter cache has more significant energy gains since it stores all the basic blocks of the code, even if they are not accessed very often. Therefore, it might be preferable for the low-end, embedded market, where energy is more important than performance.

We also presented a modification in our scheme which targets performance as well as energy. The size of the I-Cache is reduced, and the clock period is set equal to the access time of the smaller I-Cache. This method was shown to have important performance improvements for programs for which the L-Cache has a high hit rate. Hence, it is particularly attractive for our scheme.

In Chapter 7, we will analyze a modified scheme which is adapted to integer benchmarks that do not perform well under the loop-based scheme.

Chapter 7

MODIFIED SCHEME FOR INTEGER BENCHMARKS

Integer benchmarks do not perform well under the loop-based selection algorithm as we have explained in the previous section. Most of the basic blocks in the SPECint95 benchmarks are not nested; hence, they cannot be placed in the L-Cache during execution. From a performance point of view, however, the L-Cache is still preferable to a filter cache in a processor that runs integer code.

The previous methodology was based on the detection of nested basic blocks in loops which did not contain function calls. These basic blocks were candidates for compiler-driven placement in the L-Cache. As is evident from the experimental results, the method is not succesful for a large category of integer benchmarks, such as interpreters and compilers. Figure 7.1 gives insight into the failure of the algorithm for some of the integer benchmarks.

Shown is the classification of the dynamic mix of instructions for the most troublesome SPECint95 benchmarks for a 0.5-kbyte L-Cache. An instruction belongs to one of the six following categories: "P" if it has been selected by the algorithm to be positioned in the L-Cache, "U" if it is in a basic block with a small execution frequency (unimportant), "NN" if it is in a block with large execution frequency but not nested in a loop, "SD" if it is in a nested block with large execution frequency but small execution density, "SS" if it belongs to a nested block with large frequency and execution density but small size, and "L" if it satisfies all the above criteria but does not fit in the L-Cache. For this experiment, the frequency threshold is $\frac{1}{10000}$ of the execution time of the program, the execution density threshold is five executions per function invocation, and the size threshold is five instructions.

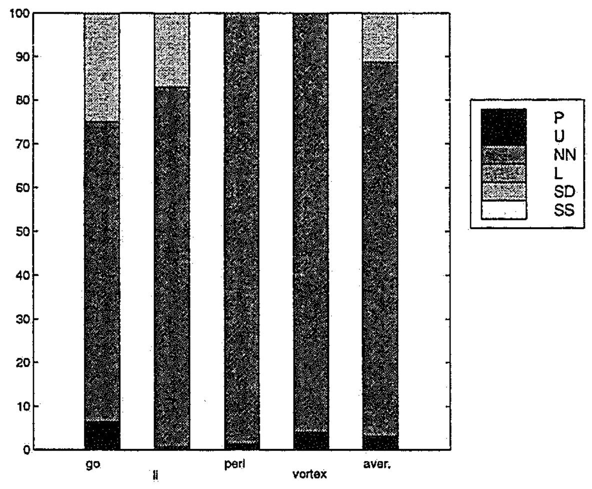


Figure 7.1 Instruction placement results for the SPECint95 benchmarks with a 128-instruction L-Cache.

The single most important reason that disqualifies the basic blocks of the integer benchmarks from being cached is nesting. Most of the basic blocks do not belong to a loop, or they belong to a loop that has a function call (85% of them). More than 10% of the basic blocks have small execution density.

The problem seems to be inherent to the structure of integer programs, especially when they are written in C/C++. This programming methodology favors small sections of sequential code, procedural abstraction (many functions), and lack of very deeply nested loops. The execution time is distributed among a larger number of basic blocks, many of which do not execute many times per function invocation. An alternative approach for selection of blocks for the L-Cache is therefore appropriate for these programs.

The proposed solution selects a function and places its most important basic blocks permanently in the L-Cache. In other words, they are not replaced when the thread of control leaves the function. Naturally, we select the function with the largest contribution in the execution time, as this has been designated by the profile data. The method consists of two steps as before.

7.1 Function Inlining

Before placement, our method performs function inlining to maximize the gains of this approach. For example, the function with the largest execution time might contain function calls to other functions. If these functions are inlined, the contribution of the original function in the total execution time will increase.

We use the annotated call graph of the program for inlining. However, we use different profile data to guide the process, since our target is different now. As before, the call graph is a weighted graph $G = (N, E, init, F_n, F_e)$. The definitions of N, E and $init$ are identical to the definition given in Chapter 3. The function $F_n : N \rightarrow [0, 1]$ is the contribution of a function to the total execution time. The function $F_e : E \rightarrow [0, 1]$ is the percentage of the function calls of a function n_j from n_i. The term $F_e(e_{ij})$ is the percentage of the function calls that were made from function n_i to function n_j with respect to all the function calls made to n_j.

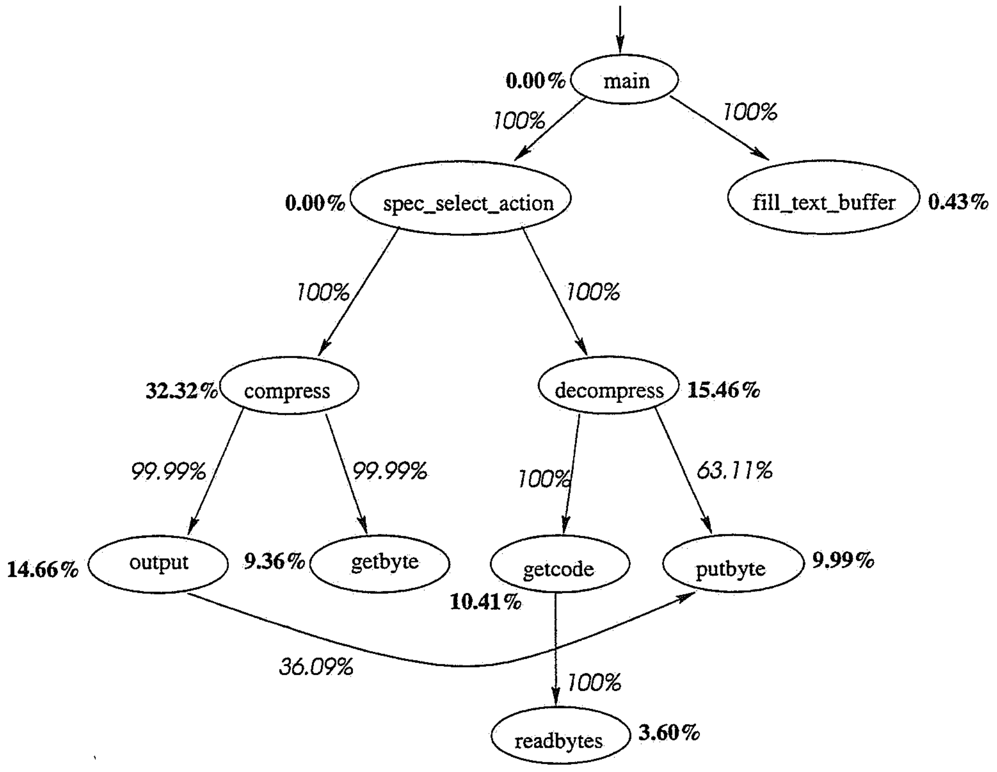


Figure 7.2 Call graph of the *129.compress* benchmark.

The inline heuristic scans each node n_i of the graph and computes the quantity

$$F_n(n_i) + \sum_{n_j} (F_e(e_{ij}) * F_n(n_j))$$

for every child n_j of n_i. It selects the function with the largest such quantity. This quantity reveals the potential of each function to contribute to the total execution time if it absorbs all of its callees. For example, the call graph of *129.compress* is given in Fig. 7.2. The function *putbyte* is responsible for 9.99% of the execution time of the program (only the function itself, not any of its descendants). In addition, 63.11% of all the function calls to *putbyte* are from *decompress*, whereas 36.09% are from *output*.

In practice, the inliner poses some restrictions on the process. For instance, if the resulting code is larger than a threshold, the inlining cannot proceed further.

We perform four inline experiments to test the applicability of function inline in the integer benchmarks. The first three apply brute force inlining to the callees of the most frequently executed functions (Table 7.1). The last experiment is based on the previously described heuristic.

Table 7.1 Inline experiments for the integer benchmarks.

Experiment A	No inlining
Experiment B	Inline the callees of the most frequently executed function
Experiment C	Inline the callees of the second most frequently executed function
Experiment D	Inline the callees of the third most frequently executed function
Experiment E	Use the inline heuristic

The execution frequency of the most heavily executed functions for the five experiments is shown in Table 7.2. Function inlining has a beneficial effect in exposing larger parts of code to frequently executed functions. In most cases, inline has a beneficial impact on execution time as well.

In theory, the numbers of Table 7.2 give the percentage of instructions which can be fetched from the L-Cache. In practice, this percentage is somewhat smaller since not all important basic blocks fit in the L-Cache, and the inliner of the MIPSpro compiler will not proceed to inline a function if the code size exceeds a limit.

7.1.1 Block placement

After inlining, the heuristic selects the most frequently executed function. If all the important basic blocks of the function fit in the L-Cache, the block placement algorithm will proceed to place them all. The size of the L-Cache is therefore important, unlike in the loop-based heuristic in which the integer benchmarks were almost insensitive to size variations.

In general, the problem can take the form of the *0-1 Knapsack* problem which is NP-complete [65]: Given a finite set U of basic blocks bb, each one with a size $s(bb)$, a value $n(bb)$ which is

Table 7.2 The contribution of the most frequently executed functions in the execution time.

Benchmark	*Experiment*				
	Exp. A	Exp. B	Exp. C	Exp. D	Exp. E
go	13.29%	13.47%	13.29%	13.17%	13.19%
m88ksim	30.39%	30.39%	30.39%	30.39%	25.49%
compress	32.32%	40.02%	35.20%	34.48%	40.02%
li	15.49%	15.49%	20.40%	15.61%	20.40%
perl	41.13%	41.67%	41.67%	42.30%	41.67%
vortex	17.05%	17.05%	17.05%	18.39%	18.39%

the number of executed instructions in bb, and a positive L-Cache size C, find a subset $U' \subseteq U$ of basic blocks such that $\sum_{bb \in U'} s(bb) \leq C$ and such that $\sum_{bb \in U'} n(bb)$ is as large as possible. Since a basic block can either be placed in the L-Cache or not (we cannot place part of the block), an optimal solution requires exponential time in the number of basic blocks.

We apply a greedy approximation algorithm which works as follows: we order the set U of basic blocks by the "key": $\frac{n(bb)}{s(bb)}$ so that $\frac{n(bb_1)}{s(bb_1)} \geq \frac{n(bb_2)}{s(bb_2)} \geq \cdots \geq \frac{n(bb_n)}{s(bb_n)}$. Starting with U' empty, we proceed sequentially through the list, each time adding a basic block bb whenever the sum of the sizes of the blocks already in U' and bb does not exceed C.

In addition, we perform another greedy procedure in which the list has been sorted using only the number of cycles $n(bb)$ of each basic block, so that $n(bb_1) \geq n(bb_2) \geq \cdots \geq n(bb_n)$. The best solution among the two is selected. A near optimal solution is obtained using this approach in our experiment.

7.2 Experimental Evaluation of the Modified Scheme

In the new experiments we did not set any size or density constraints. Since the basic blocks are placed in the L-Cache when the selected function is executed for first time and remain there afterwards, it does not make sense to pose extra limitations in their selection. The results have been taken after the transformations of Experiment B of Table 7.1 have been applied to the initial code.

