HARDWARE/SOFTWARE TECHNIQUES FOR MEMORY POWER OPTIMIZATIONS IN EMBEDDED PROCESSORS

by

Rajiv A. Ravindran

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:

Associate Professor Scott Mahlke, Chair
Professor Richard B. Brown
Professor Trevor N. Mudge
Associate Professor Dennis M. Sylvester
Assistant Professor Chandrasekhar Boyapati

To my family

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude towards every individual who has contributed to this dissertation, and helped me both academically and personally during my graduate student years at the University of Michigan, Ann Arbor.

Firstly, I would like to thank my adviser, Prof. Scott Mahlke, for his priceless guidance, patience, and support throughout my graduate studies. I had the privilege of working with him since my first summer internship at the Hewlett Packard Labs, Palo Alto in 2000. It was he who motivated me into the field of backend compilation and encouraged me to think, solve, and clearly articulate my research problems.

Next, I would like to thank the members of my doctoral committee Prof. Brown, Prof. Chandra, Prof. Mudge, and Prof. Sylvester. Their invaluable comments and insights helped to improve the quality of my thesis. I would especially like to thank Prof. Brown, with whom I had the honor of collaborating on my research. His vision of a system-level design helped to broaden the scope of my thesis and make it applicable under real engineering circumstances.

I am greatly indebted to the Center for Wireless Integrated Microsystems (WIMS) at the University of Michigan, Ann Arbor for funding me throughout my graduate school. The WIMS microprocessor formed the basis of my research. Working with the environmental and cochlear testbeds provided me with some invaluable hands-on experience which will always be of help to me in my future research career.

I would like to thank the support of my colleagues in the WIMS project, especially,

Eric Marsman and Robert Senger. They helped in numerous ways by providing a solid platform to explore my research. I would also like to thank my colleagues on the compiler side, Ganesh Dasika and Pracheeti Nagarkar, for all their help in developing and debugging the WIMS Trimaran compiler and the supported software development framework. I am thankful to all the members of the CCCP group for those intellectually stimulating brainstorming sessions, the positive feedback on my research, and also for making my stay at Ann Arbor memorable.

I would like to thank my brother, Arun, for encouraging me towards a research career and my parents for their support. I will be forever grateful to the constant support, love, and care of my wife, Mansi, that helped me remain focused on my research goals.

TABLE OF CONTENTS

DEDICATION . . . . . **ii**

ACKNOWLEDGEMENTS . . . . . **iii**

LIST OF FIGURES . . . . . **viii**

LIST OF TABLES . . . . . **xi**

LIST OF ALGORITHMS . . . . . **xii**

ABSTRACT . . . . . **xiii**

CHAPTER

I. Introduction 1

1.1 Compiler Controlled Windowed Register File Architecture . . . 4
1.2 Compiler Managed Dynamic Instruction Placement in Scratch-Pad Memories 5
1.3 Compiler Managed Partitioned Data Cache Management 6
1.4 Contributions 7
1.5 Organization 8

II. WIMS Microcontroller Architecture 9

III. Reducing Spill Pressure Using A Windowed Register File Architecture 14

3.1 Introduction 14
3.2 Windowed Architecture Example 17
3.3 Register Window Partitioning 19
3.3.1 Overview 19
3.3.2 Edge Weight Calculation 23
3.3.3 Partition Weight Calculation 26
3.3.4 Node Partitioning 28
3.3.5 Partitioning Algorithm Optimizations 35

3.3.6 Window Swap Insertion and Optimization 37
3.4 Experimental Results 39
3.4.1 Methodology 39
3.4.2 Results 40
3.4.3 Comparison among different partitioning heuristics of varying estimation accuracy 49
3.5 Related Work 52
3.6 Conclusion 54

IV. Compiler Managed Dynamic Instruction Placement In A Low-Power Scratch-Pad Memory 56

4.1 Introduction 56
4.2 Dynamic Placement Motivation 60
4.3 Dynamic Placement 62
4.3.1 Overview 62
4.3.2 Trace Selection/Placement 63
4.3.3 Copy Placement 71
4.4 Experimental Evaluation 79
4.4.1 Methodology 79
4.4.2 Results 82
4.4.3 Comparison to ILP-based Solutions 90
4.5 Conclusion 92

V. Compiler Managed Partitioned Data Cache Architecture 94

5.1 Introduction 94
5.2 Background 96
5.3 Partitioned Cache Architecture 99
5.4 Compiler Partitioning of Memory Instructions 105
5.4.1 Cache Estimation 106
5.4.2 Cache Assignment 116
5.4.3 Partitioning for Scratch-pads 119
5.5 Experimental Evaluation 120
5.5.1 Methodology 120
5.5.2 Results 122
5.6 Related Work 129
5.7 Conclusion 133

VI. Conclusion 135

6.1 Summary 135
6.2 Putting it All Together 138
6.2.1 Methodology 138

6.2.2 Results 139
6.2.3 Discussion 142
6.3 Future Directions 144

BIBLIOGRAPHY **147**

LIST OF FIGURES

Figure

2.1 The WIMS microcontroller in TSMC 0.18μ CMOS and the WIMS datapath. 10

3.1 Register window example. (a) C-source; Assembly code for (b) 1-window of 8-registers (c) 1-window of 4-registers (d) 2-windows of 4-registers. Registers are denoted by window number '-' the allocated register number. 17

3.2 Overview of the compiler system with extensions to support register windows. 20

3.3 Example loop to illustrate the register window partitioning algorithm. . 22

3.4 Example of partition and edge weight calculations. (a) Edge weight matrix: each cell contains $\{$swap cost, move cost$\}$. (b) Spill cost of VRs. (c) Partition weight computation assuming all VRs assigned to one partition. 23

3.5 Partitioning applied to example: (a) Initial partition, (b) Gains for each VR moving from P1 to P2 after the initial partition. 33

3.6 Example after window assignment. The notation $VR : R_i - j$ is used, where VR is the original virtual register number from Figure 3.3, i is the window number, and j is the allocated register number. 34

3.7 Performance improvement shown as percent reduction in cycles for the 8-register WIMS processor (top) and VLIW processor (bottom) for 2 and 4 window designs. For each benchmark, the results for w2.r8 and w4.r8 are plotted relative to w1.r8. 41

3.8 Performance analysis for the 8-register WIMS processor (top) and VLIW processor (bottom). For each benchmark, w2.r8 is plotted relative to w1.r8. 43

3.9 Performance improvement shown as percent reduction in cycles in increasing the number of register windows in a 4-register WIMS processor (top) and VLIW processor (bottom). For each benchmark, three sets of data are shown: w2.r4 (left), w4.r4 (middle), and w8.r4 (right), plotted relative to w1.r4. 45

3.10 Performance degradation while partitioning a 16-entry register file into 2 and 4 windows for WIMS (top) and VLIW (bottom). For each benchmark, w2.r8 and w4.r4 are plotted relative to w1.r16. 46

3.11 Percent dynamic energy improvement of w2.r8 and w4.r8 over the base case of w1.r8. 49

3.12 Comparing performance between *fast*, *region*, and *global* heuristics for the 8-register WIMS processor (top) and the VLIW processor (bottom). For each heuristic, w2.r8 is plotted relative to w1.r8. 51

4.1 Example (a) weighted control flow graph (b) static allocation (c) dynamic allocation as a function of time. 60

4.2 Overall compiler system for dynamic instruction placement in scratch-pad. 62

4.3 Trace selection and placement example. (a) CFG (b) Benefit and Copy-Cost computation for traces T1, T2, and T3. 64

4.4 Trace selection and placement example. (a) Dynamic execution trace (b) Temporal relationship graph. (c) Edge weight calculation between nodes T1 and T2 (d) Placement of T1, T2, and T3 into the SP. 67

4.5 Copy placement example. (a) Initial copies inserted (b) Hoisting of copies and live-range computation 72

4.6 Final copy placement for example in Figure 4.5(a). 77

4.7 Comparing energy savings and hit rate of static and dynamic over on-chip main memory for the WIMS processor. 83

4.8 Effect of varying scratch-pad size on energy savings and hit rate over on-chip main memory for the WIMS processor. 85

4.9 Comparing energy savings and hit rate of static, dynamic, and icache over off-chip main memory using CACTI. 86

5.1 Hardware/software co-managed vertically partitioned cache 100

5.2 (a) Example kernel to illustrate compiler-managed cache partitioning. (b) Fused load/store instructions. 105

5.3 (a) Trace consisting of array references, cache blocks, and load/stores from the example in Figure 5.2. (b) Reuse distance (D) for each fused instruction. (c) Hit-rate estimate for different cache configuration using Equation 5.1. 110

5.4 (a) Interference graph with cliques shaded. (b) The new reuse-distance for each of the cliques. (c) Assignment of loads/stores to partitions/ways. (d) Annotating with partition bit-vectors. 114

5.5 (a) Average data/tag-array accesses and partition assignment for different cache sizes and configurations and (b) Percentage reduction in energy for a 1-Kb cache 123

5.6 For a 1-Kb cache, (a) percentage reduction in energy-delay and (b) percentage annotated ld/st instructions and percent code size overhead. . . 125

5.7 Percentage energy-delay reduction of partition data cache vs. way-prediction vs. oracle way-prediction for a 1-Kb cache 4-way associative baseline cache 127

5.8 Percentage energy reduction for different scratch-pad (sp) configurations w.r.t a 4-way cache of 4-Kb 129

6.1 Combined energy savings on the WIMS processor using 2-windows of 8-registers, a 512-byte loop cache, and a 1-Kb partitioned data cache. . 140

6.2 Pareto optimal graph with area overhead incurred on the x-axis and the energy savings obtained on the y-axis for different window, loop-cache, partitioned data cache configurations. 141

LIST OF TABLES

Table

3.1 Per instruction class energy and execution time for the WIMS processor at 100MHz. 48

4.1 Per access scratch-pad energy for the WIMS processor (top) and per access scratch-pad and icache energy using CACTI (bottom, .18μm) for different sizes. 81

4.2 The left table shows the code cache size (in bytes) for at least 95% hit rate for static, dynamic, and icache schemes for the CACTI energy models. The right table shows the scratch-pad size required for highest energy gains in the WIMS processor. 88

4.3 The scratch-pad size at maximum energy gain of dynamic over static. . 89

4.4 Benchmark characteristics on a 256-byte scratch-pad for dynamic allocation showing number of hot traces selected, fraction of the total number of traces, dynamic execution frequency, average number of trace overlaps per chunk in scratch-pad, and percent performance degradation due to copy overhead. 90

4.5 Size, time taken, and percent energy gain over the dynamic scheme for a full program ILP-formulation using CPLEX on a 1-GHz UltraSPARC-IIIi processor. '*' denotes failure to complete within 72 hours of run-time. 92

5.1 Misses/1000 instructions with different 1-Kb partitioned cache configurations 121

LIST OF ALGORITHMS

Algorithm

3.1 Initial partitioning of node (VRs). 29

3.2 Node (VR) partitioning. 31

3.3 Find the best VR to be moved to a different partition. 32

4.1 Pseudo code to select and place traces in the scratch-pad. 69

4.2 Compute_Copy_Cost function that computes the cost of copying trace T 69

4.3 Pseudo code to hoist and eliminate redundant copies. 75

5.1 Estimating working-set size of load/store instructions 111

ABSTRACT

Power has become one of the primary design constraints in modern microprocessors. This is all the more true in the embedded domain where designers are being pushed to create faster processors that operate for long periods of time on a single battery. It is well known that the memory sub-system is responsible for a significant percentage of the overall power dissipation. For example, in the StrongARM-110 and the Motorola MCORE, the memory sub-system consumes more than 40% of the overall system power. Traditional power savings techniques like voltage/frequency scaling tend to sacrifice performance for power. But, current generation embedded processors perform computationally intensive tasks like audio, video, or packet processing that require high performance. Hence, power savings should not be at the price of performance.

On-chip memory structures, such as register files, caches, and scratch-pads, provide fast and energy efficient access to program and data by reducing the slower and power hungry off-chip accesses. Memory power, therefore, can be reduced by employing techniques to effectively utilize these storage elements. In this dissertation, three different compiler orchestrated, but hardware-assisted techniques, are proposed that target these on-chip storage elements while being performance neutral: use of a windowed register file to provide more physical registers without increasing code size, a software-managed loop-cache to reduce instruction fetch energy, and a compiler-controlled partitioned data cache architecture to eliminate redundant data and tag accesses of a traditional data cache. The compiler utilizes whole program knowledge to better orchestrate the program, while

hardware support allows run-time adaptation and enhances execution efficiency. These techniques are evaluated within the context of the WIMS microprocessor, which is used in sensor-based systems and thus, has a very tight power budget.

Registers provide faster, low-power storage for program variables. By maximizing register utilization, the burden on the memory system can be significantly reduced. A windowed register file architecture that provides a large number of physical registers without compromising on the instruction encoding is proposed. A novel graph partitioning compiler algorithm that partitions virtual registers within a given procedure across multiple windows was designed and implemented. Allocating program values across multiple windows help reduce spill loads/stores to memory, thus improving both power and performance. On average, a 25% reduction in system energy and 11% improvement in performance were recorded as an eight-register design is scaled from one window to four windows.

Modern embedded microprocessors use low power on-chip SRAMs called scratch-pad memories to store frequently executed instructions or data. Unlike traditional instruction or data caches, scratch-pads lack the complex tag checking and comparison logic, thereby proving to be efficient in area and power. To effectively utilize the limited scratch-pad space, a compiler-managed dynamic instruction placement algorithm was designed wherein, multiple hot code sequences, or traces, are made to overlap each other in the scratch-pad at different points in time during execution through specially provided copy instructions. For a 64-byte scratch-pad, the compiler-managed dynamic scheme achieved over 64% energy improvement over a static-based solution.

Data caches have been effective in dealing with more irregular data access patterns. But, they employ hardware-based lookup and replacement schemes that are inflexible and have high energy overheads. A hardware/software co-managed partitioned cache architec-

ture is proposed in which enhanced load/store instructions are used to control fine-grain data placement within a set of cache partitions. In comparison to traditional partitioning techniques, load and store instructions can individually specify the set of partitions for lookup and replacement. This fine grain control can avoid conflicts, thus providing the performance benefits of highly associative caches, while saving energy by eliminating redundant tag- and data-array accesses. Using four direct-mapped partitions, 25% of the tag/data-array checks were eliminated that resulted in an average 15% reduction in the energy-delay product compared to a hardware-managed 4-way set-associative cache.

CHAPTER I

Introduction

With the proliferation of cellular phones, digital cameras, PDAs, and other portable computing systems, power consumption in microprocessors has become a dominant design concern. Power consumption directly affects both battery lifetime and the amount of heat that must be dissipated and thus, it is critical to create power-efficient designs. However, many of these devices perform computationally demanding processing of images, sound, video, or packet streams. Thus, simply scaling voltage and frequency to reduce power is insufficient as the desired performance level cannot be achieved. Hardware and software solutions that maintain performance while reducing power consumption are required.

In this dissertation, an important component of the overall system power, namely, the memory sub-system power, is explored. Recent research has shown that the memory subsystem, including the instruction and data cache, is the highest contributer to the overall system power. Caches consume around 42% and 23% of the total processor power in the StrongARM 110 and the Power PC [100] processors, respectively.

Current techniques to attack this growing memory power problem can be classified into

hardware, software, or a hybrid hardware/software solutions. Common hardware solutions include memory banking [32, 68], dynamic voltage/frequency scaling [49], dynamic cache resizing [66], and reducing memory bus switching power [10, 17]. Software-based solutions include the use of software-controlled scratch-pads [8, 43, 71, 86, 87, 103] and data/code reorganization [69, 70]. Although these techniques have their own merit, their use and effectiveness has been limited. Hardware-only techniques suffer from the disadvantage of adding complex structures that can complicate the design and verification process. In addition, hardware techniques must often resort to local program state information, like the program execution history. This localized view can result in sub-optimal solutions.

On the contrary, software-only techniques tend to make conservative decisions to ensure correctness. Moreover, they are limited to analyzing programs with highly constrained code and memory access behavior. This includes limiting the analysis to array-only code that are indexed through affine functions [44], ignoring pointer/heap based code and recursive function calls [98], and restricting analysis to inner-most loops with no procedure calls or control-flow [44, 103].