The memory hierarchy subsystem is described in Table 6.3. The energy gains of the L-Cache are given in Tables 7.3 and 7.4. The results are very encouraging for benchmarks that have poor performance under the initial method. On average, the energy dissipated in the I-Cache/L-Cache subsystem is 84.5% of the energy in the original I-Cache subsystem with almost no performance overhead. Similar results are obtained for most of the integer benchmarks that do not score well under the old scheme (e.g. *130.li, 134.perl*).

Table 7.3 Normalized energy and delay relative to the base machine for a 256-byte extra cache, using the modified scheme for integer benchmarks.

Benchmark	*Energy*	*Delay*
compress95	0.808	0.979
li	0.286	0.984
perl	0.823	1

Table 7.4 Normalized energy and delay relative to the base machine for a 512-byte extra cache, using the modified scheme for integer benchmarks.

Benchmark	*Energy*	*Delay*
compress95	0.776	0.979
li	0.269	0.981
perl	0.823	1

The execution time overhead is negligible in this scheme for an L-Cache of 0.5-kbyte. This is because the hit rate is almost 100% and the L-Cache is large enough to accommodate all the important basic blocks of a function. The improved delay is mainly due to the function inlining we performed before block placement.

7.3 Summary

In this chapter we focused on the integer benchmarks, and we gave an alternative approach for selecting basic blocks for the L-Cache in these programs. This method is based on profile data, and it selects the most frequently executed basic blocks of a function even if they do not belong to a loop. The selected basic blocks are placed once in the L-Cache, and they are not replaced thereafter. Function inlining is used to expose as many basic blocks as possible in the function that provides the basic blocks.

Experimental results show that this method offers negligible performance overhead on the SPECint95 benchmarks, and significant energy reduction in the memory hierarchy subsystem.

In the next chapter we will detail a different method to select basic blocks for the extra cache, which is based on run-time statistics, as opposed to profiling.

Chapter 8

DYNAMIC TECHNIQUES FOR BASIC BLOCK SELECTION

In the previous chapters we proposed and analyzed methods for selecting portions of the executing code for placement in the L-Cache. Those methods were static, i.e., they required the involvement of the compiler and previous runs of the code to select the basic blocks for the L-Cache. The compiler restructures the code and remaps some of the basic blocks in the global memory address space to facilitate the placement of those blocks in the L-Cache. The resulting extra hardware was shown to be simple and straightforward since most of the work was carried out statically by the compiler.

However, profiling might not always be an effective or even possible solution. This is because the user might not be willing to perform this preliminary step, or because the execution time of the program is prohibitively long to be carried out twice. Moreover, the previous methodology assumes the involvement of the compiler, and the cooperation of the compiler with the hardware, which might not always be possible. If a microprocessor company or a design house has no jurisdiction over the compiler technology, the hardware/software codesign idea might not be applicable.

In this chapter we propose various alternative schemes for the selection of basic blocks for placing them in the L-Cache. The schemes are based on the *dynamic* selection of basic blocks during run-time. Our approach seeks to manage the L-Cache in a manner that is sensitive to the frequency of accesses of the instructions executed. It can better exploit the temporalities of the code and can make decisions on-the-fly, i.e., while the code executes. It adapts itself to the variability in the behavior of the program during execution without using any compiler support.

The problem that the dynamic techniques seek to solve is how to select basic blocks to be stored in the L-Cache while the program is being executed. If a block is selected, the CPU will access the L-Cache first; otherwise, it will go directly to the I-Cache. In case of an L-Cache miss, the CPU is directed to the I-Cache to get the instruction and, at the same time, to transfer the instruction from the I-Cache to the L-Cache. The L-Cache is loaded with instructions from the I-Cache after a miss.

8.1 Related Work

There has been an extensive research effort lately on techniques to improve the memory hierarchy performance through dynamic techniques. This effort has almost always targeted delay rather than energy reduction. Memory latency remains the single most important problem in the design of high-performance processors, and researchers in industry and academia have devoted considerable effort to reduce it.

A great deal of work is focusing on developing ways to improve the management of the caches, and to store there only the most frequently executed references. In [66] the authors propose methods with which memory references can bypass the Level-1 (L1) caches and be fetched from the Level-2 (L2) caches straight to the CPU. In other words, if the L1 cache misses, and the L2 cache has to provide tha data, those references will not be transfered in the L1 cache. This aims at reducing the *cache pollution* in the L1 cache, i.e., at avoiding placing in the L1 caches data with small spatial or temporal locality that will replace useful data.

The authors in [66] examine the behavior of the *load* and *store* instructions. If a *load* or *store* instruction causes a large number of misses, its memory references bypass the L1 data cache altogether. This method needs profiling to characterize which *load* or *store* instructions cause a large number of misses. In the same paper, a dynamic method is proposed that can be used instead of profiling.

Along the same line, the authors in [67] and [68] present techniques for dynamic analysis of program data access behavior, which are then used to proactively guide the placement of data within the memory hierarchy. Data that are expected to have little reuse in the cache are bypassed and are not placed in the L1 D-Cache. To implement this scheme, the authors use extra hardware, the *Memory Address Table* (MAT), which is a table of counters. They partition the main memory into blocks so that the number of blocks is equal to the number of counters in the MAT. Each time a location within a block is accessed in the main memory, the counter that corresponds to that block is incremented by one. That way, the MAT keeps statistics about the frequency of access of the blocks in the main memory. If a reference is from a block that has not been accessed very often in the past, the CPU bypasses this reference and does not store it in the L1 D-Cache.

In a similar work [68], the authors propose a scheme that dynamically adjusts the amount of data fetched on a cache miss from the L2 to the L1 D-Cache. They introduce the *spatial locality detection table* (SLDT), which is used together with the MAT to detect spatial locality of data in the cache, and to fetch a larger number of words from the L2 cache to the L1 cache when the spatial locality is large.

In a more recent work, the authors propose a cache-conscious data placement in the global address space to reduce the misses in the D-Cache [69]. In [70], the authors target the problem of *variable spills*, i.e., the explicit transfer of variables from registers to memory and back again by the program. This is done when there are not enough registers to accomodate all the variables in the program in a given time. They propose the addition of a *compiler-controlled memory* which is under the exclusive control of the compiler and is used to serve the memory traffic due to the spill of variables.

All these techniques attempt to guide the memory hierarchy as to which references should be cached and which should be bypassed. Clearly, frequently accessed references with large spatial and temporal locality should be allowed to stay longer in the L1 caches without being replaced by references with low reuse. This selection is done dynamically by investing extra hardware to keep statistics for the access pattern.

These techniques can also be used in our scheme to manage the caching of instructions in the L-Cache. They can detect the most frequently executed portions of the code dynamically, and direct the L-Cache to store only those portions. However, they require the addition of extra hardware in the form of extra tables or counters to keep statistics during execution. The extra hardware dissipates energy, and can offset the possible energy gains from the usage of the L-Cache. To make the dynamic techniques attractive for low energy, we need to use hardware that already exists in the CPU. The hardware we will use in this work is the *branch prediction* mechanism.

In Section 8.2, we give a thorough review of branch prediction and the mechanisms and technniques that have been proposed to improve it. In Section 8.3 we refer to the relatively new concept of *confidence estimation* and explain how it can be used to quantify the results of branch prediction. In Section 8.4, we detail our solution to dynamic selection of basic blocks to be cached in the L-Cache, and give several techniques that trade off delay and energy reduction. These techniques use the foundations developed in Sections 8.2 and 8.3. The experimental results for each technique are given in Section 8.4.

8.2 Branch Prediction

As the processor speed increases and instruction-level parallelism becomes widely used, conditional branches pose an increasingly heavy burden for the growth of uniprocessor performance. To fully exploit the potential of the very powerful CPU cores, programs need to have as few branches as possible. Various compiler techniques, such as loop unrolling, can help towards that direction, yet they cannot fully solve the problem in integer programs, which have few loops and small basic blocks.

Branch prediction is the most popular method to increase parallelism in the CPU, by predicting the outcome of a conditional branch instruction as soon as it is decoded. Provided that the branch prediction rate is high, the pipeline executes from the correct path and avoids unnecessary work most of the time. In such a case, the pipeline only executes from a straight line code and can issue more than one instruction per clock cycle.

The branch prediction problem can actually be divided into two subproblems. The prediction of the direction of the branch and the prediction of the target address if the branch is predicted to be taken. Both subproblems should be solved for the branch prediction to be meaningful. The most common method to solve the target prediction subproblem is to use a *Branch Target Buffer*, which is a special cache used to store the target addresses of branches. We are only interested in the prediction of the branch direction in this work.

8.2.1 Previous work on branch prediction

There has been an extensive amount of research on hardware-based branch predictors in the last few years [71]. Successful branch prediction mechanisms take advantage of the non-random nature of branch behavior. Most branches are either taken or not-taken. Moreover, the behavior of a branch usually depends on the behavior of the surrounding branches in the program.

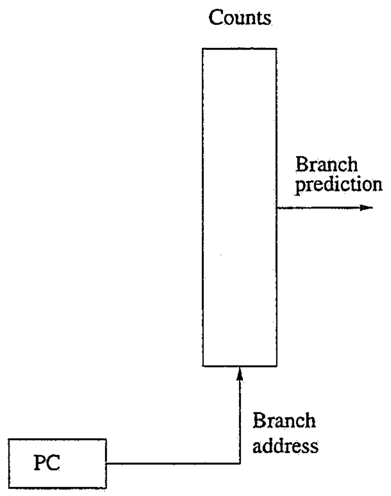


Figure 8.1 Bimodal branch predictor. Each entry in the table is a 2-bit saturated counter.

Bimodal branch predictor. The *bimodal* branch predictor in Fig. 8.1 takes advantage of the bimodal behavior of most branches. Each entry in the table shown in Fig. 8.1 is a 2-bit saturated counter which determines the prediction. Each time a branch is taken, the counter is incremented by one, and each time it falls through it is decremented by one (Fig. 8.2). The prediction is done by looking into the value of the counter: if it less than 2, the branch is predicted as not taken; otherwise, it is predicted as taken. By using a 2-bit counter, the predictor can tolerate a branch going into an unusual direction once. More generally, an n-bit saturated counter could be used, but experiments have shown that a 2-bit counter is almost as good.

The table is accessed through the address of the branch using the PC. Ideally, each branch has its own entry in the table, but for smaller tables multiple branches may share the same entry. The table is accessed twice for each branch: first to read the prediction, and then to modify it when the actual branch direction has been resolved.

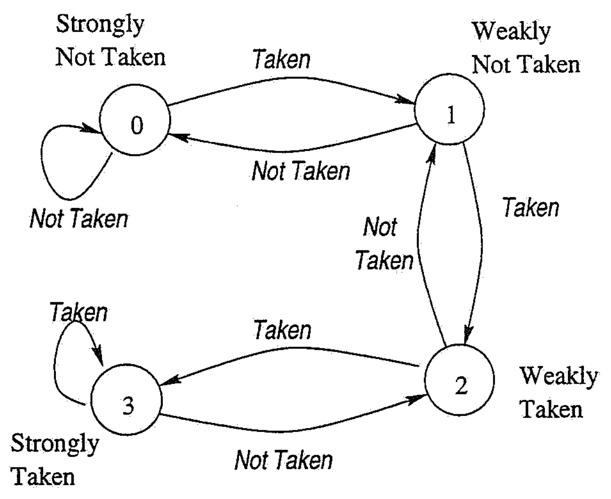


Figure 8.2 FSM for the 2-bit saturated counters.