In this dissertation, compiler-controlled hardware-assisted techniques to efficiently use on-chip memory structures are proposed. Software-based management can help reduce hardware inefficiencies using global knowledge of the program behavior. Hardware assistance can help reduce software overheads while capturing dynamic program behavior, and thus aid in making software techniques more aggressive and effective.

On-chip storage structures, such as register files, caches, and scratch-pad memories,

supply instruction and data values to the processor. They form a hierarchy of storage structures that are progressively smaller in size while being faster and more energy efficient. Off-chip memories are slower as they are larger in size and are located farther away from the processor. In addition, driving high capacitance off-chip buses requires a large amount of energy. Thus, hardware and software techniques generally try to reduce off-chip accesses as aggressively as possible to drive down the energy consumed.

Register files are positioned closest to the functional units. They are usually small, typically consisting of 16 to 32 entries, and are directly accessed. Register files are the fastest and the most energy efficient of all on-chip storage structures. Almost all instructions source their operands from the registers. Registers reduce memory demand and help save energy. Compilers, thus try to allocate as many program values to the registers so as to maximally utilize them. But, more number of physical registers can increase the number of bits required within the instruction encoding. This in turn increases the code size, and thus the instruction memory power.

Instruction and data memories represent the next level of storage within a processor. Instruction fetch is one of the most active portions of a processor as instructions are fetched almost every cycle. Hence, retaining as many of the instruction fetches on-chip is desirable. Similarly, data memories store frequently accessed data values that cannot fit in the registers. Although data accesses are less frequent than instruction accesses, they typically have a much larger memory footprint and exhibit more irregular access patterns. Caches and scratch-pad memories are commonly deployed solutions to retain frequently accessed instructions and data within the chip. One of the primary objectives of hardware and soft-

ware techniques is to effectively manage these memory hierarchies so as to reduce off-chip accesses that are slower and energy inefficient.

The focus of this dissertation is to explore hardware/software techniques to effectively utilize these on-chip storage structures to reduce the overall memory system power. The proposed techniques attempt to intelligently utilize the higher levels of the memory hierarchy so as to provide faster and energy efficient accesses to instructions and data. In particular, three different compiler-managed hardware-assisted techniques targeting registers, caches, and scratch-pads are investigated. The following sections briefly summarize them.

1.1 Compiler Controlled Windowed Register File Architecture

In order to reduce the code footprint and thus the instruction fetch power, modern embedded processors use instruction sets with narrow encoding (eg., 8 or 16-bits per instruction). But, reduced encoding limits the bits available to specify source and destination operand specifiers, thus restricting the number of architected registers to a small number (e.g., eight or less). Restricting the number of addressable registers often limits performance by forcing a large fraction of program variables/temporaries to be stored in memory. Spilling to memory is required when the number of simultaneously live program variables and temporaries exceeds the register file size. This has a negative effect on power consumption as more burden is placed on the memory system to supply operands each cycle.

A windowed register file architecture provides a large number of physical registers while retaining the instruction encoding. At any point in execution, only one of the win-

dows is active. Special instructions are used to activate and move data between windows. Traditionally, register windows have been used to reduce the register save and restore overhead at procedure calls, such as in the SPARC [85] and the IA-64 architectures [41]. Embedded microprocessors use windowing techniques to reduce context switch overhead. In this dissertation, a non-traditional approach to using register windows is proposed. A graph partitioning technique is used to partition program variables inside a procedure to multiple windows so as to aggressively reduce spill code while minimizing the overhead of the window management instructions.

1.2 Compiler Managed Dynamic Instruction Placement in Scratch-Pad Memories

The instruction fetching subsystem can contribute to a large fraction of the total power dissipated by the processor. Hardware managed caches, referred to as filter, L0 caches [50], or loop caches [56, 57], suffer from high miss rates, controller complexity, and the inability to relocate loops with control flow or subroutine calls. Software managed scratch-pad memories [71, 98] reduce hardware management overhead by relying on the compiler to relocate code segments. By avoiding the complex tag checking and comparison logic and through better placement, scratch-pad memories have proven to be more energy efficient than traditional caches.

Software static schemes place frequently executed regions of the code prior to program execution. The contents of the scratch-pad are never modified after initial placement. Although this technique is effective for particular types of applications, it breaks down when a program has multiple important loops that collectively cannot fit in the scratch-

pad.

In this dissertation, a compiler-directed technique for dynamic management of scratch-pads is designed and evaluated. An inter-procedural heuristic for identifying hot instruction traces to insert in the scratch-pad is proposed. Based on a profile-driven power estimate, the selected traces are packed into the scratch-pad by the compiler, possibly sharing the same space. Copy instructions are used to overlap multiple traces in the scratch-pad at different points in time during execution such that the run-time cost due to copying the traces is minimized.

Compiler-directed dynamic placement combines the benefits of the hardware-based schemes with the low overhead of the software-based schemes. The dynamic scheme packs more frequently executed code regions into the scratch-pad, and thus better utilizes the available space as compared to the static schemes.

1.3 Compiler Managed Partitioned Data Cache Management

Data caches have proven to be effective as they help to dynamically capture both temporal and spatial locality without software intervention. But, their use in embedded domains have been limited due to their energy inefficient tag checking and comparison logic. On the one hand, although set-associative caches achieve high hit-rates, they come at the expense of high energy overhead. On the other hand, direct mapped caches consume much less power per access, while incurring more misses.

A partitioned cache architecture is proposed that attempts to bridge the performance and energy gap between direct and set-associative caches. By maintaining multiple smaller direct mapped caches with the same size as a unified direct or set-associative cache, per-

formance can be improved while reducing the energy consumed. Data placement within these partitioned caches are controlled by the compiler through load/store instruction set extensions. Each load/store instruction can independently specify the partitions that are to be probed for data lookup and replacement. This provides a two-fold advantage. Firstly, by restricting cache lookups, tag/data access energy can be reduced. Secondly, by carefully orchestrating data placement in different partitions, conflict misses are reduced that can improve performance.

Further energy reduction is achieved by allowing each of the partition to be configured as a software-controlled scratch-pad. The data-arrays are exposed as part of the physical address space. By disabling the tag-arrays of selected partitions, a highly configurable data memory that can be tuned to the application's memory needs is provided.

1.4 Contributions

This dissertation proposal makes the following contributions:

- Use of a new graph partitioning based compiler technique to exploit a windowed register file architecture to reduce spill code.
- A compiler managed dynamic code placement scheme to place multiple frequently accessed code sequences within on-chip scratch-pad memories.
- Use of a compiler directed partitioned cache architecture to reduce the energy dissipated, while maintaining or improving performance.

1.5 Organization

The remainder of this dissertation report is organized as follows. Chapter II provides a brief description of the WIMS microcontroller. The WIMS microcontroller is used for environmental sensing and in biomedical implants and hence has a very tight power budget. This architecture forms the basic platform for our explorations and evaluations into hardware/software techniques for low power. The windowed register file architecture and the compiler algorithm used to exploit it is described in Chapter III. Chapter IV discusses and evaluates compiler managed dynamic placement of code in the scratch-pad memory. In Chapter V, techniques to reduce data memory power using partitioned data caches that are managed by the compiler is investigated. Finally, conclusions and some future directions of research are presented in Chapter VI.

CHAPTER II

WIMS Microcontroller Architecture

The WIMS Microcontroller was designed to control a variety of miniature, low-power embedded sensor systems [60]. It was designed at the University of Michigan, and forms a central part of a high performance low power microsystem used for remote environmental monitoring and in cochlear implants [14]. The microcontroller, fabricated in TSMC 0.18μm CMOS, is shown in Fig. 2.1 and consists of three major sub-blocks: the digital core, the analog front-end (AFE), and the CMOS-MEMS clock reference. Power minimization was a key design constraint for each sub-block.

A 16-bit load/store architecture with dual-operand register-to-register instructions was chosen to satisfy the power and performance requirements of the microcontroller. The 16-bit datapath was selected to reduce the complexity and power consumption of the core while providing adequate precision in calculations, given that the sensors controlled by this chip require 12 bits of resolution. The datapath pipeline consists of three stages: fetch, decode, and execute. Typically in sensor applications, processing throughput requirements are minimal and power dissipation is a key design constraint; therefore, clock frequencies should be kept as low as possible. A three-pipeline-stage architecture was chosen

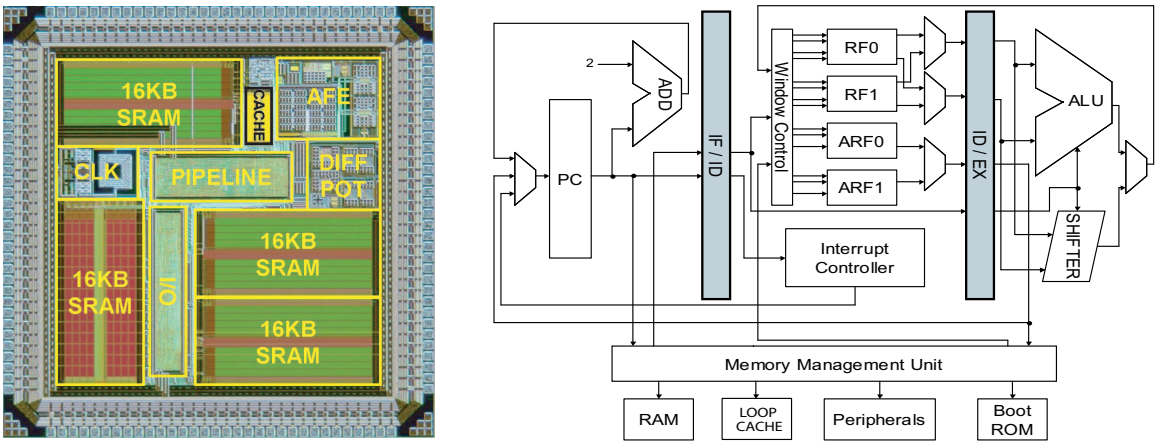


Figure 2.1: The WIMS microcontroller in TSMC 0.18μ CMOS and the WIMS datapath.

to obtain adequate performance without incurring the hardware overhead of more deeply pipelined machines. A unified 24-bit address space for data and instruction memory satisfies the potentially large storage requirements of remote sensor systems. The 16MB of supported memory is byte addressable and provides sufficient storage for program, data, and memory-mapped peripheral components. The current implementation has four 16KB banks of on-chip SRAM with a memory management unit that disables inactive banks of memory. This memory topology permits simultaneous instruction fetch and data accesses to different banks of memory without stalling the pipeline.

A 16-bit WIMS instruction set was custom designed and includes seventy-seven instructions and eight addressing modes. The 16-bit instruction encoding supports a diverse assortment of instructions that would be unrealizable with just 8-bit encodings. In contrast, 32-bit instructions require twice as much power to fetch from memory and the additional 16-bits would not be efficiently utilized by the applications that typically run on low-power embedded processors. The 16-bit encoding represents an intelligent compromise between the power required to fetch an instruction from memory and the versatility of the instruc-

tion set. Some two-word instructions are necessary to support 24-bit absolute addressing modes with 16-bit instructions. Address update modes facilitate manipulation of the 24-bit addresses stored in the address registers by allowing both pre- and post-update operations. Load and store instructions are available with or without update and in word or byte mode.

The core contains sixteen 16-bit data registers that are split into two register windows each containing eight data registers (RF0, RF1). Similarly, six 24-bit address registers are evenly split into two register windows (ARF0, ARF1). This windowing scheme permits instructions to be encoded in 16 bits by reducing the number of bits required to encode the sixteen register operands from 4 bits to 3 bits. In general, instructions can access only one register window at a time. The only exceptions are the non-windowed instructions which are used to copy data and addresses between the two windows. A window bit stored in the Machine Status Register (MSR) selects the active register window. Additional window bits can be added to the MSR to support extra register windows. A special instruction (WSWAP) switches register windows in a single cycle by changing the MSR window bit setting. Three additional non-windowed address registers (a stack pointer, frame pointer, and link register) are provided for subroutine support.

The WIMS design is somewhat atypical of most processors used in previous research in that it contains no caches. Caches were not needed because memory accesses to the on-chip SRAM banks complete in one cycle. Moreover, the area/power overhead associated with the tag memory and logic for tag comparisons of conventional cache organizations did not make sense in the design. However, instruction fetch contributes to a large fraction of the overall power dissipation of the chip (around 30% for the WIMS processor),

thus a simple, software-managed scratch-pad (SP) or loop cache (LC) was added to the design. The LC is a small SRAM (512-bytes in the current design) that can be designed with substantially lower power dissipation characteristics than the 16KB banks used for the rest of the memory ("main memory"). Fig. 2.1 shows the architecture block diagram of the WIMS microcontroller with the LC. The LC occupies a range of the physical address space, thus copying instructions into the LC corresponds to copying instructions into those specific physical addresses. The memory management unit treats the LC as another memory bank in terms of routing requests to it.

The WIMS microsystem operates under constrained environments with no access to a constant source of power supply. Hence, it is imperative that it operate at ultra low power in order prolong the limited battery life. Thus, the WIMS microcontroller architecture, with a tight energy budget, forms an ideal platform to explore hardware/software co-design techniques to optimize power at all levels of the design. We have implemented compiler managed schemes to effectively exploit the above described register windows and loop cache architectural features. Data caches have become very common within embedded processors, although the current generation WIMS processor does not include one. To make our techniques applicable to a wider class of embedded designs, we have investigated compiler/hardware techniques to effectively manage the data cache for power and performance.

We use the WIMS platform as our basic infrastructure to evaluate and explore different techniques towards memory power optimizations. Many of the above mentioned architectural features are common in most current generation embedded processors. Thus, our

solutions are applicable to improve power and performance for a broader class of embedded systems.

CHAPTER III

Reducing Spill Pressure Using A Windowed Register File Architecture

3.1 Introduction

In the embedded processing domain, in order to create energy efficient designs, a common approach employed by designers is to create narrow bitwidth instruction designs (eg., 8 or 16 bits per instruction). Examples of such designs include the Motorola-68HC12 [65] and the Thumb instruction set extensions in the ARM [80] processor. Tight instruction encodings offer the advantage of compact code and thus smaller instruction memory requirements.

While reduced code size can save area and power, the performance of such systems can be problematic. Many embedded applications, such as signal processing, encryption, and video/image processing, have significant computational demands. Low-power designs are often unable to meet the desired performance levels for these types of applications. This thesis focuses on one particular aspect in the design of narrow bitwidth processors, the architected registers. An instruction-set with limited encoding (8 or 16-bits) significantly reduces instruction fetch power by reducing the code footprint. But reduced encoding lim-

its the bits available to specify source and destination operand specifiers, thus restricting the number of architected registers to a small number (e.g., eight or less). For example, the TMS320C54x [92] processor has eight address registers and the ADSP-219x [5] has sixteen data registers. Similarly, the Thumb mode in the ARM [80] architecture uses a 16-bit instruction encoding with eight addressable registers. Restricting the number of addressable registers often limits performance by forcing a large fraction of program variables/temporaries to be stored in memory. Spilling to memory is required when the number of simultaneously live program variables and temporaries exceeds the register file size. This has a negative effect on power consumption as more burden is placed on the memory system to supply operands each cycle.

Our approach is to provide a larger number of physical registers than allowed by the instruction set encoding. This approach has been designed and implemented within the low-power, 16-bit WIMS (Wireless Integrated Microsystems) microcontroller (see Chapter II). The registers are exposed as a set of identical register windows in the instruction set. At any point in the execution, only one of the windows is active, thus operand specifiers refer to the registers in the active window. Special instructions are utilized to activate and move data between windows. The goal is to provide the appearance of a large monolithic register file by judiciously employing the register window.

Traditionally, register windows have been used to reduce the register save and restore overhead at procedure calls, such as in the SPARC architecture [85]. A similar but more configurable scheme called the Register Stack Engine (RSE) is implemented in the IA-64 architecture [41]. The register stack supports a variable sized window for each procedure,

wherein the size is determined by the compiler and communicated to the hardware through special instructions. When the number of physical registers is exceeded, a hardware engine is invoked to save and restore the registers to memory. The RSE is primarily targeted at reducing the save/restore overhead incurred by procedure calls. Windowing techniques have also been employed in embedded microprocessors, including the ADSP-219x [5] and Tensilica's Xtensa [91]. These processors typically use register windows to reduce context switch overhead while handling real-time critical interrupts.