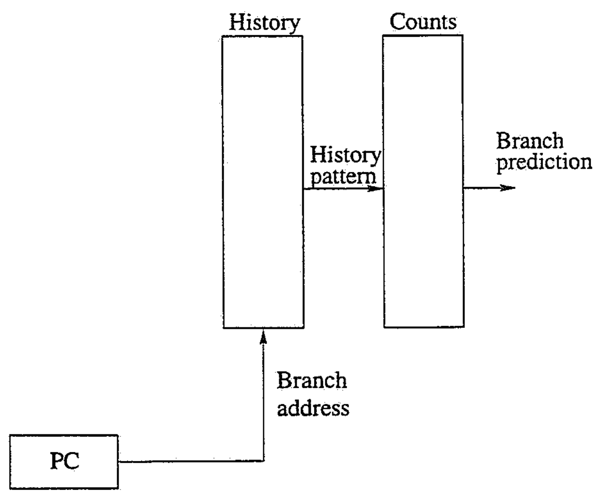


Figure 8.3 Local branch predictor.

Local branch predictor. The bimodal predictor can be improved for branches that execute repetitive patterns [72]. If 1 represents taken and 0 not taken, a 4-bit-long pattern of the form $(1110)^n$ means that the branch is taken three times and not taken once. The method described in [72] is called *local method*, and is based on using two tables to keep the necessary information for a correct prediction (Fig. 8.3). The first table records the patterns for each branch in the program. Again, aliases may occur for small history tables. The second table is the array of the 2-bit counters as in the bimodal method, and it is accessed by the branch history patterns stored in the first table. If each entry in the first table is n-bits long, the size of the second table will be 2^n entries.

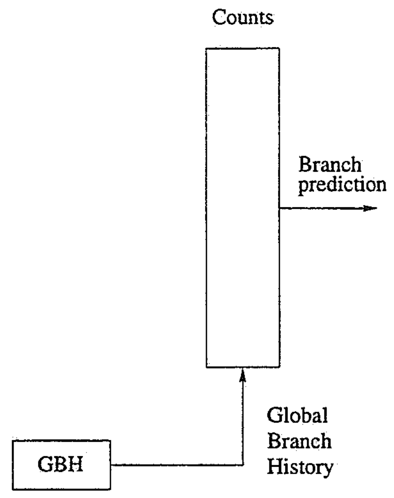


Figure 8.4 Global branch predictor.

This way, the predictor has more information about the recent behavior of a branch and can distinguish between more refined cases than in the bimodal method. For large predictors, the accuracy of the prediction is about 97%. This method, however, requires a larger investment in hardware, and might adversely affect the clock cycle.

Global branch predictor. In the previous methods, only the past behavior of the current branch was considered. Another scheme was proposed in [73] which also considers the behavior of other branches to predict the behavior of the current branch. This is called global prediction, and the hardware implementation is similar to the implementation of the bimodal method (Fig. 8.4). The difference is that the table with the counters is accessed with the *Global Branch History* (GBH) register, which contains the outcome of the n more recent branches. A single shift register, which records the direction taken by the n most recent branches, can be used.

The global method can be improved if the predictor combines the GBH register with the branch address [74] [75]. This means that each branch will have its own set of predictions, and the scheme will be able to distinguish between the behavior of different branches. An implementation of this method is shown in Fig. 8.5, where the GBH and the PC are combined by XORing them. This method has a branch prediction rate close to the 97% for large prediction tables.

McFarling branch predictor. Finally, McFarling [75] combines two predictors to achieve better results. In Fig. 8.6, a McFarling predictor is shown which consists of three

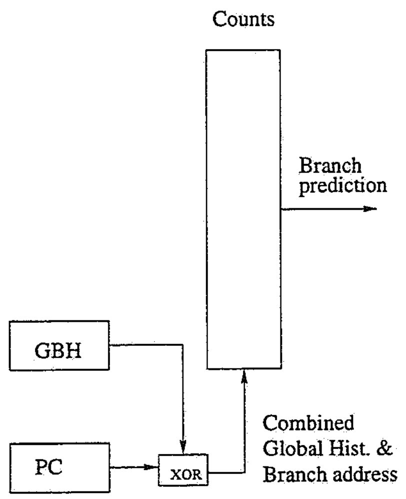


Figure 8.5 Global branch predictor with index sharing.

tables. The tables PR1 and PR2 contain the counters for the two independent predictors, and the selector counter determines which predictor will be used to give the prediction. The two predictors can be any of the predictors we discussed in the previous paragraphs. McFarling found out that the combination of a local and a global predictor with index sharing gives the best results.

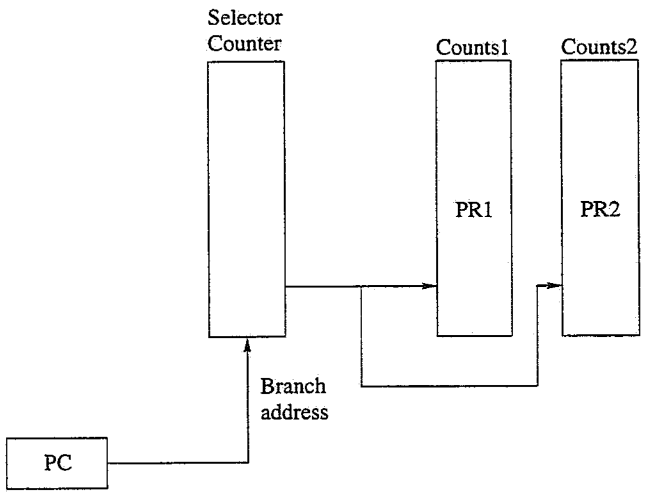


Figure 8.6 McFarling branch predictor.

Each entry in the selector counter contains a 2-bit saturated counter. This counter determines which predictor will be used for the prediction and is updated after the direction of the branch has been resolved. The counter is biased towards the predictor that gave the correct prediction. The accuracy of the McFarling predictor approaches 98% for large arrays.

8.3 Confidence Estimation

In many cases computer architects want to assess the quality of a branch prediction and determine how confident the machine is that the prediction will be correct. The relatively new concept of *confidence estimation* has been introduced recently to quantify this confidence and keep track of the quality of branch predictors [76].

The confidence estimators are hardware mechanisms that are accessed in parallel with the branch predictors when a branch is decoded, and they are modified when the branch direction is resolved. They characterize a branch prediction as "high confidence" or "low confidence" depending upon the history of the branch predictor for the particular branch. For example, if the branch predictor predicted a branch correctly most of the time, the confidence estimator would designate this prediction as "high confidence," otherwise as "low confidence." We should note that the confidence estimation mechanism is orthogonal to the branch predictor used. In other words, we can use any combination of confidence estimators and branch predictors.

There are a number of applications in which the confidence estimation mechanisms can be used [77]. In a multithreading CPU, the machine would be more willing to switch threads if there is uncertainty for the outcome of a branch. If there is a large probability that the branch is mispredicted, the machine can switch to another thread to hide the misprediction latency. A "low confidence" branch can trigger this thread switch.

In [78], the authors show how the confidence estimators can be used to reduce the energy dissipation in the pipeline of speculative, high-performance processors. In such a processor the machine predicts the outcome of a branch and executes from the predicted path speculatively. It only commits the results if the path is the correct one, i.e., after the branch has been resolved. The machine executes 20-100% more instructions than it commits due to speculative execution. Extrapolating current trends, speculation seems to become more important in the future [79].

In modern processors, the machine might need to predict many branches before any of them is resolved, and all of them need to be predicted correctly for the speculative execution to be beneficial. A series of "low confidence" branches in the pipeline indicate that there is a large probability that the machine fetches and executes instructions from the wrong path. For example, if a "low confidence" branch has a misprediction rate of $p = 30\%$, the probability that the machine executes from a wrong path after three such branches have been encountered is $1 - (1-p)^3 = 0.657$.

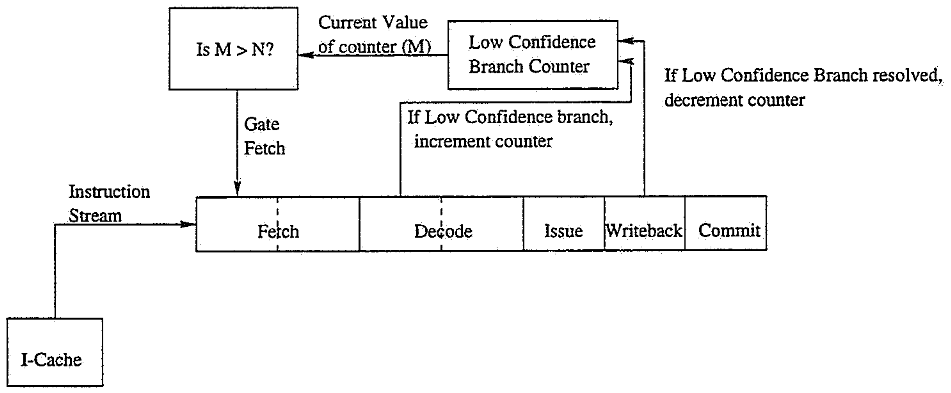

Is M > N?
Current Value of counter (M)
Low Confidence Branch Counter
If Low Confidence Branch resolved, decrement counter
If Low Confidence branch, increment counter
Gate Fetch
Instruction Stream
Fetch
Decode
Issue
Writeback
Commit
I-Cache

Figure 8.7 Pipeline gating schematic. The fetch and decode stages take two cycles.

The authors in [78] propose a *pipeline gating* scheme in which the machine disables instruction fetching from the I-Cache, when there are more than M "low confidence" branches in the pipeline (Fig. 8.7). They use the pipeline model of a superscalar, speculative machine, in which more than one instruction can be issued and executed in a clock cycle. The machine uses a branch predictor which attempts to predict the direction of a branch as soon as the branch is decoded in the front-end of the pipeline. The fetch unit will then start accessing instructions from the predicted target address. Since this execution is speculative, the pipeline needs a final stage which is used to commit the results of the speculative execution only if the branch prediction was correct. Otherwise, the pipeline is flushed, and the execution starts again from the mispredicted branch.

In their approach, when the confidence estimator detects such a branch, it increments the low confidence counter. This counter is decremented when a "low confidence" branch is resolved in the writeback stage of the pipeline. Using this modification, the machine can stop execution in the pipeline when there is a large probability of wrong path execution, until the troublesome branches are resolved. The scheme was shown to have a minimal effect in execution time and a large reduction in the number of instructions executed and, hence, the energy dissipated.

In [77], the authors introduce some useful metrics to compare various confidence estimators. Those metrics combine the confidence of a branch ("low" or "high") with the correctness of a prediction ("correct" or "incorrect"). For example, the *sensitivity* is defined as the fraction of correct predictions identified as "high confidence." The *predictive value of positive test* (PVP) is defined as the fraction of "high confidence" predictions that are correct. Each of these metrics is simple to compute, and each is a "higher is better."

8.3.1 Past work in confidence estimation

Various confidence estimation techniques have been proposed in [76] and [77]. Unlike the branch predictors, whose performance can be easily measured using the prediction rate, the confidence estimators are not easy to characterize. Different confidence estimators can be useful for different applications.

JSR confidence estimator. A one-level and a two-level resetting counter estimator was proposed in [76]. The one-level estimator has the same organization as the global branch predictor with index sharing, and the two-level estimator has the same organization with the local branch predictor. The one-level estimator has a table in which each entry is an n-bit saturating counter (Fig. 8.8).