Our studies have shown that for the more loop-dominated applications found in the embedded domain, the use of register windows to reduce procedure call overhead has limited impact on performance. We did a study where each procedure used a separate window with 8-registers per window. An infinite supply of windows was assumed, thus eliminating all caller/callee save/restore overhead. This resulted in less than 2% improvement in performance. The central problem is that a majority of embedded applications spend most of their time in loop nests contained within a single procedure [36]. Thus, the overhead due to register spills dominates the save and restore code at procedure boundaries. Our approach is to make use of multiple register windows within a single procedure to reduce spill code. Eliminating spill loads and stores reduces memory accesses and thus improves performance and power consumption.

To support intra-procedural window assignment, the compiler employs a graph partitioning technique. A graph of virtual registers is created and partitioned into window groups. In the graph, each virtual register is a node and edges represent the affinity (the desire to be in the same window) between registers. Spill code is reduced by aggressively

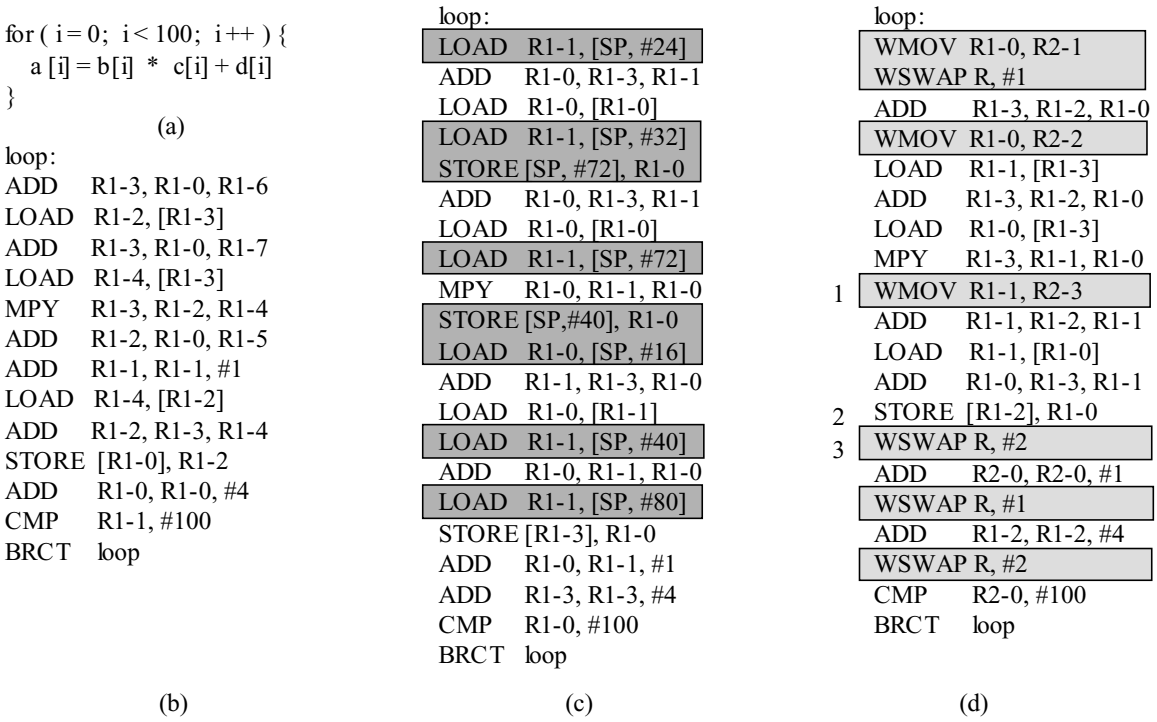


Figure 3.1: Register window example. (a) C-source; Assembly code for (b) 1-window of 8-registers (c) 1-window of 4-registers (d) 2-windows of 4-registers. Registers are denoted by window number '-' the allocated register number.

assigning virtual registers to different windows, hence exploiting the larger number of physical registers available. However, window maintenance overhead in the form of activating windows (also known as window swaps) and moving data between windows (also referred to as inter-window moves) can become excessive. Thus, the register partitioning technique attempts to select a point of balance, whereby spills are reduced by a large margin at a modest overhead of window maintenance.

3.2 Windowed Architecture Example

In order to demonstrate the benefits of register windowing for reducing spill code while incurring the overhead of the window management instructions, consider the example

shown in Figure 3.1. The original C-source is shown in Figure 3.1(a). The loop segment has been mapped to three different register window configurations: Figure 3.1(b) shows 1-window of 8-registers, Figure 3.1(c) shows 1-window of 4-registers, and Figure 3.1(d) shows 2-windows of 4-registers. For clarity, we use a generic RISC-like instruction set instead of the WIMS instruction set and assume a unified register file instead of disjoint address and data files throughout all examples in this chapter. In the assembly code in Figure 3.1, the leftmost operand is the destination. We use the notation *Ri-j*, where *i* denotes the window number and *j* denotes the register number. For the window swap operation (*WSWAP*), the first operand specifies the register file, while the second argument specifies the new active window. This allows windows within each register file to be controlled independently.

The windowed register file architecture restricts all operands within a single instruction to refer to the current active window. All operations following a *WSWAP* access their operands from the new active window. *WMOV* denotes the inter-window move instruction which can move values between any two register windows. If an operation refers to registers in different windows, one or more *WMOV* operations are required. Considering Figure 3.1(d), the *WMOV* instruction (instruction marked 1) transfers the value from register (*R2-3*) to the register (*R1-1*), which is then used in the following *ADD* instruction. The *STORE* instruction (instruction marked 2) accesses all of its operands from window 1. The *WSWAP* (instruction marked 3) toggles the active window from 1 to 2 so that the following *ADD* instruction can source all of its operands from window 2.

In Figure 3.1(b), all program variables and temporaries can fit in registers and hence

no spill is generated with 8-registers. Conversely with 4-registers, significant spill code (the load and store instructions shaded in dark gray) is generated as there are insufficient registers to hold the necessary values as shown in Figure 3.1(c). In Figure 3.1(d), by partitioning the variables and temporaries into 2-windows of 4-registers, no spill is generated although there is an overhead of 4-window-swaps and 3-inter-window moves (all shown in light gray). This configuration has the same number of total registers as that in (b) with the encoding benefits of (c). On the WIMS processor, where every instruction executes in a single cycle, Figure 3.1(c) has an 8-cycle overhead as compared to Figure 3.1(b), while Figure 3.1(d) has only 7 extra instructions. More importantly, Figure 3.1(d) has fewer loads and stores (0 spill operations) to memory as compared to (c) (8 spill operations) and thus consumes significantly less memory power.

The remainder of the chapter explains the compiler algorithm to automatically partition the variables and temporaries referenced in a procedure into the available register windows, as illustrated in Figure 3.1(d), such that the spill cost is reduced while minimizing the extra overhead due to the window swaps and inter-window moves.

3.3 Register Window Partitioning

3.3.1 Overview

The overall compilation system for register window partitioning is based on the Trimaran infrastructure [96] and is shown in Figure 3.2. Ignoring the gray boxes, the base compiler system consists of the machine independent frontend which does profiling, classical code optimizations (such as common sub-expression elimination, constant folding, induction variable elimination, etc.), loop unrolling, and procedure inlining to produce a

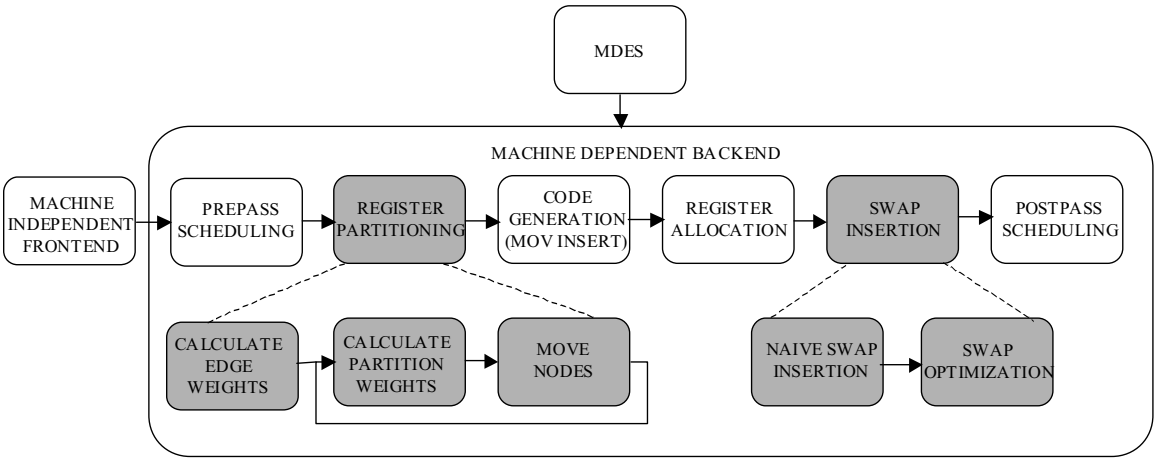


Figure 3.2: Overview of the compiler system with extensions to support register windows.

generic assembly code for a load-store architecture. The assembly code uses an infinite supply of virtual registers (VRs) to communicate values between operations. A machine description file (MDES) is used to describe the architecture of the target machine for generating machine-specific assembly code. The MDES contains a detailed description of the register files including the windows into which each file is partitioned, number of registers, connectivity of register files to function units, instruction format, and a detailed resource usage model which is used by the instruction scheduler. The connectivity model helps the compiler's code generator conform to the architectural specifications of the target machine. After prepass scheduling, all VRs are partitioned into the available register windows. For each register file, the register allocator uses a graph coloring algorithm [48] to assign physical registers to the VRs, generating spills if required. Finally, the resultant code is postpass scheduled to produce the fully bound assembly code.

The new phases added to handle register window partitioning are shown in gray boxes in Figure 3.2. Register partitioning treats each window/partition as a separate register

file and binds VRs allocated to a given partition to the corresponding register file. The partitioning algorithm could assign VRs referenced by an operation to different windows. The code generator inserts appropriate inter-window moves to honor the architectural constraints of all registers accessed by a single operation coming from the same window. The swap insertion phase inserts window swaps in the code so that two operations that access different register windows are separated by a window swap. The swap optimizer then removes the redundant swaps. Prior to postpass scheduling, an edge drawing phase inserts additional dependence edges to ensure that operations do not move across window swaps.

The register window partitioning algorithm is modeled as a graph partitioning problem where the nodes in the graph correspond to VRs[1] used in the assembly code and the edges represent the affinity between VRs. The goal is to partition the VRs into different register windows/partitions so as to minimize the overall spill, inter-window moves, and window swaps, which indirectly leads to our overall goal of performance/power improvement by reducing the number of memory accesses for data operands.

Partitioning consists of two distinct phases - weight calculation and node assignment. Each partition is assigned a weight that measures the cost of spilling the VRs assigned to that partition. The affinity between VRs is captured using edge weights, which represents the penalty incurred if two nodes connected by the edge are assigned to different partitions. The penalty can be an inter-window move, window swap, or both. If two VRs referenced within a single operation are assigned to different partitions, the code generator is forced to insert an inter-window move. Similarly, if two VRs in different operations are not assigned to the same window, a window swap is required at some point between the

[1]Nodes and VRs will be used interchangeably in the rest of the chapter.

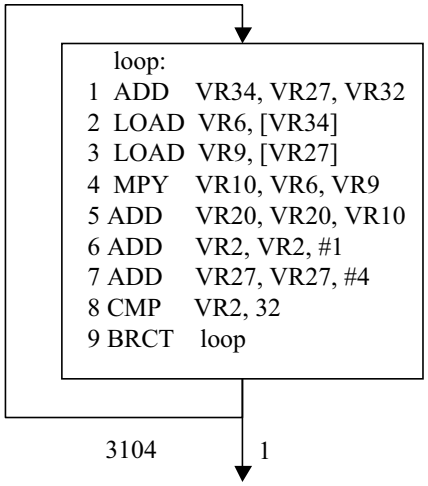


Figure 3.3: Example loop to illustrate the register window partitioning algorithm.

two operations. Unlike traditional graph partitioning that uses statically computed node weights, the partitioning algorithm uses partition weights that change dynamically during the partitioning process.

The node assignment phase uses the calculated weights to consider moving nodes between partitions so as to minimize the sum of all the partition weights and the inter-partition edge weights. The register partitioning algorithm uses a modified version of the Fiduccia-Mattheyses graph partitioning algorithm [30] which is an extension of the Kernighan-Lin algorithm [47]. The partitioning algorithm is region-based[2], i.e., all the VRs in the most frequently executed region are partitioned, followed by the VRs in the next most frequently executed region, and so on. The node assignment phase must ensure that the partitioning decisions are honored across all regions.

Figure 3.3 is a code segment from the inner-most loop of the finite impulse response (FIR) filter. The dynamic execution frequency, obtained from profiling the application on a sample input, is 3104. This example will be used throughout this section to illustrate the

[2]A region is any block of code considered as a unit for scheduling like a basic block or superblock [63].

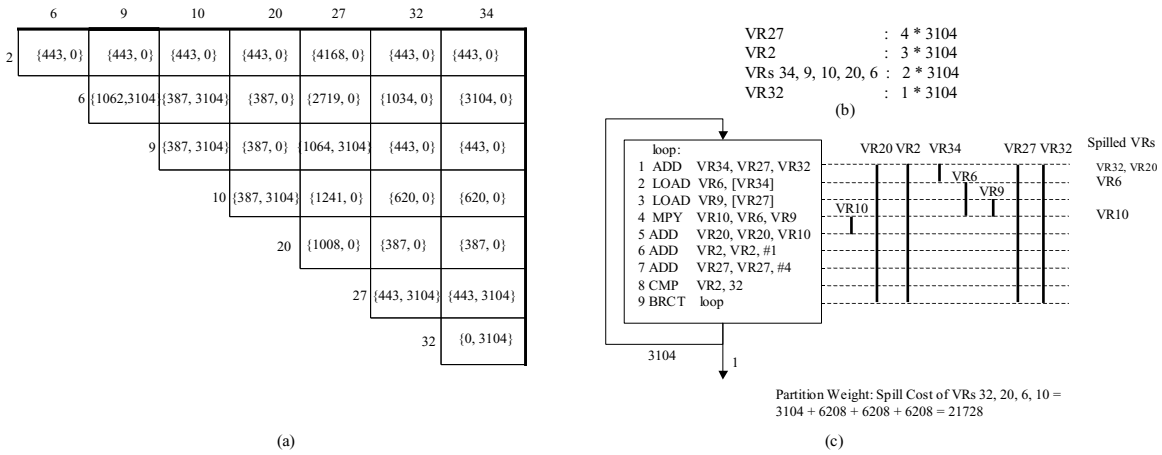


Figure 3.4: Example of partition and edge weight calculations. (a) Edge weight matrix: each cell contains {swap cost, move cost}. (b) Spill cost of VRs. (c) Partition weight computation assuming all VRs assigned to one partition.

weight calculation and node assignment process. For illustrative purposes, the goal here is to partition this region into 2-windows of 4-registers each, although the WIMS processor has 2-windows of 8-registers per window. In this work, profile information is used in the edge and partition weight calculations. Alternately, static weights based on the nesting depth of loops can also be used.

3.3.2 Edge Weight Calculation

An edge is associated with every pair of VRs. The edge weight is used by the partitioning algorithm to represent the degree of affinity between two VRs. The algorithm tries to place two nodes with high affinity in the same partition, while trying to minimize the sum of the edge weights between nodes placed in different partitions. An edge weight is an estimate of the number of dynamic moves and swaps required when two VRs are placed in different windows. By placing two VRs with high affinity in a single partition, the algorithm reduces the number of swaps and moves. The edge weight between VRs is

expressed as a matrix (see Figure 3.4(a)) computed prior to the node assignment process. The edge weight is the sum of two components: the estimated move cost and swap cost.

Move Cost: An operation may only reference registers from a single window. Thus, VRs referenced by a single operation that are assigned to different partitions require an inter-window move (*WMOV*) operation. For every pair of VRs, the total number of such dynamic instances is the estimated move cost. In Figure 3.3, VRs 6 and 9 are referenced in operation 4. If these VRs are in different windows, a *WMOV* is required for this operation. Hence, the move cost for VRs 6 and 9 is the frequency of operation 4 or 3104. Conversely, VRs 27 and 6 are not referenced together in any operation and hence do not require a move. This process is carried out for all pairs of VRs producing the matrix of values in Figure 3.4(a) (right entry in each cell).

Swap Cost: If two VRs are assigned to different windows/partitions, a window swap (*WSWAP*) is required before the operation that refers to the second VR. Swap cost estimates the number of swaps required between every pair of VRs assuming that they are assigned to different partitions. For every pair of VRs, the region is scanned in linear order. On reaching the first VR, the current active window is assumed to be 1. On encountering the second VR, the current active window becomes 2 and hence a swap is required right before the operation which references the second VR. Continuing further, on seeing an instance of the first VR again, the active window changes and another swap is required. No swap is required for consecutive references to the same VR. At the end of the region, the total number of swaps gives an estimate of the number of swaps required between this pair of VRs. The swap cost is therefore the number of swaps times the profile weight of

the region under consideration.

In Figure 3.3, between VRs 27 and 34, the swaps are computed as follows. We assume that VRs 27 and 34 are assigned to windows 1 and 2 respectively. VR27 is referenced in operations 1, 3, and 7 and so these operations are assigned to window 1. VR 34 is referenced in operation 2 and so the operation is assigned to window 2. In the sequential execution, swaps need to be inserted after operations 1 (window 2 activated) and 2 (window 1 activated). Hence, the total swap cost is $2 * 3104 = 6208$.

Adding swap cost between every pair of VRs can over-estimate the importance of swaps as the number of swaps is a function of the partition assignment of all the VRs and not just between two VRs. For example, consider operations 3 and 4 in Figure 3.3. If VRs 9 and 27 are assumed to be in window 1 and VRs 10 and 6 in window 2, the above method would count the swap four times, between 9-10, 9-6, 27-10, and 27-6, although only a single swap is necessary.

To deal with this over-counting, swap counts are used to normalize the swap cost between every pair of VRs. The swap count is the number of swaps between every pair of operations due to every pair of VRs. For example, between operations 6 and 7, 5 swaps are required. These swaps are due to VR pairs 10-27, 20-27, 2-27, 27-9, and 27-6. Generalizing this, let $c_1, c_2 ... c_k$ be the swap count due to swaps required by all pairs of VRs after operations $op_1, op_2 ... op_k$. If two VRs vr_i and vr_j, require a swap after these k operations, then the normalized *swap cost estimate* between vr_i and vr_j is $(1/c_1 + 1/c_2 + ... + 1/c_k) * cost\_of\_swap$, where $cost\_of\_swap$ is the cost of a single swap operation. Intuitively, a swap after an operation could be shared by multiple VR

pairs. Further, regardless of the number of windows, at most one swap is required between every pair of operations. Thus, if n VR pairs introduce a swap after an operation, then the contribution to the swap cost by any one of those VR pairs is $1/n$. In Figure 3.3, VRs 10 and 27 require a swap after operations 3 and 6. Since operation 3 has a swap count of 5 (due to the 5 pair of VRs including 10 and 27 listed above) and operation 6 also has a swap count of 5 (including 10 and 27), the swap cost estimate between VRs 10 and 27 is $(1/5+1/5)*3104 = 1241$ (Figure 3.4(a), left entry in each cell).

3.3.3 Partition Weight Calculation

The partition weight estimates the spill cost for the VRs assigned to each partition. The node assignment phase tries to minimize the sum of the weights of all the partitions. The partition weights are computed using a crude linear scan register allocation algorithm [73] to compute the estimated dynamic spill cost.

Given a set of VRs assigned to a partition, the live-ranges (the range of operations from all defines to all uses of the value) of the VRs are computed. For each VR, its dynamic reference count is calculated using the profile information. If the VR is spilled, then the dynamic reference count gives an estimate of the load/store overhead for spilling that VR. For every operation in the region spanned by the live-range of the VRs under consideration, the interfering VRs are considered as candidates for spill. If the number of overlapping live-ranges for that operation is more than the number of registers in that partition (size of the register window)[3], the interfering VRs are spilled until the overlapping live-ranges are less than the register window size. Note, we are only estimating the weight of the partition

[3]In our implementation, we assume the number of available register is one less than the window size. This is done to factor in the interferences due to inter-window moves that are inserted later.

by estimating spills. The actual spill code insertion is done during register allocation within each window after the window assignment process.

The VRs are chosen for spilling in increasing order of dynamic reference count. If two VRs have the same dynamic reference count value, the one with larger live-range is spilled. Once a VR is spilled, it no longer interferes with the rest of the operations and hence is not considered for subsequent operations if there is an overlap. The cost of the partition is the sum of the dynamic reference counts of the spilled VRs. If a VR is already assigned to the register window (from an earlier region) then, for the rest of the VRs interfering with this VR, the number of available registers is reduced by one.

In Figure 3.4(b), the spill cost/dynamic reference count for each VR is shown while in Figure 3.4(c), the live-ranges of the VRs are shown on the right. Assume that all VRs are assigned to a single partition and three physical registers are available per partition. At operation 1, five VRs (20, 2, 34, 27, and 32) are live simultaneously. Since there are only three registers available in the partition, VRs 32 and 20 are spilled. VR 32 has a spill cost of 3104 as there is only a single reference of that VR in operation 1, while other VRs are referenced more than once and have spill cost greater than 3104. Hence, VR 32 is picked first. VRs 20 and 34 both have a dynamic reference count of 6208, but VR 20 has a larger live-range and is chosen next for spilling. At operation 2, VRs 20, 2, 34, 6, 27, and 32 are live. Since 32 and 20 are already spilled, only VR 6 gets spilled as it has a smaller dynamic reference count than VRs 2 and 27, and larger live-range than VR 34. At operation 3, VRs 20, 2, 6, 9, 27, and 32 are live. Since 32, 20, and 6 are already spilled, no more VRs are spilled as the number of remaining live VRs is equal to three. VR 10 is

spilled at operation 4. For the rest of the operations, no additional VRs are spilled. So, for this partition, the partition weight is the spill cost of the spilled VRs 32, 20, 6, 10, which is $3104 + 6208 + 6208 + 6208 = 21728$. In actual implementation, instead of considering every operation, only operations which are at the start/end points of any live-range are considered. So in Figure 3.4(c), only operations 1, 2, 3, 4, 5, and 9 are considered. For the other operations, the live-range information does not change and hence are ignored.

Since region-based partitioning is performed, window assignments of a higher priority region can affect the decisions in a lower priority region. While computing the partition weights, it is possible that there are live VRs that are already assigned to partitions from processing higher priority regions. If these VRs were not spilled, then they are assumed to be pre-bound to a register. Thus, the window has one fewer register available per such pre-bound register.

3.3.4 Node Partitioning

The goal of the node partitioning phase is to reduce the overall spill cost while minimizing the impact due to inter-window moves and swaps. Starting from an initial partition, the node partitioning algorithm tries to iteratively distribute the VRs into different partitions so as to reduce the sum of the weights of all partitions, while trying to minimize the edge weights between nodes assigned to different partitions. The node partitioning technique that we used is a modified version of Fiduccia-Mattheyses's [30] graph partitioning algorithm (FM). It consists of two phases - *initial partitioning* and *iterative refinement*.

Initial Partitioning: Placing all VRs in the first partition can create an unbalanced initial configuration which can affect the quality of the partitioning algorithm. The initial

Algorithm 3.1: Initial partitioning of node (VRs).

```
1  InitPartition() ;
   Input: List of VRs within a procedure and the number of windows/partitions
2  sortedList = Sort all VRs in decreasing order of dynamic reference count ;
3  foreach (partition p) do
4      foreach (vr ∈ sortedList) do
5          LR = list of ops in which vr is live ;
6          spill = false ;
7          foreach (op ∈ LR) do
8              allocatedVrs = list of vrs at op that are live and allocated to partition p ;
9              numAvailRegs = number of registers in window/partition p - number of
               allocatedVrs ;
10             if (numAvailRegs ≤ 0) then
11                 spill = true ;
12                 break ;
13             end
14         end
15     end
16     if (!spill) then
17         add vr to partition p ;
18         mark vr as allocated to partition p ;
19         remove vr from sortedList ;
20     end
21 end
22 if (!sortedList.isEmpty()) then
23     add all VRs in sortedList to first partition p ;
24 end
```

partitioning algorithm tries to distribute the VRs into partitions so as to start with an initial configuration of relatively less register pressure while being incognizant of the swap and move overhead. If a given window/partition has sufficient registers to accommodate the VRs, then all of the VRs are allocated to that partition. If not, the VRs are assigned based on a priority order to a particular window/partition until no more VRs can be assigned to it without the need for spilling. The remaining VRs are then placed in the next partition until it gets saturated and so on.

The algorithm for the initial distribution of VRs to windows/partitions is given in Al-

gorithm 3.1. Initially, all VRs are sorted based on the dynamic reference count (step 1) so that the most important VRs are assigned first. For a given partition, allocation is attempted for every VR in the sorted list. The allocation is done using a simple linear scan register allocation algorithm similar to the technique described in Section 3.3.3. It should be again noted that this heuristic generates an initial partition of VRs to register windows without actually register allocating them. For every VR, its live-range (*LR*) is computed (step 4) as a list of operations. For every operation *op* in *LR* (step 6), *num_avail_regs*, which is the difference between the total number of registers in the current partition and number of allocated registers that are live at *op*, is computed (steps 7 and 8). If there are free registers (step 9), then this VR is assigned to the current partition (steps 16, 17, and 18), else it is assumed to be spilled. This process continues for all VRs in the sorted list such that they are either assigned to the current partition or spilled. The spilled VRs are then attempted for assignment to the next partition (step 2) using the same algorithm. Once all of the partitions are processed, any remaining spilled VRs that could not be assigned to any partition are simply assigned to the first partition (step 22).

Iterative Refinement: After the initial assignment, the iterative refinement phase tries to move VRs across partitions to reduce the overall spill cost (partition weights) while minimizing the overhead due to swaps and moves (edge weights between partitions). The algorithm for the node partitioning is given in Algorithm 3.2. Initially, a set of *n*-partitions (where *n* is the number of register windows) is created (*pset*, step 1) and initialized using the initial partitioning algorithm described in Algorithm 3.1 (step 2). This partition configuration is used as the seed configuration for the first pass. In step 3, the overall weight

Algorithm 3.2: Node (VR) partitioning.

```
   Input: List of VRs within a procedure
   Input: n: number of windows/partitions
 1 pset = create n partitions ;
 2 InitPartition() ;
 3 minOverallWt = compute overall weight of pset ;
 4 save current partition configuration ;
 5 savedInfo = false ;
 6 while (minOverallWt > 0) do
 7 |   srcPart = find an unbalanced partition from pset ;
 8 |   if (srcPart = NULL) then
 9 |   |   if (savedInfo) then
10 |   |   |   restore part configuration ;
11 |   |   |   savedInfo = false ;
12 |   |   end
13 |   |   else
14 |   |   |   break /* terminate partitioning */ ;
15 |   |   end
16 |   |   unlock all locked vrs in pset ;
17 |   end
18 |   vr,destPart = FindBestVr(srcPart) ;
19 |   if (destPart = 0) then
20 |   |   remove srcPart from pset ;
21 |   |   continue ;
22 |   end
23 |   move vr to destPart ;
24 |   lock vr in destPart ;
25 |   overallWt = compute overall partition weight of pset ;
26 |   if (overallWt < minOverallWt) then
27 |   |   minOverallWt = overallWt ;
28 |   |   save current partition configuration ;
29 |   |   savedInfo = true ;
30 |   end
31 end
32 restore partition configuration ;
```

of the set of partitions is computed. The overall weight is the sum of the partition weights and the weights of the cut edges across all partitions. The initial partition configuration is then saved in step 4 assuming that it is the best seen yet. During a single iteration of the loop in step 6, the partition (*srcPart*) with the maximum weight is selected (step 7). If such a partition exists (step 17), the node with the largest gain (*FindBestVr*) is selected.

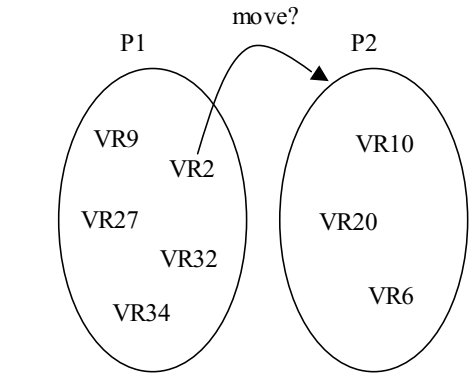
Algorithm 3.3 gives the algorithm for *FindBestVr*. For every node in the source partition, *FindBestVr* computes the gain in moving the node to all other destination partitions.

Algorithm 3.3: Find the best VR to be moved to a different partition.

```
FindBestVr(srcPart) ;
Input: Partition srcPart from which the best VR to be moved is selected
return bestnode: Best VR that is to be moved from srcPart, destPart: destination partition
to which the VR has to be moved  foreach (node in src partition) do
    foreach (destPart ∈ pset) do
        move node to destPart ;
        oldTotalWt = srcpWtOld + destpWtOld ;
        newTotalWt = srcpWtNew + destpWtNew ;
        partitionWtGain = oldTotalWt − newTotalWt ;
        edgeWtGain = oldEdgeWt − newEdgeWt ;
        gain = partitionWtGain + edgeWtGain ;
        if (gain > maxgain) then
            bestnode = node ;
            maxgain = gain ;
        end
    end
end
```

Gain is defined as the sum of the *partitionWtGain* and *edgeWtGain*, where *partitionWtGain* is the reduction in total partition weights when the node is moved from the source to the destination partition. Similarly, *edgeWtGain* is the reduction in edge weights between nodes in the source and destination partitions. In Algorithm 3.3, *srcpWtOld*/*destpWtOld* is the weight of the source/destination partition before the node is moved, while *srcpWtNew*/*destpWtNew* is the weight of the source/destination partition after the node is moved. The node (*bestnode*) with the highest gain and the destination partition (*destPart*) to which it is to be moved are returned.

The partitioning algorithm then picks the node with the highest gain (*vr*) and performs the move to the destination partition (*destPart*, step 22). It should be noted that the highest gain could be a negative value. Allowing negative gains helps avoid local minima. Once a node is moved over to the new partition, it is locked in the new partition and not considered in the current pass (step 23). The overall partition weight is then recomputed in step 24.



(a)

VR	Partition Gain	Edge Gain	Total Gain
2	-3104	-4168	-7272
9	-6208	2547	-3661
34	0	-3426	-3426
27	-3104	-10462	-13566
32	-3104	-5496	-8600

(b)

Figure 3.5: Partitioning applied to example: (a) Initial partition, (b) Gains for each VR moving from P1 to P2 after the initial partition.

If the overall weight is less than the minimum overall weight, it implies that the resultant partition configuration is the best configuration seen so far and hence the configuration is saved. If *FindBestVr* has no more VRs to move (either because all VRs have been locked or there are no destination partitions to move to), the *srcPart* is removed from the set of partitions (steps 18 and 19). In step 8, if there are no more partitions left, the current pass is ended. If during the previous pass, a better configuration was seen and saved (*savedInfo* flag is set to true), then the best configuration seen so far is restored and used as the seed for the next pass (step 9). Before commencing the next pass, all VRs are unlocked (step 15). If *savedInfo* is set to false, it implies that during the previous pass a better configuration was not seen and the partitioning is terminated. The partitioning is also terminated if the *minOverallWt* reaches 0 (step 6). On exit, the best partition seen over all passes is restored as the final partition (step 31).