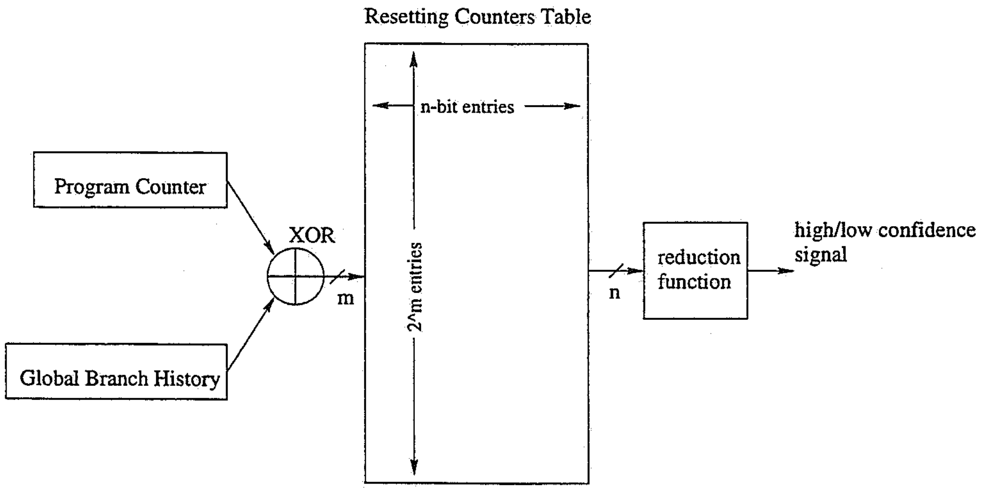


Figure 8.8 One-level JSR confidence estimator.

The table is accessed in the same manner as the global branch predictor with index sharing. It has an n-bit saturating and resetting counter which is used to record the history of predictions for each branch as follows: Each time the branch predictor makes a correct prediction, the counter is incremented by one, whereas each time it mispredicts it is reset to zero. When a branch is predicted, the estimator is accessed, and the value of the counter is compared to a specific threshold. If the value is above that threshold, the branch is considered to have "high confidence"; if it is below it is considered to have 'low confidence." Therefore, branches that are executed after a mispredicted branch are tagged as "low confidence," since they are probable to be mispredicted.

Saturating counters confidence estimator. Another method proposed was the *saturating counters estimator.* In this method the saturating counters of the branch predictors are used to determine the confidence of a prediction. This method does not need any extra hardware for the implementation of the estimator, but it is less accurate than the JSR method.

8.4 Using Branch Prediction and Confidence Estimation in the L-Cache Scheme

The dynamic scheme for the L-Cache should be able to select the most frequently executed basic blocks for placement in the L-Cache without any compiler support. It should also rely on existing mechanisms without much extra hardware investment if it is to be attractive for energy reduction.

The branch prediction in conjunction with the confidence estimator mechanism is a reliable solution to this problem. During program execution, the branch predictor accumulates the history of branches and uses this history to guess the branch behavior in the future. Since the branch predictor is usually successful in predicting the branch direction, we can assume that predictors describe accurately the behavior of the branch during a specific phase of the program. Confidence estimators provide additional information about the steady-state behavior of the branch.

For example, a "high confidence" branch that was predicted "taken" will be expected to be taken during program execution in that particular phase of the program. If it is not taken (i.e., in case of a misprediction), it will be assumed to behave unusually. Of course, what is "usual" or "unusual" behavior in the course of a program for a particular branch can change. Some branches can change behavior from mostly taken to mostly untaken during execution. Moreover, many branches, especially in integer benchmarks, can be in a gray area, and not have a stable behavior with respect to direction, or can follow a complex pattern of behavior.

If a branch behaves "unusually," it will probably drive the thread of control to a portion of the code that is not very frequently executed. The loop shown in Fig. 8.9 executes the basic

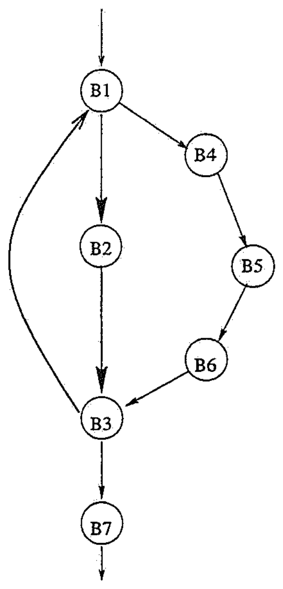


Figure 8.9 An "unusual" branch direction leads to a rarely executed portion of the code.

blocks B_1, B_2, and B_3 most of the time, and it seldom executes B_4, B_5, and B_6. The branch at the end of B_1 will be predicted "not-taken" with "high confidence." If it is taken, it will drive the program to the rarely executed branch, i.e., it will behave unusually. A similar situation exists for B_3 and B_7.

These observations lay the foundation for the dynamic selection of basic blocks in the L-Cache scheme. In our approach, we attempt to capture the most frequently executed basic blocks by looking into the behavior of the branches. The basic idea is that, if a branch behaves "unusually," our scheme disables the L-Cache access for the subsequent basic blocks.

Not all branches can be characterized as "high confidence." In Fig. 8.10, the dynamic branches are classified according to how many times they are "taken." For example, the height of the column 30-39% gives the percentage of the dynamic branches that are "taken" from 30 to 39% percent of the time. For the 130.*li* benchmark, almost 11% of all the dynamic branches were "taken" between 30% and 39% of the times.

This experiment shows that different benchmarks demonstrate different branch behavior. For most integer, and many FP benchmarks there are many branches that cannot be classified as "mostly taken" or as "mostly not taken." These benchmarks can be "mostly taken" or "mostly not taken" in different stages of the execution, or they may follow a more random pattern.

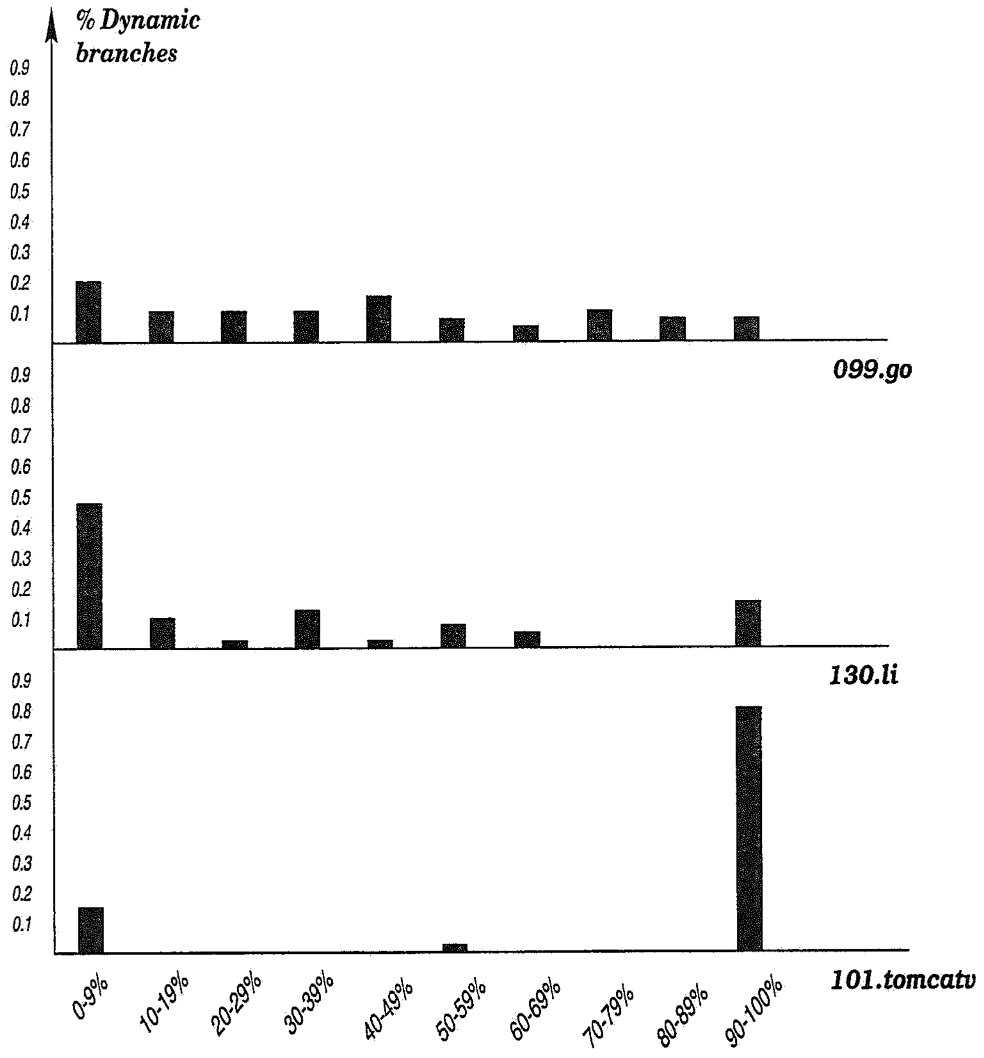


Figure 8.10 Branch direction percentages for four SPEC95 benchmarks.

Branch predictors have a much easier task in programs like 101.*tomcatv* in which the behavior of branches is predictable, than in 099.*go* where it is more erratic. For a lot of integer programs, branches do not demonstrate a bimodal behavior. In Fig. 8.11, the branch misprediction rates for most of the SPEC95 benchmarks are shown. We use a McFarling branch predictor in which each one of the three tables used has 2048 entries. The bimodal and the global branch predictor with index sharing are used.

Programs whose branches are evenly distributed in all the columns of Fig. 8.10 suffer from a large misprediction rate even with the McFarling predictor. On the other hand, programs whose dynamic branches are concentrated near the edges do not have such problems. In general, branch predictors are quite successful with numerical or computation-intensive code.

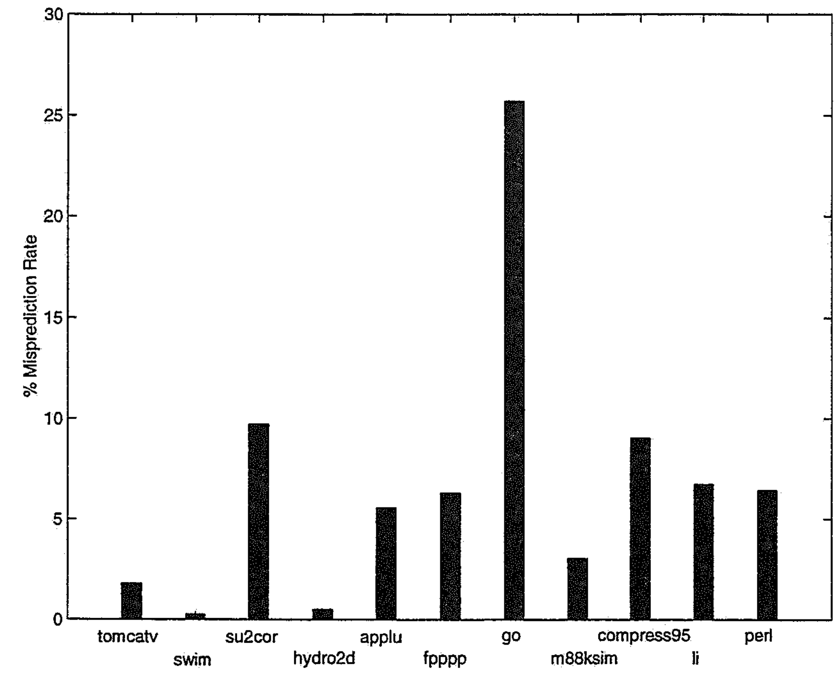


Figure 8.11 Misprediction rate for the SPEC95 benchmarks.

In the following subsections we propose various dynamic methods for the selection of basic blocks that span the range of accuracy and complexity. We make the realistic assumption that the processor is already equiped with a branch prediction mechanism. We are assuming a McFarling predictor for all experiments.

Tables 8.1 and 8.2 show the normalized energy dissipation and delay of the filter cache configuration with respect to the original scheme. We only present results for a block size of 8 and 16 bytes, since a larger block size requires a larger bus to link the L-Cache with the I-Cache. The memory hierarchy subsystem is the same one shown in Table 6.3.