Example: The initial partition configuration is shown in Figure 3.5(a). To compute

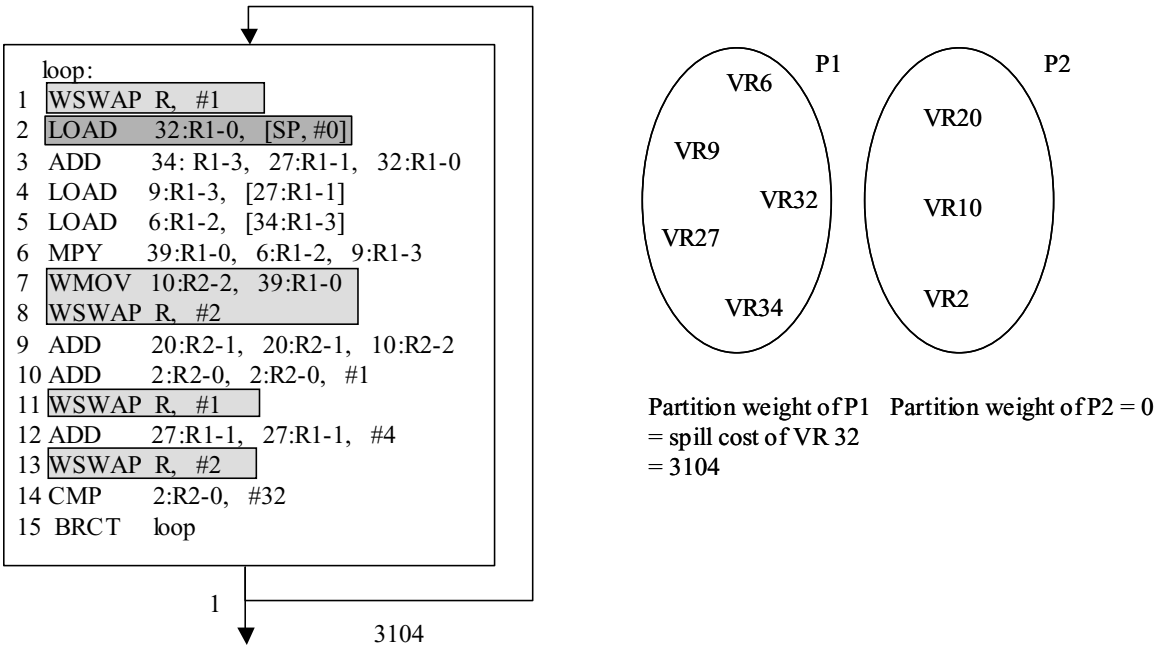


Figure 3.6: Example after window assignment. The notation $VR : R_i - j$ is used, where VR is the original virtual register number from Figure 3.3, i is the window number, and j is the allocated register number.

the initial partitions, the live-ranges and the dynamic reference counts of the VRs are shown in Figures 3.4(b) and (c), respectively. The VRs are considered in the decreasing order of dynamic reference count. We assume 3-registers per window. Initially, the top two VRs, 27 and 2, are allocated to partition 1. Next, VRs 34 and 9 are allocated as they do not interfere with each other and the number of maximum interfering live-ranges is three. Since partition 1 is now saturated, the rest of the VRs (except VR 32) are assigned to partition 2. VR 32, which has the least count, is assigned to partition 1 and is spilled. The initial partitioning algorithm thus distributes the VRs such that the total partition weight is minimized (3104).

The iterative refinement phase then tries to move each VR from P1, which is the highest weight partition, to P2 and computes the resultant partition and edge gains. Figure 3.5(b)

shows the partition, edge, and total gain for moving each VR. The VR with the maximum total gain is chosen. Here, all gains are negative, so VR 34 with the smallest negative gain is moved to partition 2. This VR is then locked in partition 2. Subsequently, VR 6 (with gain 3246) is moved to partition 1 and VR 2 (with gain 8682) is moved to partition 2.

Edge weights are computed statically before partitioning. So, *find_best_vr* need only do a lookup of the edge weight matrix (Figure 3.4(a)) to get the edge weights between a pair of VRs. But, this is not the case with the partition weights. As nodes migrate from partition to partition, the interferences among VRs can change and so the partition weight (spill cost) has to be recomputed (Section 3.3.3) on the fly. Each move of a node would thus require an $O(n)$ scan of the operations in the region and hence the complexity of the partitioning algorithm is $O(n^2)$ per round of refinement. Figure 3.6 shows the final partition configuration. The total partition weight is 3104 which is same as the initial partition weight. The initial partitioning algorithm minimized the number of spills but did not consider the impact of swaps and moves. The iterative refinement converged at a configuration such that both spill cost (partition weights) and swap and move cost (edge weights) are minimized. The final code after register allocation and swap insertion is also shown in Figure 3.6. The code has 1 spill (operation 2) 4 swaps (operations 1, 8, 11, and 13) and 1 inter-window move (operation 7).

3.3.5 Partitioning Algorithm Optimizations

In order to speed up the partitioning process, two optimizations were implemented over the core algorithm described above.

Fast Spill Pressure Estimation: A cycle-by-cycle linear scan of the operations in a region to determine what VRs are spilled is inherently slow as this has to be done every time a VR is moved from a source to a destination partition. This dynamic computation of partition weight was required because the set of VRs that are spilled is a function of the interfering live-ranges of the current assignment of VRs to that partition. Although accurate, since this is done within the core of the FM partitioning algorithm, it slowed down the partitioning process. To optimize this process, instead of scanning every operation, one could only scan the operation with the maximum number of interfering live-ranges to approximate what VRs are spilled. The process is still dynamic, but less accurate than the linear scan approach.

Restricting the Number of VRs: Although the partitioning was performed a region at a time, some of the larger benchmarks had regions with a large number of VRs that slowed down the partitioning algorithm. This large number of VRs was generated mainly because the original application source had core kernels that were written in a unrolled manner. In order to restrict the number of VRs, a compile-time fixed subset of VRs is partitioned at a time. The VRs in a region are sorted in decreasing order of dynamic reference count. By sorting the VRs, the algorithm can consider the most important subset of VRs for partitioning, followed by the next most important subset, and so on. This is done until all VRs in that region are exhausted. Partitioning decisions for a given subset are honored while partitioning the next subset (similar to the pre-bounds described at the end of Section 3.3.3).

3.3.6 Window Swap Insertion and Optimization

Window swap operations are inserted after window assignment and register allocation (see Figure 3.2). Initially, a naïve window assignment is performed by walking the region in sequential program order. A window swap operation is inserted at the beginning of the region to set the active window appropriately for the first operation in the block. Scanning each operation, if the assigned window is different from the current register window, a swap to the new window is inserted. Following every procedure call, a window swap operation is used to set the current active window to the window of the operation following the procedure call. This is necessary, as we assume separate compilation, and the state of the active window is unknown after a procedure return.

Swap optimizations: This naïve method inserts many unnecessary swap operations. Three swap optimizations were implemented to reduce the swap overhead.

- A swap at the beginning of a region is unnecessary if all control paths leading to that block have trailing operations which are in the same window as the first operation in the region.

- It is also possible to hoist a window swap upwards from the beginning of a more frequently occurring region to the end of less frequently occurring predecessors and thus reduce the total number of dynamic swaps. This is legal provided that the new window swap instruction inserted at the end of the predecessor is the last instruction of that predecessor (this might not be the case for superblocks which have multiple exits).

- To prevent redundant swaps after procedure calls, the return from subroutine operation forces the window to be set to 1. So, if an operation following the procedure call is assigned to window 1, a swap is not needed. A simple inspection of the control flow graph is used to remove such redundant swaps.

Instruction combining: To further reduce the swap overhead, experimental studies were conducted on which operations frequently preceded a swap to identify opportunities for merging swap with regular operations. Combining must be constrained by the availability of free opcode encoding bits within the instruction encoding. Inter-window register move and window-swap was found to be a likely candidate as the WIMS move operations have free encoding bits. In Figure 3.6, swap operation 8 can be combined with the inter-window move operation 7. Another interesting complex operation can be formed by combining a conditional branch with a swap such that the swap would be executed only if the branch is either taken or fallthrough. But in our experiments the frequency of their occurrence was found to be less than 2%. By combining the move with the swap we observed an average of 2% improvement in performance.

Edge drawing: After the swaps are inserted, control dependence edges are inserted between the swap and all operations preceding and following it. This is done so that the postpass scheduler does not intermix operations from different windows. All operations preceding the swap except procedure calls and branches have a 0-cycle control dependency with the swap[4]. All operations following the swap have a 1-cycle dependency with the swap and so would be executed strictly after the swap. The postpass scheduler is then

[4]branches and procedure calls have a 1-cycle control dependency edge to the following swap as the swap has to take into effect after the branch or function call.

invoked to schedule the swaps, inter-window moves, and the spill code while honoring the new control dependences.

3.4 Experimental Results

3.4.1 Methodology

We implemented the register partitioning algorithm using the Trimaran infrastructure, a retargetable compiler for VLIW processors [96]. For our study, only the integer register file was assumed to be windowed and so a set of integer-dominated benchmarks from a mix of Mediabench [54] and MiBench [36] suites were evaluated. All of the benchmarks were compiled with control-flow profiling, superblock formation, function inlining, and loop unrolling. For the experiments, the number of windows and the number of registers per window were varied to evaluate the energy and performance impact. Two machine configurations were used - the WIMS processor and a 5-wide VLIW machine with the following function units: 2 integer, 1 floating-point, 1 memory, and 1 branch. The VLIW-machine uses the HPL-PD [46] ISA with latencies similar to an Itanium machine with perfect caches and support for compile-time speculation and predication. The VLIW-style architecture was chosen due to its increasing popularity in the embedded domain [93]. For the VLIW machine, the swap instruction is assumed to be compatible with any slot in the VLIW word, and thus can be assigned to any free slot. In our experiments, the floating-point unit is often free, thus the swap occupies that instruction slot. Inter-window moves execute on the integer unit.

We considered the energy/performance improvement of a range of register file configurations consisting of 1, 2, 4, and 8 identical windows containing 4 and 8 registers per

window. The following specific configurations were evaluated: 2-window, 4-window, and 8-window with 4-registers per window (w2.r4, w4.r4, w8.r4) were compared against a base 1-window of 4-registers (w1.r4); and 2-window and 4-window with 8-registers per window (w2.r8, w4.r8) were compared against a base 1-window of 8-registers (w1.r8). This helped illuminate the energy/performance benefits of increasing the effective number of registers without changing the instruction set architecture. We also fixed the total number of registers while partitioning them into 2 and 4 equally sized register windows. In particular, the performance of window configurations w2.r8 and w4.r4 were compared against the base w1.r16. This helped clarify the performance degradation suffered by a windowed architecture compared to a non-windowed architecture having the same number of architected registers. The performance numbers were obtained by multiplying the schedule length of each region by its execution frequency to get the total dynamic execution cycles for the whole program. Since we use a single cycle memory system for the WIMS and the VLIW processors, this approach is quite accurate.

3.4.2 Results

Increasing the number of available registers with a fixed window size: The graph in Figure 3.7 (top) compares the percent performance improvement in total execution cycles of the w2.r8 and w4.r8 configuration against the base w1.r8 configuration for the WIMS processor. Averages of 11% and 12% improvement in performance are observed for the 2-window and 4-window designs, respectively. It should be noted that the performance improvement is the result of increasing the effective number of registers while retaining a 3-bit operand encoding. The performance ranges from a maximum of 28% for

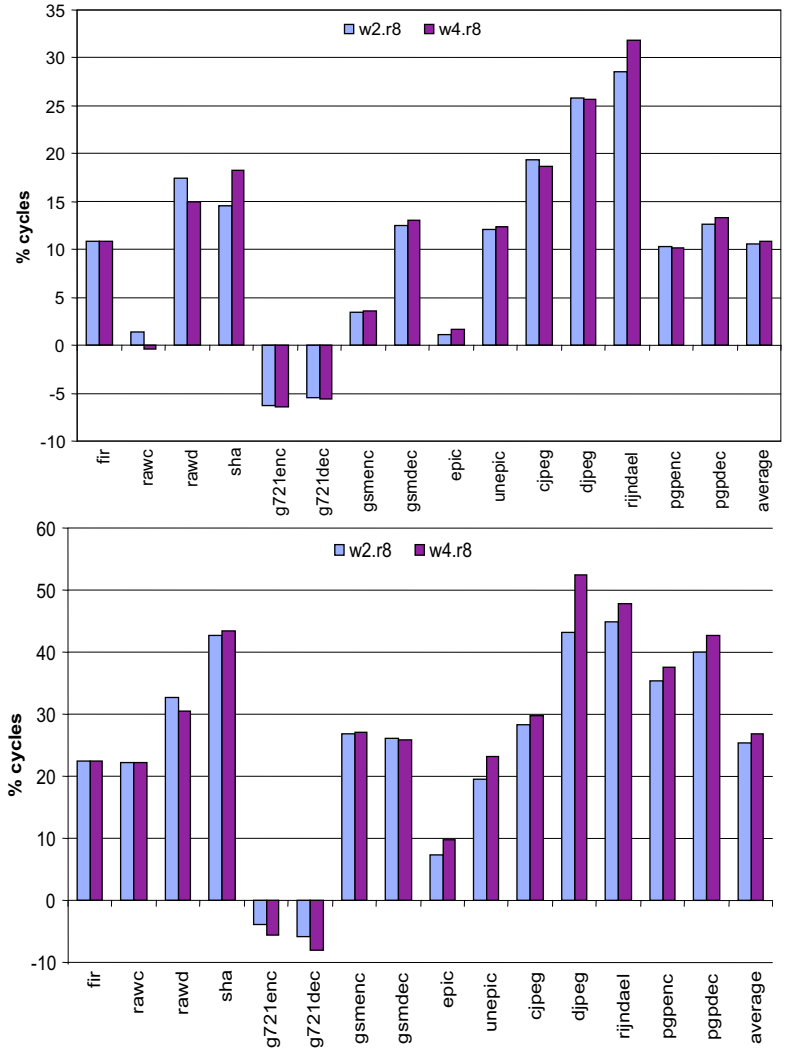


Figure 3.7: Performance improvement shown as percent reduction in cycles for the 8-register WIMS processor (top) and VLIW processor (bottom) for 2 and 4 window designs. For each benchmark, the results for w2.r8 and w4.r8 are plotted relative to w1.r8.

djpeg to a loss of 6% for g721enc. We observe only a marginal improvement in performance for a 4-window design. The additional two windows enable significant reduction in spill code, but the resulting advantage is offset by an almost equal increase in inter-window swaps and moves. The partitioning algorithm is able to identify this overhead using the edge weights, and hence prevents excessive partitioning. But in benchmarks such as rawd, the heuristic breaks, and a net loss in performance is suffered in scaling

from 2 to 4 windows.

Figure 3.7 (bottom) compares the performance improvement for the w2.r8 and w4.r8 configurations over the base w1.r8 configuration for the VLIW-machine. We observe an average of 25% and 28% improvement in performance for 2-window and 4-window designs, respectively. This gain is more than double the gain observed for the WIMS processor. The larger gains are due to several reasons related to the multi-issue capabilities. First, the spill code often sequentializes program execution by increasing the lengths of critical dependence chains through the code. For the VLIW machine, these critical dependence chains often determine the program execution time. Thus, the elimination of spills translates into more compact schedules and larger performance gains than for a single-issue WIMS processor. Second, there is a larger demand for registers to maintain the necessary intermediate values to support the inherent instruction level parallelism. Thus, the affects of eliminating spill code are more pronounced. Third, the overhead of swaps and moves is lower as they can execute in parallel with other instructions. In particular, the swap often executes in the floating-point slot making it almost ”free” for the integer dominated applications that are evaluated.

An analysis of the performance for the w2.r8 configuration is presented in Figure 3.8 for the WIMS processor (top). The percent performance improvement in total execution cycles, which is identical to the plot shown in Figure 3.7, is shown in the leftmost bar of each set. The rightmost bar shows the percent savings in dynamic spill code. The middle pair of bars show two components - (i) spill benefit, which is the percent savings in total execution cycle count due to savings in spill (second bar), and (ii) percent swap and move

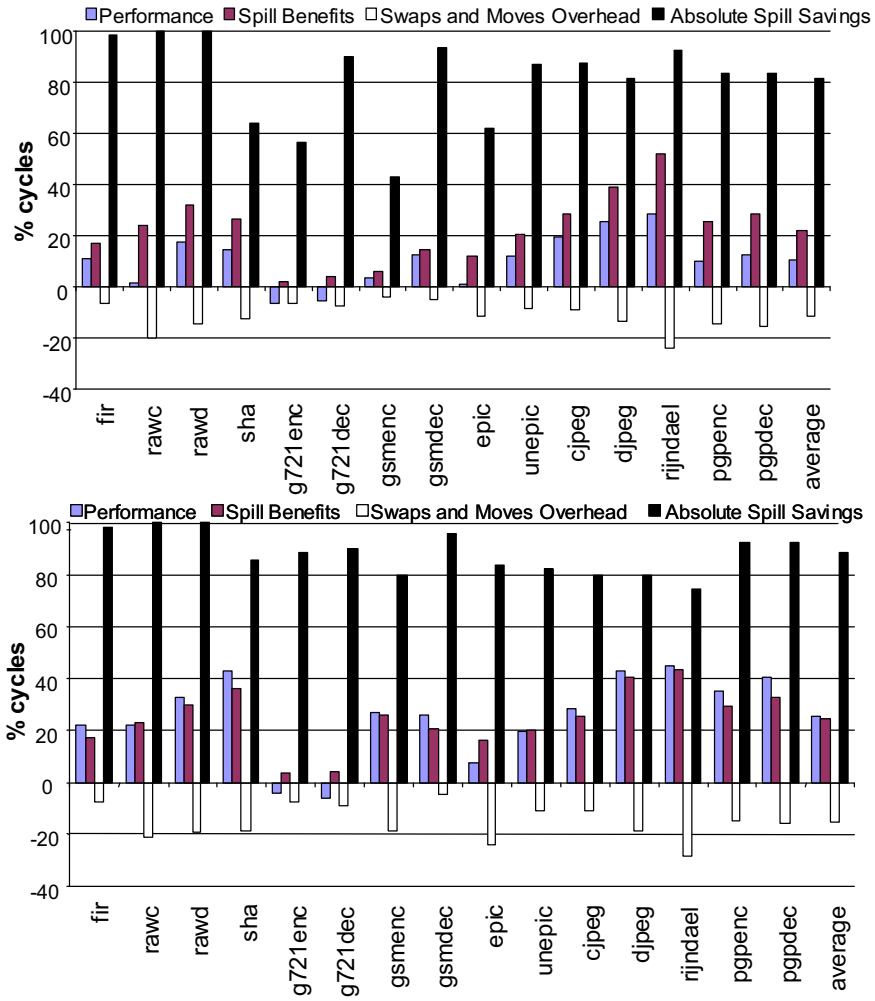


Figure 3.8: Performance analysis for the 8-register WIMS processor (top) and VLIW processor (bottom). For each benchmark, w2.r8 is plotted relative to w1.r8.

overhead, which is the percent of overall execution cycles due to the extra inter-window moves and window swaps thereby reducing performance (third bar).