8.4.0.1 Simple method without using confidence estimators

The branch predictor can be used as a stand-alone mechanism to provide intuition about which portions of the code are frequently executed and which are not. A mispredicted branch is assumed to drive the thread of execution to an infrequently executed part of the program, assuming that the branch predictor is correct most of the time.

Table 8.1 Normalized energy for the filter cache.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.182	0.174	0.097	0.117
swim	0.123	0.138	0.099	0.118
su2cor	0.198	0.176	0.127	0.135
hydro2d	0.122	0.134	0.088	0.112
applu	0.298	0.220	0.257	0.196
fpppp	0.566	0.352	0.557	0.348
go	0.478	0.332	0.426	0.302
m88ksim	0.424	0.300	0.370	0.273
compress95	0.385	0.288	0.268	0.219
li	0.409	0.315	0.359	0.272
perl	0.481	0.355	0.415	0.310

Table 8.2 Normalized delay for the filter cache.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.049	1.031	1.010	1.006
swim	1.028	1.017	1.015	1.008
su2cor	1.062	1.036	1.026	1.016
hydro2d	1.021	1.013	1.007	1.004
applu	1.111	1.057	1.090	1.045
fpppp	1.240	1.121	1.235	1.118
go	1.224	1.126	1.194	1.109
m88ksim	1.190	1.106	1.160	1.091
compress95	1.210	1.124	1.131	1.077
li	1.202	1.126	1.171	1.100
perl	1.242	1.149	1.202	1.117

Our strategy is as follows: If a branch is mispredicted, the machine will access the I-Cache to fetch instructions. If a branch is predicted correctly, the machine will access the L-Cache. In a misprediction, the pipeline will flush, and the machine will start fetching instruction from the correct address by accessing the I-Cache. In a correct prediction, the machine will start fetching instructions from the L-Cache as soon as the branch is resolved. This might well be several instructions after the branch in a high-performance, superpipelined processor has executed.

Figure 8.12 shows the microarchitecture for the simple method. We assume a more advanced pipeline as in Fig. 8.7 for illustration purposes. The results, however, are reported for the R-4400 pipeline that implements the MIPS2 ISA.

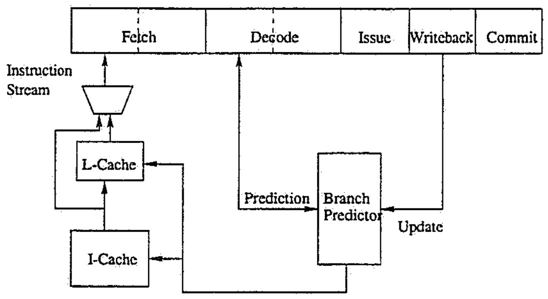


Figure 8.12 Microarchitectural modifications for the simple method.

We implemented this method using MINT [64]. The memory hierarhcy is similar to the one in the compiler-driven experiments: both the I-Cache and the D-Cache are direct-mapped, 32-kbyte large, with a block size of 32 bytes. The L-Cache is 256 or 512 bytes, direct-mapped with a block size that varies between 8 and 16 bytes. We compare the energy and delay characteristics of the dynamic scheme with the filter cache, as we did for the compiler method. The filter cache is direct-mapped, its size varies between 256 and 512 bytes, and its block size varies between 8 and 16 bytes.

As before, we denote the energy dissipation and the execution time of the original configuration that uses no extra caches as unity, and normalize everything else with respect to that. Our model accounts for all possible stalls in the R-4400 CPU which is used as the base machine. In addition, we account for a branch misprediction stall, which is equal to two clock cycles.

Table 8.3 Energy results for the simple method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.185	0.177	0.100	0.121
swim	0.123	0.134	0.099	0.118
su2cor	0.238	0.208	0.161	0.172
hydro2d	0.125	0.137	0.091	0.115
applu	0.329	0.253	0.292	0.232
fpppp	0.574	0.365	0.566	0.361
go	0.609	0.509	0.572	0.488
m88ksim	0.435	0.315	0.382	0.288
compress95	0.437	0.349	0.338	0.290
li	0.453	0.363	0.403	0.322
perl	0.513	0.396	0.451	0.355

Table 8.4 Delay results for the simple method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.050	1.032	1.011	1.006
swim	1.028	1.017	1.015	1.008
su2cor	1.056	1.030	1.021	1.012
hydro2d	1.020	1.013	1.006	1.004
applu	1.108	1.056	1.089	1.045
fpppp	1.235	1.118	1.230	1.116
go	1.159	1.091	1.138	1.079
m88ksim	1.184	1.103	1.155	1.085
compress95	1.193	1.115	1.126	1.074
li	1.189	1.118	1.159	1.093
perl	1.225	1.138	1.188	1.114

Tables 8.3 and 8.4 show the normalized energy and delay results for the SPEC95 benchmarks. Our method performs better with respect to performance compared to the filter cache, but it is not as good for energy gains. The two methods have similar delay and energy characteristics for FP benchmarks, but they differ for integer benchmarks. This is because the branch predictor has a smaller prediction rate for integer benchmarks; thus, it selects fewer basic blocks for placement in the L-Cache. This is shown in Table 8.5, which shows the percentage of dynamic instructions that cause the L-Cache to be accessed. For FP benchmarks, the percentage is near 100%.

Table 8.5 Dynamic instructions that cause the L-Cache to be accessed in the simple method.

Benchmark	% dyn. instructions
tomcatv	99.47%
swim	99.95%
su2cor	95.01%
hydro2d	99.58%
applu	95.94%
fpppp	98.00%
go	73.09%
m88ksim	97.56%
compress95	91.69%
li	93.11%
perl	93.57%

8.4.0.2 Static method

The next technique we used to select basic blocks for the L-Cache is not dynamic. We used profiling again, and simulated the branch predictor. We captured the behavior of the branches of the program and we classified them as "high confidence" if they were predicted correctly most of the time, and "low confidence" if not. A threshold was used to determine confidence, so that the branches that were predicted correctly at least 90% of the time were tagged as "high confidence," whereas all other branches were tagged as "low confidence."

After profiling, we ran the benchmarks again and we selected the basic blocks as follows: If a "high confidence" branch was predicted incorrectly, the I-Cache is accessed for the subsequent basic blocks. Moreover, if more than two "low confidence" branches have been decoded, the I-Cache is accessed. In any other case, the machine accesses the L-Cache.

The first rule for accessing the I-Cache is due to the fact that a mispredicted "high confidence" branch behaves "unusually" and drives the program to an infrequently executed portion of the code. The second rule is due to the fact that a series of "low confidence" branches will also suffer from the same problem since the probability that they are all predicted correctly is low.

There are two controlling parameters in the static method; the threshold used to classify a branch as "high" or "low confidence" and the number of successive "low confidence" branches which need to be executed before the machine turns to the I-Cache. A larger threshold of successive "low confidence" branches results in more basic blocks accessed from the L-Cache.

Tables 8.6 and 8.7 show the normalized energy and delay results for the SPEC95 benchmarks. The same experiments and the same experimental framework was used here. The results we present are from self-profiled executions where the same input was used to profile and evaluate our approach. We include this approach to indicate its potential.

The problem with the static method is that it does not exploit the temporalities of the branches, but it only assigns a confidence to them statically. It has similar performance to the simple method, except for the *099.go* benchmark for which it has much lower delay and much higher energy dissipation. This is because this particular benchmark has a large number of "low confidence" branches. Table 8.8 presents the percentage of dynamic instructions which cause the CPU to access the L-Cache.

Table 8.6 Energy results for the static method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.185	0.163	0.100	0.120
swim	0.123	0.134	0.099	0.118
su2cor	0.201	0.194	0.130	0.154
hydro2d	0.124	0.137	0.091	0.114
applu	0.309	0.231	0.269	0.208
fpppp	0.572	0.361	0.562	0.355
go	0.757	0.702	0.736	0.690
m88ksim	0.431	0.309	0.378	0.283
compress95	0.410	0.315	0.294	0.246
li	0.431	0.337	0.382	0.296
perl	0.520	0.393	0.531	0.450

Table 8.7 Delay results for the static method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.049	1.025	1.011	1.006
swim	1.029	1.017	1.015	1.008
su2cor	1.061	1.033	1.027	1.013
hydro2d	1.021	1.013	1.007	1.004
applu	1.109	1.057	1.089	1.045
fpppp	1.239	1.121	1.234	1.118
go	1.090	1.052	1.078	1.045
m88ksim	1.187	1.104	1.158	1.089
compress95	1.205	1.121	1.126	1.074
li	1.198	1.124	1.169	1.099
perl	1.226	1.138	1.158	1.095

8.4.0.3 Using a confidence estimator

As we showed in Fig. 8.11, branch predictors are not always able to give a correct prediction. Therefore, we need a confidence estimation mechanism which, coupled with the branch predictor, gives a better intuition about the behavior of the branch.

We are using a similar methodology as in the static method, but no profiling information is used. Instead, the confidence of each branch is determined dynamically using the *saturating counters* approach. In that approach, we use the prediction of each one of the component predictors (the bimodal and the global) to determine the confidence. If both predictors are strongly biased in the same direction (both "strongly taken" or both "strongly not-taken"),

Table 8.8 Dynamic instructions that cause the L-Cache to be accessed in the static method.

Benchmark	% dyn. instructions
tomcatv	99.55%
swim	99.95%
su2cor	97.45%
hydro2d	99.69%
applu	98.57%
fpppp	99.02%
go	43.52%
m88ksim	98.59%
compress95	96.42%
li	97.08%
perl	79.49%

we signal a "high confidence" branch. In any other case, we signal a "low confidence" branch. This methodology uses a minimal amount of extra hardware and has been shown to be reliable in [77].

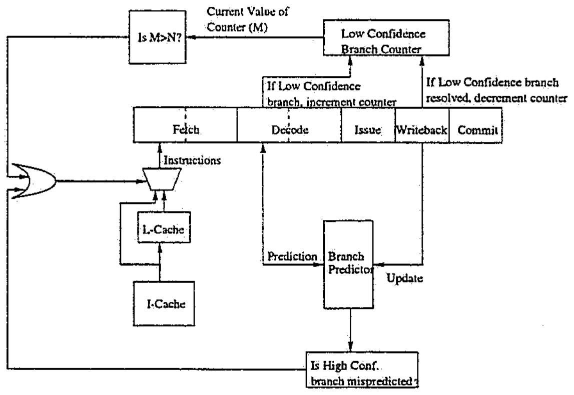

Is M>N?
Current Value of Counter (M)
Low Confidence Branch Counter
If Low Confidence branch, increment counter
If Low Confidence branch resolved, decrement counter
Fetch
Decode
Issue
Writeback
Commit
Instructions
L-Cache
I-Cache
Prediction
Branch Predictor
Update
Is High Conf. branch mispredicted?

Figure 8.13 Microarchitectural modifications for the confidence estimation method.

The management of the cache subsystem is identical to the static method. We access the I-Cache if a "high confidence" branch is mispredicted, or more than two successive "low confidence" branches are encountered. The schematic of the modified pipeline is shown in Fig. 8.13.

Tables 8.9 and 8.10 show the normalized energy and delay results for the SPEC95 benchmarks. Table 8.11 presents the percentage of dynamic instructions which cause the CPU to access the L-Cache.

This method is slightly better in terms of energy gains than the simple or the static method. The delay is lower than the previous two methods in some benchmarks and higher in some others. Likewise, the confidence estimator method is slightly worse than the filter cache in energy dissipation and slightly better in performance degradation.