In 11 of the 15 benchmarks, we observe more than 80% reduction in spill code. Performance improvement is obtained when the spill benefit exceeds the swap and move overhead. This occurs in 13 of the 15 benchmarks (except g721enc and g721dec). The graph illustrates the competing effects of spill code reduction and swap/move overhead.

For example, in gsmdec, a 92% reduction in spill code is seen, which accounts for 12% savings in total cycles. While for g721dec, there is a 90% reduction in spill code, but this contributes to only 4% savings in total cycles. This implies that the impact of spill is small for g721enc in the w1.r8 case. Since all instructions take a single cycle, any gain in performance due to savings in spill is offset by a corresponding reduction in performance due to swaps and moves. The greedy nature of the partitioning algorithm causes VRs to be aggressively separated into different partitions, thus increasing the swap/move cost.

The graph in Figure 3.8 (bottom) illustrates the same performance analysis for the w2.r8 VLIW machine. The spills, swaps, and moves shown in this figure are measured in percent dynamic operations and not dynamic cycles. Observe that for reasons described earlier, the impact due to savings in spill are greater than in the WIMS processor. Also, the overhead of swaps and moves is lower as they can execute in parallel with other instructions. Hence, performance improvement is often larger than the difference between spill benefit and swap/move overhead.

Figure 3.9 compares the performance improvement of 4-register per window configurations on the WIMS (top) and VLIW (bottom) machines. As compared to the 8-register configurations, the w4.r4 per window case shows much improved performance as compared to the w1.r4 case as four registers are insufficient for both the WIMS and the VLIW machines. Djpeg, due to loop unrolling, had a high register pressure and hence benefited significantly when the number of windows was increased for all window file sizes. As with the 8-register case, the swap and move overhead outweighs the spill savings, and hence a decrease in performance is observed in g721enc and g721dec for the WIMS processor.

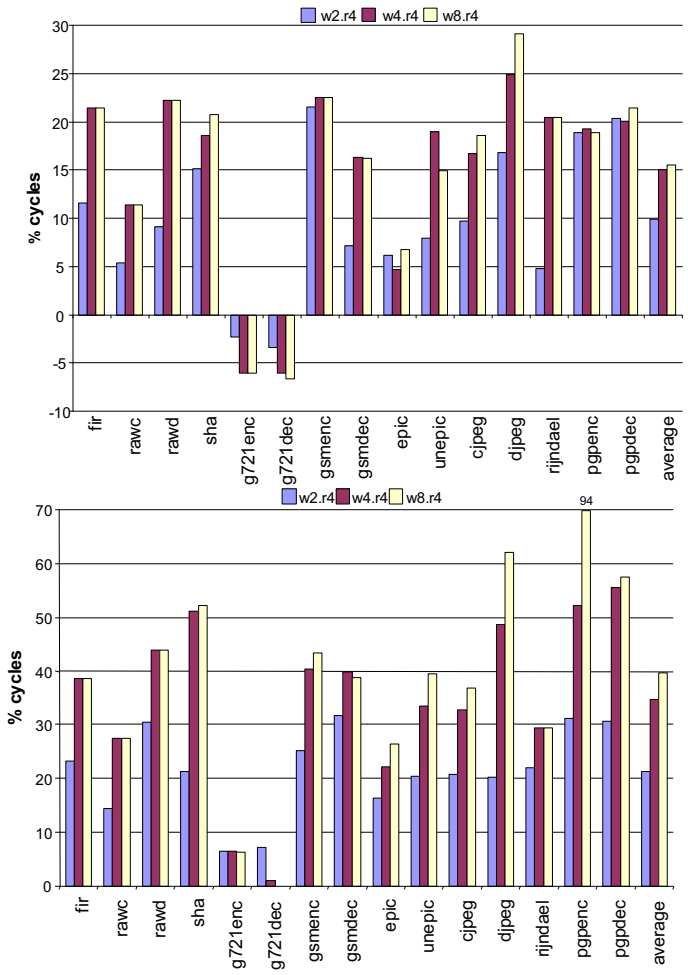


Figure 3.9: Performance improvement shown as percent reduction in cycles in increasing the number of register windows in a 4-register WIMS processor (top) and VLIW processor (bottom). For each benchmark, three sets of data are shown: w2.r4 (left), w4.r4 (middle), and w8.r4 (right), plotted relative to w1.r4.

Increasing the number of windows with a fixed number of total registers: Figure 3.10 (top) shows the percent slowdown in dynamic execution cycles due to partitioning a 16-entry register file into 2 and 4 windows on the WIMS processor. The base w1.r16 has no swap and move overhead. This provides an unachievable upper bound of the partitioning heuristic and helps to gauge how well our heuristics perform against an idealistic case. For each of the configurations, the total number of registers is the same but they have been partitioned into multiple equal sized windows. It should be noted that as we partition

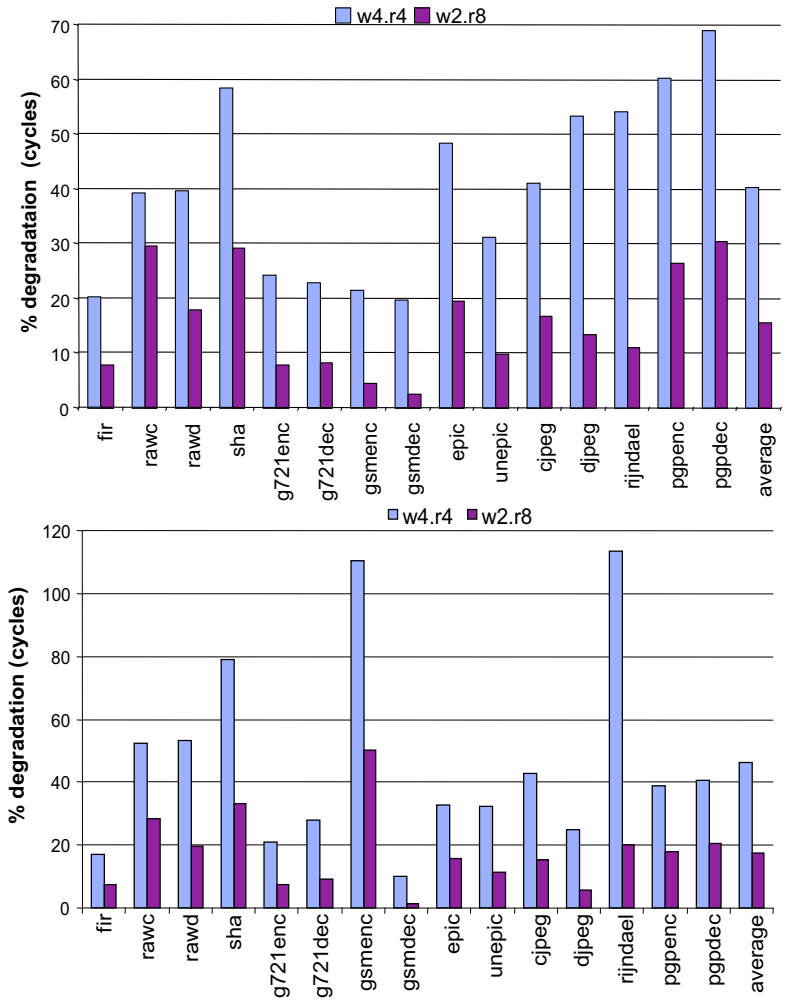


Figure 3.10: Performance degradation while partitioning a 16-entry register file into 2 and 4 windows for WIMS (top) and VLIW (bottom). For each benchmark, w2.r8 and w4.r4 are plotted relative to w1.r16.

the register file, the instruction encoding size decreases, as fewer bits are required in the register field specifier within the instruction format, but we have not accounted for this in the data. For the 16-register case, w2.r8 achieves an average of only 16% degradation in performance while the w4.r4 suffers an average of 40% degradation in performance. As we partition the register file, swaps and moves are required to distribute the VRs into all of the windows to reduce the spill pressure. Also, VRs referenced within the same instruction have to be assigned to the same partition as a single instruction must source all

of its operands from the current active window. This prevents a perfect partitioning and increases the spill pressure on a given register window. Although having a large number of registers is preferred, encoding restrictions limit the actual size. A windowed design can give the appearance of a larger register file with moderate overhead.

Figure 3.10 (bottom) illustrates a similar experiment on the VLIW processor. We observe slightly worse results compared to the WIMS processor, as partitioning can increase the register pressure on a single window, thus increasing spill code, which in turn has a larger impact on the VLIW machine.

Energy Benefits: The impact of register windows on the total system energy is now examined. By reducing spill code, the burden on the memory system to provide the operands is reduced, thereby increasing energy efficiency. Table 3.1 illustrates the energy breakdown and execution times for different instruction types for the WIMS processor. The energy measurements were obtained from Synopsys' Nanosim using post-APR (Automatic Place-and-Route) back-annotated parasitics. Input vectors were created at 1.8V and 100MHz operation by running assembled test cases through the pipeline and capturing the switching activity [94]. The energy due to different register file sizes was negligible ($< 5\%$) when compared to the pipeline and memory energy as the number of registers considered was no more than 32.

The graph in Figure 3.11 shows the improvement in total dynamic energy as the number of 8-register windows is increased on the WIMS processor. The total energy includes the pipeline and memory energy (instruction fetch, loads and stores). Unlike performance, since spills dissipate more energy to access memory in comparison to swaps/moves, spill

Instr. class	Energy (nJ)	Time (ns)
add-sub	0.55	10
bool	0.38	10
cmp	0.52	10
div	2.27	180
mul	2.22	180
shift	0.35	10
jmp-abs	0.90	30
jmp-rel	0.64	20
br-taken	1.00	30
br-nottaken	0.39	10
win-swap	0.37	10
iw-mov	0.47	10
ld-abs	0.98	20
ld-rel	0.74	10
st-abs	.93	20
st-rel	.74	10

Table 3.1: Per instruction class energy and execution time for the WIMS processor at 100MHz.

reduction can result in significant improvement in energy consumed. For example, rijndael achieves a 52% reduction in energy in the w4.r8 configuration. Here, energy reduction is obtained by exchanging spill for a swap/move. Unlike performance, where a significant reduction in spill is required to offset the overhead due to moves and swaps, equal exchange is good for energy as the number of memory accesses are reduced. It should be noted that like performance, the energy reduction is observed as we increase the effective number of available registers while restricting the ability to address only 8 registers within an instruction. Although our heuristics were geared towards improving performance as opposed to energy, it can be easily retargeted to optimize for energy by weighting the savings in spill code more than the savings in swaps and moves.

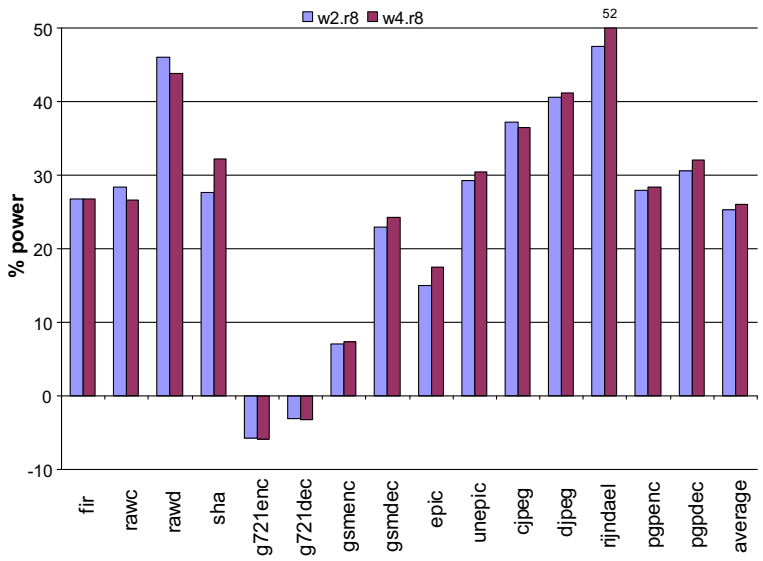


Figure 3.11: Percent dynamic energy improvement of w2.r8 and w4.r8 over the base case of w1.r8.

3.4.3 Comparison among different partitioning heuristics of varying estimation accuracy

Optimal partitioning of VRs into partitions to minimize spills, swaps, and moves is an NP-hard problem. If there are n VRs and m partitions, an exponential number ($O(m^n)$) of possible assignments needs to be evaluated, which is clearly impractical. Compiler heuristics for register partitioning provides a trade-off between quality of solution and runtime. In this section, we explore this trade-off by comparing against two other heuristics - *global* and *fast*.

A region-based heuristic (called *region* and described in Section 3.3), where edge weights were computed statically prior to the partitioning process and partition weights were computed on the fly, was used in all previous experiments. This process, although fast, is inaccurate, as the actual swap and move cost is a function of the current assignment of VRs to partitions. Accurate estimates of the swap and move penalties are possible if the number of swaps and moves are recomputed dynamically during partitioning by scan-

ning the operations in the procedure. This method, though accurate, is inherently slow. In addition, region-based partitioning can result in suboptimal decisions if there are multiple regions with comparable profile weights.

To evaluate a more accurate heuristic, a semi-brute force method was implemented wherein the basic FM-based partitioning methodology was retained, but two changes were made: the edge weights were computed during partitioning and the scope was extended to the whole procedure rather than a region. In order to reduce the computational complexity of considering too many VRs, a fixed number (set at compile time) of the most frequently occuring VRs is considered at a time. We implemented this slower semi-brute force-like method (called *global*) to compare against our preferred *region* method to quantify the loss in partition quality. The partition weights for *global* were computed just as in *region*.

To evaluate another faster (compared to the *region* method), but less accurate heuristic an FM-based *fast* heuristic was implemented with static estimation of edge weights while operating within a region scope. The spill estimation was performed by considering only the operation with the maximum intersecting live-ranges (see Section 3.3.5). Both *fast* and *global* methods considered only 64 VRs at a time during partitioning. The *fast*, *region*, and *global* methods represent heuristics that are progressively more complex, while attempting to more accurately estimate the swap, move, and spill costs.