Table 8.9 Energy results for the method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.181	0.174	0.096	0.119
swim	0.123	0.134	0.099	0.118
su2cor	0.208	0.188	0.139	0.149
hydro2d	0.125	0.137	0.090	0.114
applu	0.369	0.293	0.338	0.276
fpppp	0.572	0.361	0.564	0.357
go	0.642	0.548	0.609	0.529
m88ksim	0.432	0.311	0.379	0.284
compress95	0.416	0.329	0.308	0.264
li	0.435	0.344	0.386	0.303
perl	0.503	0.382	0.440	0.340

Table 8.10 Delay results for the method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.046	1.029	1.008	1.006
swim	1.029	1.017	1.015	1.008
su2cor	1.059	1.034	1.025	1.014
hydro2d	1.019	1.013	1.006	1.004
applu	1.104	1.053	1.089	1.045
fpppp	1.237	1.120	1.232	1.117
go	1.149	1.085	1.130	1.074
m88ksim	1.185	1.104	1.156	1.089
compress95	1.192	1.114	1.119	1.071
li	1.194	1.122	1.164	1.097
perl	1.232	1.142	1.194	1.117

8.4.0.4 Another method using a confidence estimator

The method described in the previous section tends to place a large number of basic blocks in the L-Cache, thus degrading performance. In modern processors, one would prefer a more

Table 8.11 Dynamic instructions that cause the L-Cache to be accessed in the confidence estimation method.

Benchmark	% dyn. instructions
tomcatv	99.77%
swim	99.95%
su2cor	98.26%
hydro2d	99.77%
applu	91.00%
fpppp	98.91%
go	67.50%
m88ksim	98.26%
compress95	93.78%
li	95.91%
perl	95.78%

Table 8.12 Energy results for the modified method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.202	0.183	0.119	0.141
swim	0.129	0.140	0.105	0.124
su2cor	0.256	0.248	0.205	0.219
hydro2d	0.138	0.151	0.105	0.130
applu	0.558	0.498	0.532	0.483
fpppp	0.602	0.405	0.595	0.401
go	0.800	0.758	0.783	0.748
m88ksim	0.473	0.361	0.419	0.334
compress95	0.563	0.498	0.486	0.452
li	0.601	0.529	0.560	0.498
perl	0.602	0.508	0.552	0.447

selective scheme in which only the really important basic blocks would be selected for the L-Cache.

We are using the same set up as before, but the selection mechanism is slightly modified as follows: the L-Cache is accessed only if a “high confidence” branch is predicted correctly. The I-Cache is accessed in any other case. We disregard “low confidence” branches altogether.

This method selects some of the very frequently executed basic blocks, yet it misses some others. Usually the most frequently executed basic blocks come after “high confidence” branches that are predicted correctly. This is especially true in FP benchmarks.

Table 8.13 Delay results for the modified method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.046	1.024	1.009	1.005
swim	1.028	1.017	1.015	1.008
su2cor	1.041	1.023	1.015	1.008
hydro2d	1.019	1.012	1.005	1.003
applu	1.082	1.043	1.069	1.035
fpppp	1.222	1.113	1.218	1.110
go	1.073	1.044	1.063	1.038
m88ksim	1.171	1.096	1.142	1.081
compress95	1.146	1.087	1.093	1.056
li	1.149	1.092	1.123	1.073
perl	1.190	1.119	1.159	1.098

Table 8.14 Dynamic instructions that cause the L-Cache to be accessed in the modified confidence estimation method.

Benchmark	% dyn. instructions
tomcatv	97.03%
swim	99.30%
su2cor	89.11%
hydro2d	97.84%
applu	65.70%
fpppp	92.06%
go	35.60%
m88ksim	90.98%
compress95	70.07%
li	69.75%
perl	77.37%

Again, Tables 8.12 and 8.13 present the normalized energy and delay results. As before, the delay results consider all the possible stalls in the R-4400 processor. Table 8.14 shows the percentage of the dynamic instructions which causes the CPU to access the L-Cache.

As expected, this scheme is more selective in storing instructions in the L-Cache, and it has a much lower performance degradation, at the expense of lower energy gains. It is probably preferable in a system where performance is more important than energy.

8.4.0.5 Using a distance estimator

Another confidence estimator which was proposed in [77] exploits the clustering of mispredicted branches. As was shown experimentally in that paper, a mispredicted branch triggers a series of succesive mispredicted branches. The degree of clustering depends on the particular program, and the predictor used. This correlation fades as more branches are executed, and is stronger immediately after the mispredicted branch. The observation that branches that follow a mispredicted branch are more probable to be mispredicted can be exploited in our scheme.

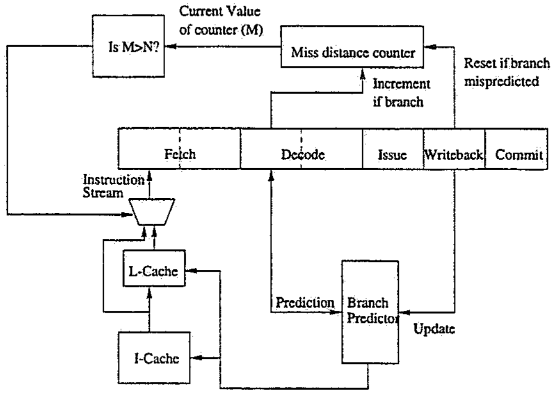


Figure 8.14 Microarchitectural modifications for the distance estimator method.

We use a counter to measure the distance of a branch from the previous, mispredicted branch. This is similar to the *miss distance counter (MDC)* proposed in [77]. The method works as follows: all N branches after a mispredicted branch are tagged as "low confidence," otherwise as "high confidence." The basic blocks after a "low confidence" branch are fetched from the I-Cache, whereas the basic blocks after a "high confidence" branch are fetched from the L-Cache. The net effect is that a branch misprediction causes a series of fetches from the I-Cache.

The parameter N was set equal to four in our experiments. It can be used to bias the mechanism towards more energy gains (smaller N) or more performance (larger N). Fig. 8.14 shows the modifications in the pipeline that are needed to implement that scheme.

Again, Tables 8.15 and 8.16 present the normalized energy and delay results. Table 8.17 shows the percentage of the dynamic instructions which causes the CPU to access the L-Cache.

This scheme is also very selective in storing instructions in the L-Cache, even more than the previous method. Provided that the L-Cache is not too small to contain the working set of

Table 8.15 Energy results for the distance confidence estimation method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.197	0.192	0.115	0.137
swim	0.126	0.137	0.103	0.122
su2cor	0.268	0.263	0.222	0.236
hydro2d	0.133	0.147	0.101	0.126
applu	0.430	0.356	0.399	0.338
fpppp	0.587	0.384	0.580	0.380
go	0.820	0.781	0.806	0.773
m88ksim	0.465	0.355	0.412	0.328
compress95	0.554	0.489	0.466	0.436
li	0.577	0.502	0.540	0.472
perl	0.569	0.481	0.531	0.450

Table 8.16 Delay results for the distance confidence estimation method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.045	1.029	1.008	1.005
swim	1.028	1.017	1.015	1.008
su2cor	1.038	1.022	1.015	1.009
hydro2d	1.018	1.012	1.005	1.003
applu	1.100	1.051	1.085	1.043
fpppp	1.229	1.116	1.225	1.114
go	1.064	1.037	1.056	1.033
m88ksim	1.169	1.094	1.139	1.079
compress95	1.141	1.083	1.081	1.047
li	1.154	1.095	1.132	1.077
perl	1.188	1.114	1.158	1.095

the program, this approach will be able to manage the L-Cache such that only the basic blocks with the larger degree of reuse will be stored there.

8.4.1 Comparison of dynamic techniques

The techniques that have been described in the previous sections present a good opportunity for dynamic management of the L-Cache using minimal extra hardware. Using the information that the branch predictor and the confidence estimation mechanisms can provide regarding the frequency of execution of different portions of the code, the machine can selectively place basic blocks in the L-Cache during program execution.

Table 8.17 Dynamic instructions that cause the L-Cache to be accessed in the distance estimator method.

Benchmark	% dyn. instructions
tomcatv	97.37%
swim	99.54%
su2cor	87.24%
hydro2d	98.21%
applu	83.54%
fpppp	95.24%
go	31.53%
m88ksim	91.31%
compress95	70.31%
li	73.39%
perl	79.49%

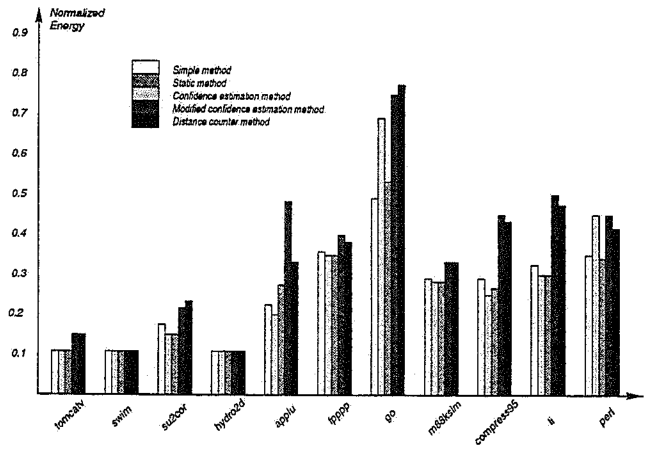


Figure 8.15 Normalized energy dissipation for the five dynamic methods. Those are the same numbers that appeared in the tables of the previous sections.

The normalized energy and delay results of the five different schemes we proposed are shown graphically in Figs. 8.15 and 8.16, respectively. A 512 bytes L-Cache with a block size of 16 bytes is assumed in all cases. The graphical comparison of the results can be used to extract useful information about each one of the five methods.

The last two methods are the most successful in reducing the performance overhead, but the least successful in energy gains. The modified method that uses the dynamic confidence estimator poses stricter requirements for a basic block to be selected for the L-Cache than the original dynamic confidence method. The method that uses the distance counter to compute

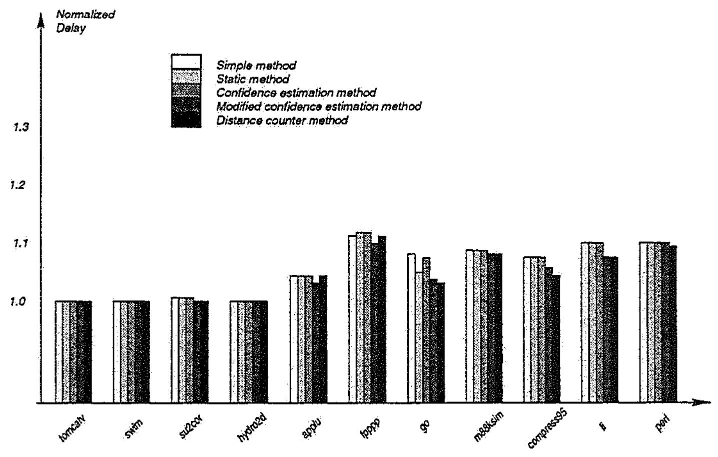


Figure 8.16 Normalized delay for the five dynamic methods. Those are the same numbers that appeared in the tables of the previous sections.

the confidence is parameterized in the distance threshold. A larger threshold favors the usage of the I-Cache, and results in smaller energy gains and smaller delay degradation.

Again, the numeric benchmarks show the largest potential for energy gains without a severe performance penalty. The dynamic techniques have a larger impact on the integer benchmarks as is shown in the two graphs. Since there is a large percentage of "low confidence" branches, the machine can be very selective when it picks up basic blocks for the L-Cache. This is why different dynamic techniques have so different energy and delay characteristics for the integer benchmarks. Regulation of the L-Cache and I-Cache utilization is more flexible in integer benchmarks.

8.5 Summary

In this chapter, we presented methods for "dynamic" selection of basic blocks for placement in the L-Cache. First, we presented an extensive overview of previous research in improving the memory hierarchy subsystem performance using dynamic techniques. Usually, extra hardware is used to keep statistics about the program execution and, accordingly, to allow or disallow the storage of specific instructions or data in the L1 caches.

Then, we proceeded by explaining the functionality of the brach prediction and the confidence estimation mechanisms in high-performance, speculative processors. After that, we explained how those mechanism can provide information to the CPU about the frequency of execution of parts of the code, and, finally, we presented five different "dynamic" techniques for the selection of the basic blocks. These techniques use existing hardware, and they try to capture the execution profile of the basic blocks by using the branch statistics that are gathered by the branch predictor.

The experimental evaluation demonstrates that the "dynamic" methods offer an attractive alternative to the compiler-based approach, especially when profiling is not possible. The energy gains are more significant than in the compiler-based approach, but not as significant as in the filter cache method. The performance degradation is also smaller than in the filter cache case.

Chapter 9

CONCLUSIONS AND FUTURE DIRECTIONS

In this research, we have developed techniques for hardware/software co-design in high-performance processors that result in energy/power reduction at the system level. To that effect, we make a more judicious use of one of the most power-consuming modules of a CPU, the I-Cache. In this chapter, we summarize our research accomplishments and we present some avenues of future research.

9.1 Thesis Contributions and Summary

Our thesis proposes a novel area for research that focuses on the energy minimization in high-performance processors using hardware/software co-design techniques. In particular, we have addressed the following problems:

(1) We proposed the insertion of an extra cache, the L-Cache, between the CPU and the I-Cache, which is used to store the most frequently executed parts of the code.

(2) We proposed and implemented compiler techniques which can be used to exploit the extra cache so that the energy gains are maximized, and performance degradation is minimized.

(3) We presented the necessary hardware for the implementation of this scheme.

(4) We developed a detailed, transistor-level energy model for the on-chip caches.

(5) We proposed and implemented an alternative compiler technique which gives better results for integer benchmarks.

(6) Finally, we developed "dynamic" techniques which do not need compiler support to select and place instructions in the L-Cache.

(7) We presented extensive experimental results for each one of the previous techniques to provide evidence for its applicability in the context of a modern processor.

More specifically, in Chapter 1 we introduced the power problem and the various sources of power dissipation in a CMOS circuit. We explained how power and performance are interrelated, and we gave evidence why the power problem is especially hard to solve in a complex processor. We also give an extensive review of related work in the area of microarchitectural and compiler techniques for power and energy minimization in microprocessors.

In Chapter 2, we demonstrated the motivation behind the proposed approach through an example, and we briefly explained our solution. The basic idea is that a small cache, the L-Cache, can be used to store and provide instructions that belong to a loop. The I-Cache can be disabled during loop execution and its energy dissipation can be saved.

The detailed compiler transformations that are needed to manage the L-Cache were presented in Chapter 3. Their main task is to remap some of the basic blocks of the code to new memory locations in the global memory address space, in order to reduce the conflicts in the L-Cache. Our approach benefits by using profiling results from previous runs of the code, in order to select the basic blocks for the L-Cache. After selection, the compiler remaps these basic blocks so that their mapping conflicts are minimized. To that effect, it uses profile information as well as the nesting relationship between the selected basic blocks. We presented algorithms for the selection of the basic blocks, for their remapping, and for their final mapping in the address space.

In Chapter 4, we presented the extra hardware needed to implement our scheme. Besides the small L-Cache, only an additional register and a multiplexer are needed. The L-Cache and the I-Cache are only accessed serially to minimize energy. In the worst case, the L-Cache will be accessed first, it will miss, and the I-Cache will be accessed with one clock cycle penalty. Our compiler-based approach ensures that the worst case does not happen often.

Extensive experimental evaluation was given in Chapter 6. We evaluated the L-Cache and the filter cache [22], and we compared them in terms of energy gains and performance degradation. We also evaluated different compiler options in selecting basic blocks for the L-

Cache. We detailed our experimental setup which was used to run the SPEC95 benchmarks. Our results show that numerical code performs better than non-numerical code in terms of energy and performance. The filter cache leads to larger energy gains, but also to higher performance degradation. We also showed how performance can be improved if we reduce the clock period of the CPU.

In Chapter 7 we focused on the integer benchmarks which, as stated before, do not perform well under the previous scheme. We explained why this is the case and, based on that, we proposed an alternative approach for selecting and storing basic blocks in the L-Cache. The experimental results show that this scheme is better than the previous one in terms of energy gains and performance degradation for integer benchmarks.

Finally, in Chapter 8, we looked into "run-time" techniques for the selection and storage of basic blocks in the L-Cache. We explained the functionality of branch prediction and confidence estimation mechanisms, and we linked them to the demands of our problem. We developed several "dynamic" techniques that use the branch history that is accumulated by the branch predictors and the confidence estimators, and we presented experimental results to prove the efficacy of our approach. Those techniques have larger energy gains than the compiler-based method, both for FP and integer benchmarks.

There is a large set of related problems which need to be investigated in this area. In the next section we present some of them.

9.2 Future Directions

In this section, we will delineate some interesting research problems in the area of microarchitectural and compiler techniques for energy reduction in modern processors.

9.2.1 Dynamic techniques

In our research we have focused on "dynamic" techniques that do not require any extra hardware. We assume that a modern processor is already equipped with a branch prediction mechanism which is used to keep track of the branch characteristics and to guide the storage of basic blocks in the extra cache. Since these techniques are used for energy reduction, any substantial hardware investment might have an adverse impact on energy.

In Section 8.1, we refered to a series of papers that deal with the problem of selectively caching data or instructions in the L1 caches. The basic idea was that if we ban infrequently accessed references from being stored in the L1 caches, we can avoid some conflict misses with frequently accessed references and thus decrease cache pollution. We cited a series of papers that propose techniques to achieve this objective [66], [67], [68]. Those techniques entail the insertion of extra hardware to keep statistics about the frequency of execution of portions of the code.

These methods can be more successful than the branch predictors in determining which portions of the code should be inserted in the L-Cache. We can keep track of which memory addresses are accessed frequently using a table of counters (the memory access table, or MAT, in [67]), and only redirect those portions in the L-Cache. Provided that the MAT is large enough, we can provide detailed information for the, and only use the L-Cache for the most heavily accessed part of the code.

However, it is not clear whether the extra hardware will offset any energy gains from the better management of the L-Cache. The extra tables should be accessed in every clock cycle, and their energy consumption is not negligible. An interesting problem is to determine if such schemes can potentially reduce the overall energy consumption, and what trade-offs between performance and energy are involved.

9.2.2 Techniques for the D-Cache

In our work, we have focused only on the problem of reducing energy for the I-Cache subsystem. This is justified from the fact that I-Caches are accessed in every clock cycle to provide an instruction (or more than one instructions for a superscalar machine), whereas D-Caches are only accessed in a load or store instruction. These instructions account for about 35% of the dynamic instructions in a set of five SPECint92 benchmarks [71]. Therefore, the number of accesses, and the energy dissipation of the D-Cache will be less than in the I-Cache.

An interesting problem is to compare the energy/performance opportunities of static and dynamic schemes for caching of data in an extra buffer placed between the D-Cache and the CPU. Since the access time of the D-Cache is often the critical path in the processor, we need to explore the impact on performance of the extra buffer for different configurations. The insertion of an extra buffer for both the I-Cache and the D-Cache will probably have large energy gains at the expense of performance.

There has been an extensive research effort aiming at increasing the hit rate of the D-Cache, and, thus, at increasing the performance of the system [80], [81], [82], [83]. The authors in [80] present a methodology and an algorithm for loops transformations that enhance the D-Cache locality. Loop blocking is an especially attractive transformation [83] which is sensitive to the size of the cache.

A series of compiler optimizations can be used to improve the efficacy of this approach. For example, loop blocking, loop exchange, and loop reversal are some optimizations that are used for locality enhancement in the D-Cache, and they can also be used here to exploit the new memory hierarchy. As before, profile data can be used to tag basic blocks within loops that are executed very often. These basic blocks will be stored in the extra cache.

In addition, the "dynamic" approach can also be applied in this case. For example, only the data accesses from the most frequently executed portion of the code can be stored in the extra cache, or only the most frequently accessed data. Again, any extra hardware that is used should not offset the gains from the utilization of the extra cache.

9.2.3 Novel architectures for low energy

A more general and still open question is what kind of architectures are more energy efficient for a given performance level. Gonzalez and Horowitz [3] suggest that the energy–delay product of a wide range of processors does not vary significantly, and therefore an increase in performance is more or less offset by an increase in energy, and vice versa. This result is pessimistic since it suggests that as processors become faster and more complex, the power will also increase, no matter what techniques are applied in the microarchitectural level.

Novel memory organizations show more promise. The intelligent RAM (IRAM) has been proposed as a low-energy alternative to the conventional memory hierarchy in a recent paper [29]. Since the main memory is in the same die with the processor, no off-chip access is necessary, and the energy of the system will drop. However, the energy consumption within the processor might increase since the IRAM is larger than the SRAM-based caches. Additional research and implementation of an IRAM-based processor is required to justify the claim that IRAM is a low-energy alternative.

Asynchronous logic has been proposed as a low-power alternative, especially for systems that suffer from a high power dissipation in the clock distribution network. Moreover, systems-on-chip will most likely be composed of many cores which will carry out tasks independently, and which will only communicate asynchronously. Thus, they will not need a globally distributed clock.

9.3 Conclusions

We believe that since performance is the most important objective of today's high-end microprocessors, no energy reduction technique will be acceptable, unless it only marginally affects the execution time, or unless its overhead can be hidden by other compiler/architectural techniques. If this is the case, even a moderate energy reduction will be welcome.

This research presented a paradigm for hardware/compiler co-design that targets activity minimization in a processor. These techniques are orthogonal to the standard circuit- or gate-level techniques that are traditionally used by designers to reduce energy and can therefore be used to further reduce energy consumption without impairing performance. This paradigm describes a more judicious use of the I-Cache unit of a processor when the flow of control is caught within a loop. The compiler is given the responsibility to restructure the code. The aim is to minimize the overlap between basic blocks that are selected to be placed in an extra cache.

Moreover, dynamic techniques that do not need compiler support were developed. They use existing hardware which is used to capture the dynamic characteristics of the program and, accordingly, to direct the basic blocks in the L-Cache or the I-Cache.

We feel that most of the energy gains in high-performance and embedded processors alike will be extracted from the high level of the design flow, when the designers have not yet committed to major design decisions. Major energy gains can be obtained if the compiler and the hardware are designed with low energy in mind. As mentioned in this work, there are problems to be addressed to make the software/hardware co-design approach we propose more effective and practical for the industrial design flow. In our opinion, this thesis is an important step since it proposes and advocates the cooperation of compiler and hardware to reduce energy in a real-life processor.

APPENDIX A

LABELTREE PROOF

Theorem 1 *The DAG that was constructed using the LabelSets is a tree.*

To prove this, we will show that there is only one path between a predecessor and a successor node in the graph. Assume a node S_1 which corresponds to the LabelSet $\{L_1, L_2, \ldots, L_k\}$, and let S_2 be a successor node of S_1. Assume that there are two paths between S_1 and S_2. We will show that this cannot be the case. Let S_3 be a node in the first path and S_4 be a node in the second path between S_1 and S_2 (S_3 or S_4 can be null). Node S_3 will correspond to a LabelSet $\{L_1, L_2, \ldots, L_k, x, \ldots\}$, i.e., it will represent basic blocks that are enclosed by the loops $L_1, L_2, \ldots, L_k, x$. Node S_4 will correspond to a LabelSet $\{L_1, L_2, \ldots, L_k, y, \ldots\}$, i.e., it will represent basic blocks that are enclosed by the loops $L_1, L_2, \ldots, L_k, y$. Then, node S_2 will correspond to LabelSet $\{L_1, L_2, \ldots, L_k, x, y, \ldots\}$ since it is a successor of both S_3 and S_4. We notice that loops x and y cannot overlap since S_4 is not a successor of S_3 or vice versa in the LabelTree. No basic block can be nested within both x and y; and therefore, node S_2 cannot have both these loops in its LabelSet. There should only be one path between S_1 and S_2; therefore, the graph is acyclic. An acyclic, connected graph is a tree. □

REFERENCES

[1] F. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 446–455, Dec. 1994.