The graph in Figure 3.12 (top) compares the percent performance improvement in total execution cycles of the 3 heuristics for the w2.r8 configuration against the base w1.r8 for the WIMS processor. Overall, an average performance improvement of 8%, 11%, and 13% was obtained for the *fast*, *region*, and *global* methods, respectively. The *region*

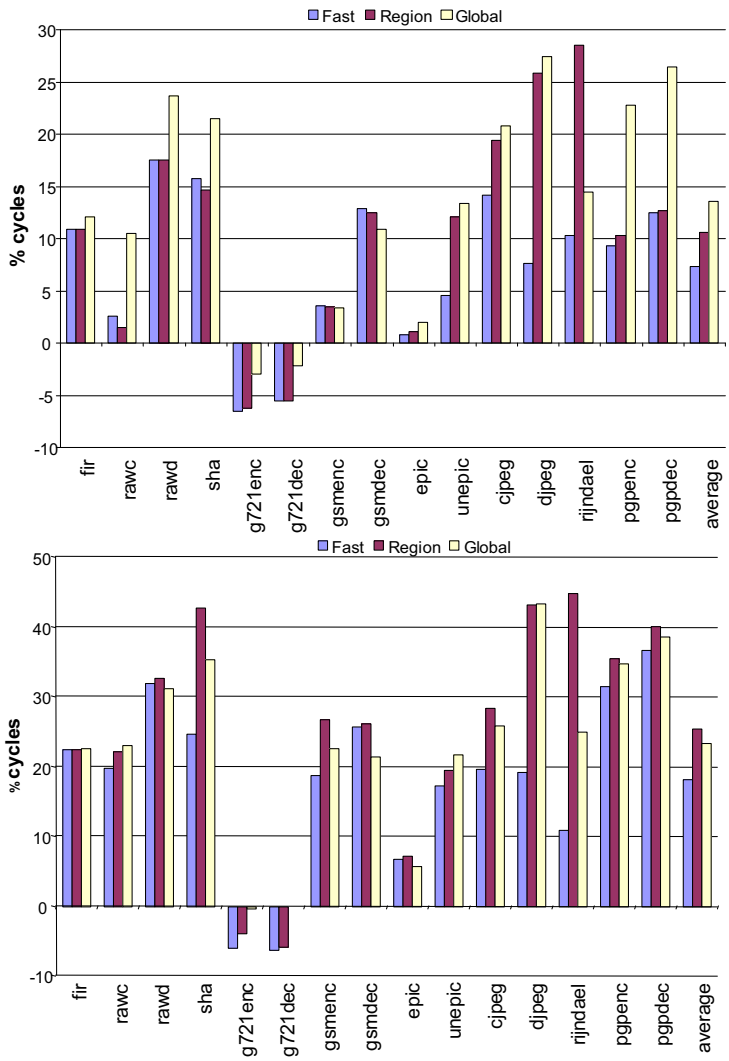


Figure 3.12: Comparing performance between *fast*, *region*, and *global* heuristics for the 8-register WIMS processor (top) and the VLIW processor (bottom). For each heuristic, w2.r8 is plotted relative to w1.r8.

heuristic did considerably better than the *fast* method because it considered a larger set of VRs (whole region) and estimated spills more accurately. The *global* method had a procedure scope that used a more accurate dynamic estimation of swaps and moves, thereby performing better than the *region* method.

However, some of the results are not monotonically increasing as one would expect for more accurate methods. In some cases (gsmdec, rijndael), the *region* method performed

better than the *global* method. These benchmarks had a single region that dominated execution time. The *region* method was more effective because it considered all VRs in the dominant region, whereas the global method could only examine 64 at a time. Also, since the underlying partitioning algorithm was greedy, an inaccurate estimation of swap, move, or spill cost sometimes results in the *fast* or the *region* heuristics doing better than the *global* method, or the *fast* method doing better than the *region* method. While running our experiments, we observed that the compile time for the *global* method was 75% more than the *fast* method, while the *region* method was 40% less then the *global* method.

The graph in Figure 3.12 (bottom) repeats the previous experiment for the VLIW machine. On an average, a performance improvement of 19%, 25%, and 23% is observed for the *fast*, *region*, and *global* methods, respectively, Again, as described previously, the heuristic uncertainties combined with the larger set of analyzed VRs caused the comparatively faster *region* method to perform better than the slower *global* method.

In summary, we prefer the *region* method as it performed close to the *global* method by employing a more intelligent heuristic with substantially faster compilation times.

3.5 Related Work

As an alternative to register windows, hardware and software schemes have been proposed in prior work to increase the effective number of registers. On the hardware side, register connection [51] and register queues [29, 84] have been proposed to increase the effective number of physical registers without changing the number of architectural registers using hardware/compiler support. Register connection uses special instructions to dynamically connect the core architectural registers to a larger set of physical registers.

With register queues, each register is connected to a queue of registers that are effective at maintaining values across multiple loop iterations in software pipelined loops [29, 84]. Both techniques introduce a layer of indirection to access every register operand. Further, additional hardware structures are used in their implementation to maintain the mapping between architected registers and physical registers. These techniques are generally targeted at high-performance platforms as their cost/power overhead are too large for embedded processors.

The register file can also be reorganized to deal with the problems of large register file sizes. Register caches [23] allow low latency register access while supporting a large architectural register file by caching a subset of the values of the register file in a smaller but faster register cache. The function units source their operands from the register cache. Clustering breaks up a centralized register file into several smaller register files, thereby creating a decentralized architecture [27, 28]. Each of the smaller register files supplies operands to a subset of the function units, and can be designed more efficiently. However, these techniques are used to reduce register file access time, porting, and interconnect complexity. They do not deal with the problem of limited encoding space and thus focus on orthogonal problems.

A combination of 16-bit and 32-bit instructions have been used in mixed-mode architectures like the Thumb instruction set extensions in ARM [80] and MIPS-16 [89] to provide a balance between reducing code size and retaining performance [53]. The register windows have the advantage over this approach of allowing scalability: the number of effective registers can be increased to any large number using a fixed encoding.

On the software side, code generation for DSP processors has proven to be a challenge for compilers [61]. The irregularities of such architectures have motivated the use of new compiler techniques which were initially considered to be complex and time consuming. Graph partitioning has been used in compilers for multi-clustered VLIW processors [3, 21, 24]. Several graph partitioning based tools like Chaco [37] and Metis [45] have been widely used to implement mutli-level Fiduccia-Mattheyses and other more sophisticated algorithms. These tools assign static weights to nodes and edges while our problem requires dynamic assignment of partition weights. A global register partitioning and interference graph-based approach has been used in the context of multi-cluster and multi-register file processors [20, 38]. Graph partitioning has also been explored in the context of partitioning program variables into multiple memory banks [59]. Our approach, on the other hand, tries to partition virtual registers into multiple register windows within a given procedure scope while trying to minimize spill code, inter-window moves and window swaps.

3.6 Conclusion

In this chapter, the design and implementation of a graph partitioning compiler algorithm to evaluate the benefits of a windowed register file design was discussed. Such a design increases the effective number of available registers while maintaining a fixed instruction encoding. The compiler partitions the virtual registers in a procedure into multiple register windows, thus reducing the overall spill code while minimizing the overhead due to inter-window moves and window swaps. The design was evaluated over a wide range of processor and window configurations. Increasing the number of windows from 1

to 2 yielded an average performance improvement of 10% for the 4-register case and 11% for the 8-register case on the WIMS processor. The corresponding experiment on a 5-wide VLIW machine, achieved an average performance improvement of 21% and 25% for the 4 and 8 register configurations, respectively. An average energy reduction of 25% for the 2-window 8-register over the 1-window case was observed on the WIMS processor.

CHAPTER IV

Compiler Managed Dynamic Instruction Placement In A Low-Power Scratch-Pad Memory

4.1 Introduction

In embedded processors, the instruction fetching subsystem can contribute to a large fraction of the total power dissipated by the processor. For example, instruction fetch alone contributes to, around 27% in the StrongARM SA-110 [25] and almost 50% in the Motorola MCORE [57], of the total processor power. Intuitively, this makes sense as instruction fetch is one of the most active portions of a processor. Instructions are fetched nearly every cycle, involving one or more memory accesses, some of which may be off-chip accesses. In this chapter, we focus on reducing the instruction fetch energy.

A number of approaches have been adopted by designers to reduce instruction fetch energy. First, more efficient instruction cache designs can be employed to reduce dynamic or leakage power [49]. Second, instruction compression techniques can be employed to reduce the number of instruction bits that need to be fetched [58]. Or third, bus encoding schemes can be employed to reduce the number of bits that switch each cycle [10].

Another approach that is particularly effective for embedded systems is to use loop

caches (LCs) or scratch-pads (SPs) [9, 22, 50, 56, 99, 102] [1]. LCs are small instruction buffers that can be designed to have extremely low-power per access. They are most effective when execution is dominated by small loops whose bodies can reside entirely within the LC. LCs can be broadly classified into two categories: hardware or software managed.

With a hardware managed approach, loops are dynamically copied into the LC and fetch is re-directed from the L1 instruction cache to the LC using limited hardware support [22, 57]. Hardware managed caches, referred to as filter or L0 caches [50], use cache-like tags but are often small and direct-mapped, hence their power characteristics and access time are much better than those of conventional caches. But, they suffer from high miss rates and cache management overhead.

To eliminate the overhead of tag comparisons, a tag-less LC has been proposed [56, 57]. Here, the LC is a small instruction buffer placed between the processor and the L1 instruction cache. A LC controller is responsible for identifying recurring code segments in the dynamic instruction stream, filling the LC, and redirecting fetch to the LC. This design can be more power efficient than the hardware tagged approach. However, there are several negatives of this approach, including LC controller complexity and the inability to relocate loops with control flow or subroutine calls. Moreovoer, the controller can make sub-optimal decisions as it does not have a complete view of the program execution.

Conversely, software managed SPs [71, 98] reduce hardware management overhead by relying on the compiler to insert code segments into the SP. A recent study [8] showed that

[1] In the literature, hardware managed instruction memories are called LCs, while the software managed memories are called SPs. In this chapter, we will be using both the names interchangeably to denote a sofware managed memory

the SP memory has 40% lower power consumption than a cache of equivalent size. The most common strategy is to statically map hot blocks into the SP using profile information [9, 35, 86]. Software static schemes have the advantage of no run-time copy overhead. Further, a global optimal placement can be performed to maximize the SP effectiveness over the entire run of the application. However, the major negative is that the SP contents cannot change during execution.

Software static schemes break down when a program has multiple important loops that collectively cannot fit in the SP. As a result, only a subset of the loops can be mapped into the SP. To overcome this problem, compiler-directed dynamic placement has been recently proposed [87, 103]. With this approach, the compiler inserts copy instructions into the program to copy blocks of instructions into the SP and redirect instruction fetch to the SP. As a result, the compiler can change the contents of the SP during program execution as it desires by inserting copy instructions at the appropriate locations. Compiler-directed dynamic placement has the potential to combine the benefits of the hardware-based schemes with the low-overhead of the software-based schemes. Previous approaches to dynamic placement use an integer linear programming (ILP) technique to find an optimal placement of instructions/data into the SP [87, 103]. But ILP-based approaches may not be practical in terms of run-time and often fail for moderate to large sized applications.

In this chapter, we evalute a new approach for compiler-directed dynamic placement. An inter-procedural heuristic for identifying hot instruction traces to insert in the SP is proposed. Based on a profile-driven power estimate, the selected traces are packed into the SP by the compiler, possibly sharing the same space, such that the run-time cost due

to copying the traces, is minimized. Through iterative code motion and redundancy elimination, copy instructions are inserted in infrequently executed regions of the code to copy traces into the SP. The approach works with arbitrary control flow and is capable of inserting any code segment into the SP (i.e., not just a loop body). A more detailed comparison of our work with the ILP-based solution is provided in Section 4.4.3.

The goal of the compiler support proposed in this thesis is to make effective use of the SP by dynamically copying instructions into it for programs with general control structures. To this end, a single instruction was added to the WIMS (see Chapter II) architecture, *LC_COPY*. The *LC_COPY* takes three source operands: the address of the first instruction of the region of code to be copied (PC relative), the starting chunk in the SP to begin placement, and the number of chunks to copy. The SP is logically divided into chunks, each being a fixed size (16 bytes for our experiments). A chunk represents minimum granularity at which copies can occur. By subdividing the SP into chunks, fewer bits to encode the operands were needed which was important to fit into the 16-bit encoding of the WIMS architecture. The *LC_COPY* instruction copies *number of chunks * size per chunk* bytes into the SP beginning at the starting chunk. The processor stalls while the copy takes place. The copying can be implemented using a direct memory access engine or within a software interrupt routine using loads and stores. In our experimental studies, we assume that the *LC_COPY* can copy 2-bytes per cycle.

Targets of branches into regions of the code that have been selected by the compiler are modified to point to the addresses in the SP where the region would be placed dynamically by the WIMS assembler/linker. Instruction fetch is thus redirected to the SP whenever

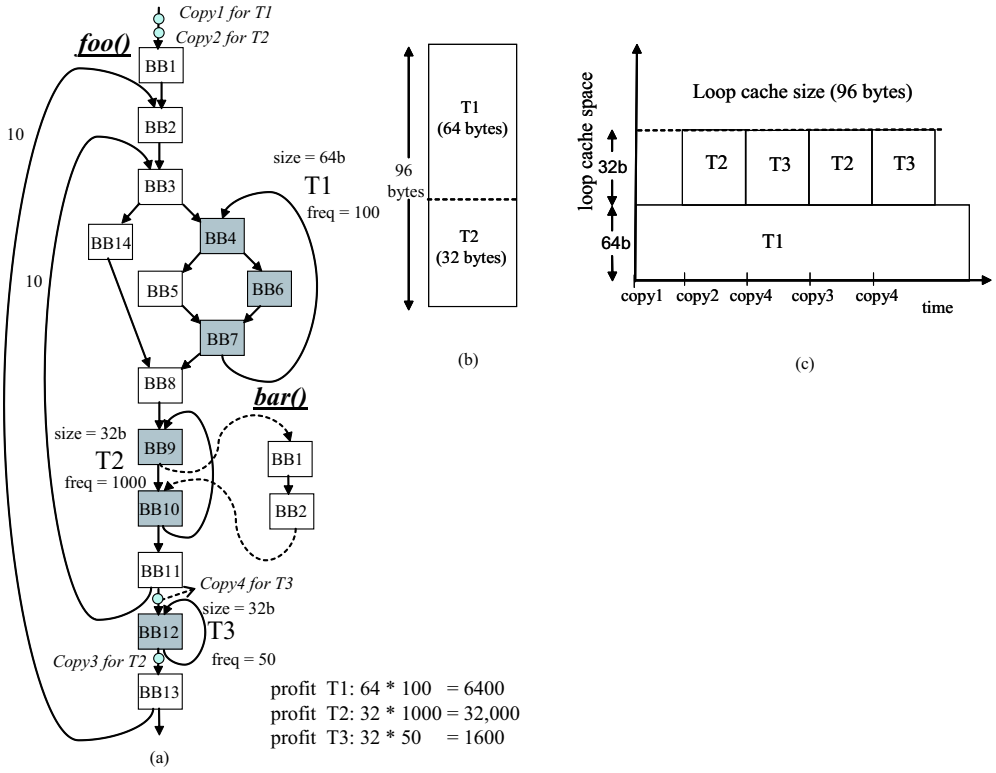


Figure 4.1: Example (a) weighted control flow graph (b) static allocation (c) dynamic allocation as a function of time.

control enters into a selected code region.

4.2 Dynamic Placement Motivation

To demonstrate the issues and benefits of dynamically copying instructions into the SP, consider the example shown in Figure 4.1. Figure 4.1(a) shows a control flow graph consisting of three hot regions shaded in gray. The shaded regions represent frequently executed sequence of basic blocks (BBs) in the code called *traces* [31]. Trace T1 consists of BBs 4, 6, and 7, T2 of BBs 9 and 10, while T3 contains a single BB, 12. These traces were identified by profiling the program on a sample input. Traces can include either a whole loop (e.g., T3), a part of a loop (e.g., T1), embedded procedure calls (e.g., T2), or any other complex control flow. The traces are annotated with the profile weights

(frequency) and size in bytes. The profile weights are 100, 1000, and 50 while the sizes are 64, 32, and 32 for T1, T2, and T3 respectively.

For illustration, assume the SP size is 96-bytes. The trace profit, which measures the desirability of placing a trace in the SP, is given by its size in bytes times the profile weight (Figure 4.1(a)). Figure 4.1(b) shows the contents of the SP for a static allocation scheme. As the SP can hold only 96-bytes, the static scheme packs only the top two profitable traces, T1 and T2, of sizes 64 and 32-bytes, respectively, into the SP. But, the dynamic scheme (Figure 4.1(c)), is able to allocate all traces by inserting copy instructions as shown on the edges in Figure 4.1(a). Copy 1 (for T1) is executed once before entering the inner loop, thus T1 remains in the SP throughout its lifetime. Copies 2 and 3 (both for T2), and copy 4 (for T3) alternately insert T2 and T3 into the same location in the SP. Each copy ensures that the trace is inserted into the SP before they are executed. It should be noted that copy 3 and copy 4 have to be placed within the outer loop to copy traces T2 and T3 prior to their execution. Placing any of these copies outside can cause the corresponding loop to be overwritten by the other causing illegal transfer of control. By effectively overlapping multiple blocks of code and placing copies appropriately, the dynamic scheme is able to capture all the hot regions and thus achieve better SP utilization. This approach was first used in pre-virtual memory management operating systems for overlaying code for different processes [83].