[2] V. Tiwari and D. Singh, "Reducing power in high-performance microprocessors," *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, June 1998.

[3] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose processors," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1277–1284, Sept. 1996.

[4] J. Edmondon, "Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor," *Digital Technical Journal*, vol. 7, no. 1, pp. 119–135, 1995.

[5] N. Jouppi, "A 300-Mhz 115-W 32-b bipolar ECL microprocessor," *Journal of Solid-State Circuits*, pp. 1152–1165, Feb. 1993.

[6] A. Kalambur and M. J. Irwin, "An extended addressing mode for low power," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 208–213, Aug. 1997.

[7] V. Tiwari and S. Malik and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[8] V. Tiwari and S. Malik and A. Wolfe and T.C. Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing*, vol. 13, no. 2, Aug. 1996.

[9] T.C Lee and V. Tiwari and S. Malik and M. Fujita, "Power analysis and low-power scheduling techniques for embedded DSP software," in *Proceedings of the International Symposium on System Synthesis*, Sept. 1995.

[10] V. Tiwari and T.C. Lee, "Power analysis of a 32-bit embedded microcontroller," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Apr. 1995.

[11] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proceedings of the IEEE Symposium on Low Power Electronics*, Oct. 1994.

[12] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 72–75, Aug. 1997.

[13] J.M. Chang and M. Pedram, "Register allocation and binding for low power," in *Proceedings of the Design Automation Conference*, pp. 29–35, June 1995.

[14] C. Gebotys, "Low energy memory and register allocation using network flow," in *Proceedings of the Design Automation Conference*, pp. 435–440, June 1997.

[15] C. L. Su and C. Y. Tsui and A. Despain, "Low power architecture design and compilation techniques for high performance processors," in *Proceedings of the IEEE Computer Conference*, pp. 489–498, Oct. 1994.

[16] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, "Instruction scheduling for power reduction in processor-based system design," in *Proceedings of the Design and Test in Europe*, pp. 855–860, Oct. 1998.

[17] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[18] J. Diguet and S. Wuytack and F. Catthoor and H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 30–35, Aug. 1997.

[19] S.Wuytack and F.Catthoor and L. Nachtergaele and H. De Man, "Power exploration for data dominated video applications," in *Proceedings of the International Symposium of Low Power Electronics and Design*, Aug. 1996.

[20] S. Wuytack, F. Catthoor, and H. DeMan, "Transforming set data types to power optimal data structures," *IEEE Transcactions on Computer-Aided Design*, vol. 15, pp. 619–629, June 1996.

[21] E. Musoll, T. Lang, and J. Cortadella, "Exploiting the locality of memory references to reduce the address bus energy," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 202–207, Aug. 1997.

[22] M. G. Johnson Kin and W. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proceedings of the International Symposium on Microarchitecture*, pp. 184–193, Dec. 1997.

[23] C. Lee and M. Potkonjak and W.H. Mangione-Smith, "Mediabench: A tool for evaluating multimedia and communication systems," in *Proceedings of the International Symposium on Microarchitecture*, Dec. 1997.

[24] M. Hiraki, R. Bajwa, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Sasaki, and K. Seki, "Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 353–358, Aug. 1996.

[25] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 417–424, Dec. 1997.

[26] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power I-Cache design," in *Proceedings of the International Symposium of Low Power Electronics and Design*, Aug. 1995.

[27] I. Bahar, G. Albera, and S. Manne, "Power and performance trafeoffs using various caching strategies," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 64–69, Aug. 1998.

[28] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the memory wall: The case for processor/memory integration," in *Proceedings of the International Symposium of Computer Architecture*, pp. 90–101, June 1996.

[29] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of IRAM architectures," in *Proceedings of the International Symposium of Computer Architecture*, pp. 327–337, June 1997.

[30] J. Zawodny, E. Johnson, J. Brockman, and P. Kogge, "Cache-in-memory: A lower power alternative?" in *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 67–72, June 1998.

[31] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 305–310, Aug. 1998.

[32] V. Zyuban and P. Kogge, "Split register file architectures for inherently lower power microprocessors," in *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 32–37, June 1998.

[33] J. Patrick Brennan, Alvar Dean, Stephen Kenyon, and Sebastian Ventrone, "Low power methodology and design techniques for processor design," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 268–273, Aug. 1998.

[34] J. Montanaro et. al., "A 160-Mhz 32b 0.5W CMOS RISC microprocessor," in *Proceedings of the International Solid State Circuits Conference*, pp. 214–145, Feb. 1996.

[35] D. Dobberpuhl, "The design of a high-performance low-power microprocessor," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 11–16, Aug. 1996.

[36] S.B Furber, J.D. Garside, and S.Temple, "Power-saving features in Amulet2e," in *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 131–134, June 1998.

[37] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power M*Core architecture," in *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 145–150, June 1998.

[38] M. Gowan, L. Biro, and D. Jackson, "Power considerations in the design of the Alpha 21264 microprocessor," in *Proceedings of the Design Automation Conference*, pp. 268–273, June 1998.

[39] L. Benini and G. D. Micheli, *Dynamic Power Management: Design Techniques and CAD Tools.* Norwell, MA: Kluwer Academic Publishers, 1998.

[40] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco, "Monitoring system activity for OS-directed dynanic power management," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 185–190, Aug. 1998.

[41] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Englewood-Cliffs, NJ: Prentice Hall, 1992.

[42] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 70–75, Aug. 1998.

[43] H. Tomiyama and H. Yasuura, "Code placement techniques for cache miss rate reduction," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, Oct. 1997.

[44] P. R. Panda, N. Dutt, and A. Nicolau, "Memory data organization for improved cache performance in embedded processor architecture," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, pp. 384–409, Oct. 1997.

[45] Pohua Chang, Scott Mahlke, William Chen, and Wen-mei Hwu, "Profile-guided inline expansion for C programs," *IEEE Transcactions on Computers*, vol. 41, Dec. 1992.

[46] A. Ayers, R. Gottlieb, and R. Schooler, "Aggressive inlining," in *Proceedings of the International Conference on Programming Language Design and Implementation*, pp. 134–145, June 1997.

[47] S. McFarling, "Procedure merging with instruction caches," in *Proceedings of the International Conference on Programming Language Design and Implementation*, pp. 71–79, June 1991.

[48] William Chen, Pohua Chang, Thomas Conte, and Wen-mei Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Transcactions on Computers*, vol. 42, pp. 1045–1057, Sept. 1993.

[49] *SpeedShop User's Guide.* Silicon Graphics, Inc., 1996.

[50] K. Pettis and R. Hansen, "Profile guided code positioning," in *Proceedings of the International Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[51] Wen-mei Hwu and Pohua Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the International Symposium of Computer Architecture*, pp. 242–251, June 1989.

[52] A. Hashemi, D. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proceedings of the International Conference on Programming Language Design and Implementation*, pp. 171–182, June 1997.

[53] S. McFarling, "Program optimization for instruction caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 16–27, June 1989.

[54] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison–Wesley, 1986.

[55] P. Landman and J. Rabaey, "Activity-sensitive architectural power analysis," *IEEE Transcactions on Computer-Aided Design*, vol. 15, pp. 571–587, June 1996.

[56] P. Landman and J. Rabaey, "Architectural power analysis: The dual bit method," *IEEE Transactions on VLSI Systems*, vol. 3, pp. 173–187, June 1995.

[57] M. Camble and K. Ghose, "Analytical energy dissipation models for low power caches," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 143–148, Aug. 1997.

[58] D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 663–670, June 1994.

[59] U. Ko, T.Balsara, and A. Nanda, "Energy optimization of multilevel cache architectures for RISC and CISC processors," *IEEE Transactions on VLSI Systems*, vol. 6, pp. 299–308, June 1998.

[60] N. Bellas, I. Hajj, and C. Polychropoulos, "A detailed, transistor-level energy model for SRAM-based caches," in *Proceedings of the International Symposium on Circuits and Systems*, Aug. 1999.

[61] S. Wilson and N. Jouppi, "An enhanced access and cycle time model for on-chip caches," tech. rep., DEC WRL 93/5, July 1994.

[62] A. Hasegawa et. al., "Sh3: High code density, low power," *IEEE Micro Magazine*, pp. 11–19, Dec. 1995.

[63] N. Bellas, I. Hajj, and C. Polychronopoulos, "A new scheme for I-Cache energy reduction in high-performance processors," *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 50–54, June 1998.

[64] J. E. Veenstra and R. J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 201–207, Jan. 1994.

[65] M. Garey and D. Johnson, *Computers and Intractability.* New York, NY: W.H.Freeman & Co., 1979.

[66] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A modified approach to data cache management," in *Proceedings of the International Symposium on Microarchitecture*, pp. 93–103, Dec. 1995.

[67] Teresa Johnson and Wen-mei Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the International Symposium of Computer Architecture*, 1997.

[68] Teresa Johnson, Matthew Merten, and Wen-mei Hwu, "Run-time spatial locality detection and estimation," in *Proceedings of the International Symposium on Microarchitecture*, June 1997.

[69] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[70] T. Harvey and K. Cooper, "Compiler-controlled memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–11, Oct. 1998.

[71] J. Hennesy and D. Patterson, *Computer Architecture–A Quantitative Approach.* San Francisco, CA: Morgan Kaufmann, 1996.

[72] T. Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the International Symposium of Computer Architecture*, pp. 257–266, June 1993.

[73] T. Y. Yeh and Y. N. Patt, "Alternative implementations of a two-level adaptive branch prediction," in *Proceedings of the International Symposium of Computer Architecture*, pp. 124–134, June 1992.

[74] S. T. Pan, K. So, and J.T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, Oct. 1992.

[75] S. McFarling, "Combining branch predictors," tech. rep., DEC WRL 93/5, June 1993.

[76] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 142–152, Dec. 1996.

[77] D. Grunwald, A. Klauser, S. Manne, and A. Plezskun, "Confidence estimation for speculation control," in *Proceedings of the International Symposium of Computer Architecture*, pp. 122–131, June 1998.

[78] S. Manne and D. Grunwald and A. Klauser, "Pipeline gating: Speculation control for energy reduction," in *Proceedings of the International Symposium of Computer Architecture*, pp. 132–141, June 1998.

[79] M. Lipasti and J. P. Smith, "Superspeculative microarchitecture for beyond ad 2000," *IEEE Computer*, vol. 30, pp. 59–66, Sept. 1997.

[80] M. Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transcactions on Parallel and Distributed Systems*, pp. 452–471, Oct. 1991.

[81] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proceedings of the International Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.

[82] S. Carr, K. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 252–262, Oct. 1994.

[83] M. Lam, E. Rothberg, and M. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, Oct. 1991.

VITA

Nikolaos Bellas was born in Thessaloniki, Greece, in 1969. He received his Diploma degree in computer engineering and informatics from the University of Patras, Greece, in 1992. Since the fall of 1993 he has been employed as a research assistant in the Coordinated Science Laboratory and the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign (UIUC). He received the M.S. degree in 1995 and the Ph.D. degree in 1998 from UIUC. He has been a summer intern at MIPS Corp. in 1995 and at Intel Corp. in 1996. His research interests are design for low power, computer architecture, and compilers.