The remainder of this chapter explains the compiler algorithm to automatically identify and place frequently executed regions in the SP. Additionally, copies are inserted such that the placement decisions are honored while minimizing the copying overhead.

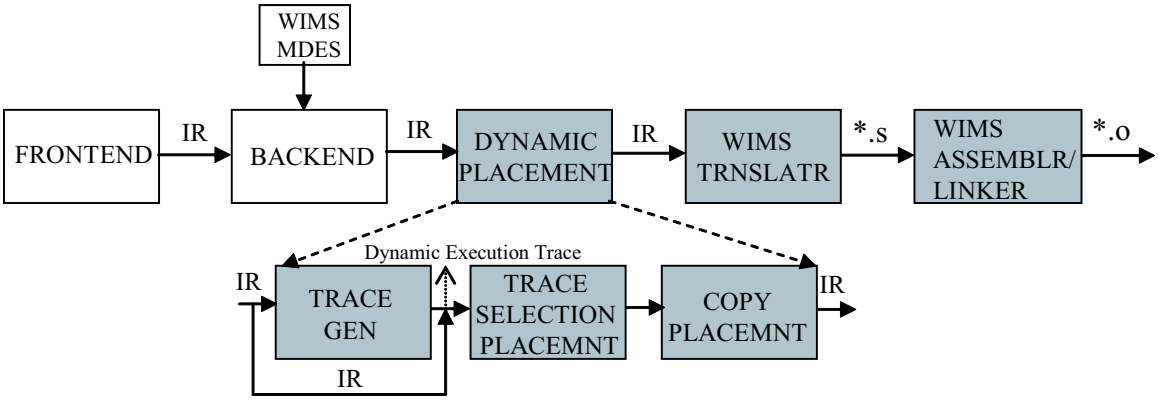


Figure 4.2: Overall compiler system for dynamic instruction placement in scratch-pad.

4.3 Dynamic Placement

4.3.1 Overview

The dynamic instruction placement scheme has been implemented within the Trimaran [96] compiler framework as shown in Figure 4.2. The compiler frontend performs control flow profiling and annotates the intermediate representation (IR) with *traces* [31]. Traces are frequently executed linear sequences of basic blocks that are contiguously laid out in memory [16]. Traces are formed with a 60% probability of an in-trace transition and with size limited to that of the SP. These traces are considered as candidates for placement into the SP. The dynamic placement phase, based on the execution profile information, then inserts the *LC_COPY* instructions into the IR. The WIMS assembler/linker assigns instructions to physical memory locations including adjusting of the branch targets for the relocated code.

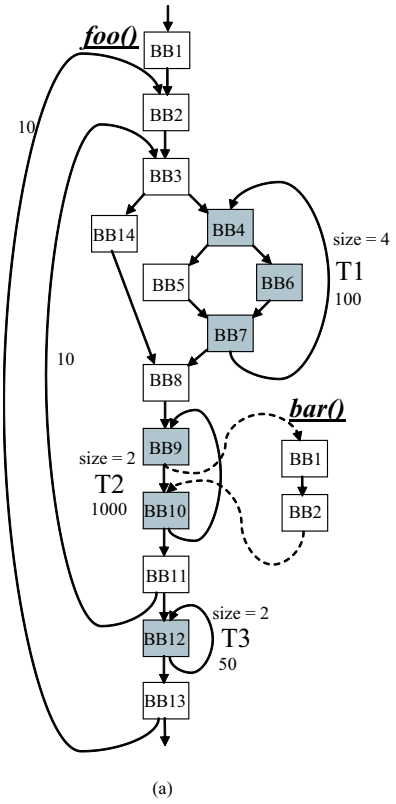
The dynamic placement algorithm has two objectives: (i) select traces from the program and place them into locations in the SP such that the energy benefit is maximized, and (ii) place copy instructions so that traces are copied prior to execution while mini-

mizing the overhead due to copying. To achieve these objectives, the dynamic placement is divided into two distinct phases - trace selection/placement and copy placement. The trace selection/placement phase, using the execution profile information and the annotated traces from the IR, selects the most beneficial traces and decides where they are to be placed in the SP using an energy benefit heuristic. The placement phase could possibly overlap the traces within the SP. The placement decisions are driven by energy considerations and do not take performance into account. Following this, the copy placement phase naïvely inserts copy operations on every entry edge of a selected trace in the IR. This ensures that whenever control reaches a trace that has been placed in the SP, it is copied prior to execution. Many of these copies may be redundant or present on highly executed paths, thus causing high copy overhead. Based on a liveness analysis scheme, the copies are then hoisted in the control flow graph (CFG) across procedure boundaries to less frequently executed blocks so as to reduce the copy overhead while maintaining correctness of execution.

The example in Figure 4.1 is used throughout this section to illustrate how the candidate traces, T1, T2, and T3 are placed in the SP and how subsequent copy insertion and hoisting are performed for these selected traces. The CFG is redrawn in Figure 4.3(a) for convenience. The trace selection/placement and the copy placement phases are detailed in the sections below.

4.3.2 Trace Selection/Placement

The trace selection phase takes as input the IR annotated with the traces. Traces are chosen as candidates for SP allocation for two reasons. First, they help reduce the num-



Benefit:

ProfileWeight * (TraceSize * ChunkSize)/FetchSize * (MMFetchEnergy - LCFetchEnergy)

T1: 100*((4*16)/2)*(2-1) = 3200 nJ
T2: 1000*((2*16)/2)*(2-1) = 16,000 nJ
T3: 50*((2*16)/2)*(2-1) = 800 nJ

CopyCost:

(TraceSize * ChunkSize)/FetchSize * (MMFetchEnergy + LCWriteEnergy)

T1: ((4*16)/2)*(2+1) = 96 nJ
T2: ((2*16)/2)*(2+1) = 48 nJ
T3: ((2*16)/2)*(2+1) = 48 nJ

(b)

NetGain = Benefit – CopyCost

T1: 3200 – 96 = 3104 nJ
T2: 16000 – 48 = 15952 nJ
T3: 800 – 48 = 752 nJ

(c)

Figure 4.3: Trace selection and placement example. (a) CFG (b) Benefit and CopyCost computation for traces T1, T2, and T3.

ber of copies as a single copy instruction can copy a large amount of frequently executed code, like a loop body, into the SP. Second, a trace is a high frequency path of execution consisting of basic blocks connected by fall-through edges; thus, the number of control flow transfers in and out of the SP is reduced. Conversely, traces are fine grained enough to enable selection of small hot program segments for general applications. For our experiments, we limit the size of the traces to 64 instructions as larger traces may exceed the size of the SP and thus will not be considered for placement. Only those traces that fit in the SP and that are executed at least 1% times are selected for potential allocation.

Trace selection/placement involves picking traces and placing them in the SP such that there is a savings in instruction fetch energy. If a trace is placed in the SP, then

whenever the trace is executed, it has to be executed out of the SP. This is required as all branches into the trace have their offsets changed to the location in the SP where the trace will be placed. Thus, the trace needs to be copied prior to execution which involves a copy overhead. Also, the exact placement of the trace in the SP is important because if traces overlap in the SP, repeated copies may be required. The trace selection/placement algorithm selects and places a trace at a particular location in the SP only if there is an overall energy benefit for that trace. The trace selection/placement consists of two steps - (i) computing the energy gain for every trace, and (ii) placing the trace into the SP.

Computing Trace Energy Gain: For a trace to be considered as a candidate for placement in the SP, the energy savings obtained in executing the trace out of the SP must be greater than a one-time copy overhead. Thus, traces with a higher copy overhead than the potential energy gain are non-beneficial and can be filtered out. For every trace, T_i, the copy cost and benefit of placing the trace in the SP is initially computed assuming that the SP is of infinite size and the trace does not overlap with any other trace. Since copying into the SP takes place at the chunk granularity, the size of the trace is computed in number of chunks. In Figure 4.3(a), T1, T2, and T3 are assumed to take 4, 2, and 2 chunks, respectively. The cost of copying a trace into the SP is the sum of the energy needed to fetch the trace from the main memory and write the trace into the SP. The copy cost (measured in nJoules) is given by the equation:

$$CopyCost(T_i) = \frac{TraceSize(T_i) * ChunkSize}{FetchSize} * (MMFetchEnergy + SPWriteEnergy) \quad (4.1)$$

where $TraceSize(T_i)$ is the size of T_i in chunks, *ChunkSize* is the size of a single chunk (assumed 16-bytes), and *FetchSize* is the number of bytes accessed per fetch from main

memory to the SP (assumed 2-bytes per access). $MMFetchEnergy$ and $SPWriteEnergy$ are the energy required for a single fetch from main memory and a single write into the SP, respectively.

The benefit of placing a trace in the SP is the savings in energy obtained when the trace is executed out of the SP as opposed to executing from main memory. The benefit (measured in nano-Joules) is given by the equation:

$$Benefit(T_i) = ProfileWeight(T_i) * \frac{TraceSize(T_i) * ChunkSize}{FetchSize} * (MMFetchEnergy - SPFetchEnergy) \tag{4.2}$$

where $SPFetchEnergy$ is the energy required for a single fetch out of the SP and $ProfileWeight(T_i)$ is the execution frequency of T_i obtained through profiling. The calculation of $CopyCost$ and $Benefit$ for traces T1, T2, and T3 for the running example are shown in Figure 4.3(b). Here, we assume that the main memory fetch energy is 2 nJ, while SP fetch/write energy is 1 nJ. The net energy gain of placing a trace is the difference between the benefit and the copy cost defined as, $NetGain(T_i) = Benefit(T_i) - CopyCost(T_i)$, as shown in Figure 4.3(c). Traces for which the net gain is less than zero are not considered further for placement.

Placing Traces into the Loop Cache: The placement algorithm decides where each trace is placed in the SP. A trace occupies continuous locations in memory and hence is assigned to a sequence of contiguous chunks. Thus, the placement algorithm has to decide on the best starting chunk. If a given trace solely occupies a sequence of chunks, then the benefit of placing the trace is the same as $NetGain$. But, this is simply static placement. Dynamic placement allows multiple traces to occupy the same SP chunk.

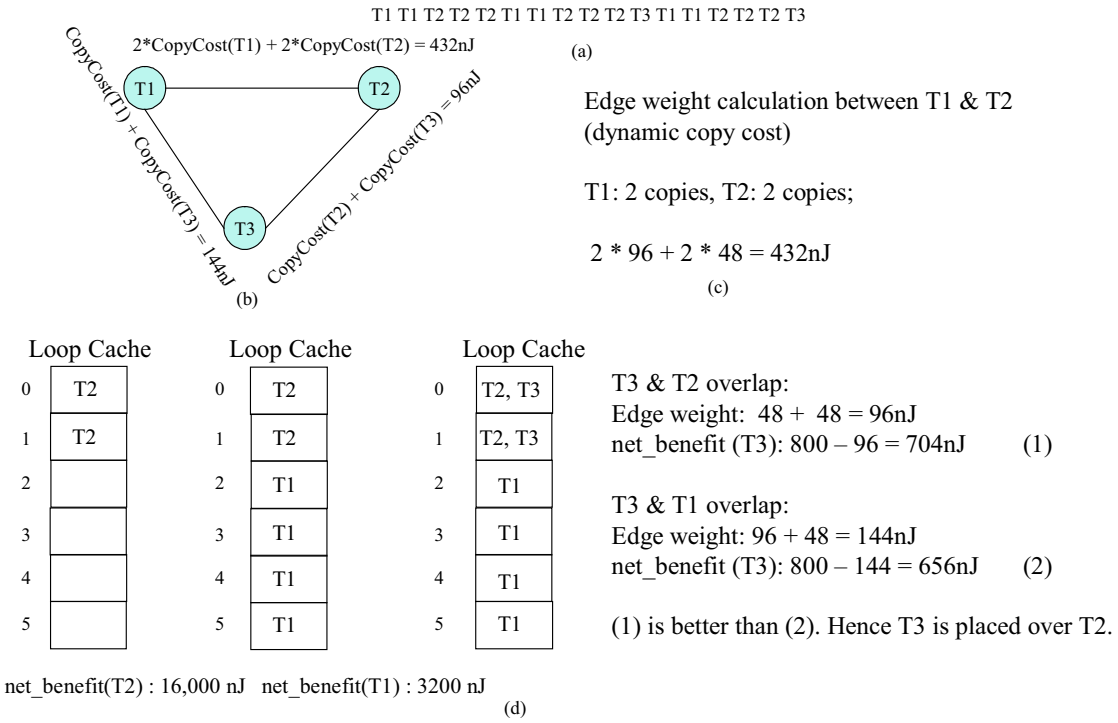


Figure 4.4: Trace selection and placement example. (a) Dynamic execution trace (b) Temporal relationship graph. (c) Edge weight calculation between nodes T1 and T2 (d) Placement of T1, T2, and T3 into the SP.

However, a trace must be recopied whenever it gets displaced by an overlapping trace. Dynamic placement is successful when the combined benefit of placing multiple traces is more than the overhead due to repeated copying.

In order to compute the overhead due to repeated copying, the previous cost/benefit analysis is extended to account for the additional copying cost (*Dynamic_Copy_Cost*). To this end, a temporal relationship graph [33] (TRG) is constructed based on a *dynamic execution trace*. The dynamic execution trace consists of traces and is obtained during execution profiling. Figure 4.4(a) shows the dynamic execution trace for a sample run

of the program in Figure 4.3(a). The TRG helps to estimate the number of dynamic recopies required if two traces overlap in the SP. Based on the TRG, the placement algorithm then overlaps multiple traces within the SP so as to reduce overall copy overhead while maximizing the energy gain.

The TRG captures the dynamic copy cost between pair of overlapping traces. The nodes in the TRG are the traces, while the edges are annotated with the dynamic copy cost. Between every pair of nodes T_i and T_j, the edge weight denotes the number of copies of T_i ($CopyCost(T_i)$) for any T_j that occurs between every two consecutive occurrences of T_i in the dynamic execution trace. A T_j occurring between two consecutive instances of T_i implies that T_i needs to be recopied prior to its second occurrence, if T_i and T_j overlap in the SP. The TRG is constructed by linearly scanning the input dynamic execution trace and maintaining a queue of currently seen traces. Each new trace T_i, seen in the input, is added to the queue. The queue is then scanned, starting from the tail, for a previous occurrence of T_i. For every unique trace T_j seen prior to the previous occurrence of T_i, the edge weight between T_i and T_j is incremented by the copy cost of T_i. The previous occurrence of T_i is then deleted from the queue as the new instance of T_i becomes the next previous occurrence.

The TRG for the dynamic execution trace in Figure 4.4(a) is shown in Figure 4.4(b). Considering T1 and T2 alone, there is an instance of T2 between every instance of T1 and vice-versa. Thus, if T1 and T2 overlap in the SP, two recopies of both T1 and T2 are required. The edge weight between nodes T1 and T2 is therefore $2*CopyCost(T1)+2*CopyCost(T2)$ (Figure 4.4(c)), where *CopyCost* is computed as shown earlier in Equa-

Algorithm 4.1: Pseudo code to select and place traces in the scratch-pad.

```
Input: SortedTraceList = Traces sorted in decreasing order of NetGain
foreach (trace T ∈ SortedTraceList) do
    benefitFound = false ;
    for (c = 0 to NUM_SP_CHUNKS) do
        NumChunks(T) = Number of SP chunks occupied by trace T ;
        if (c+NumChunks(T) > TOTAL_NUM_SP_CHUNKS) then
            break;
        end
        IntersectTraces = set of intersecting traces at SP chunks c to c + NumChunks(T) ;
        DynamicCopyCost = ComputeCopyCost(T, IntersectTraces) ;
        net_benefit = Benefit(T) - DynamicCopyCost ;
        if (netBenefit > currMaxBenefit) then
            currMaxBenefit = netBenefit ;
            bestStartChunk = c ;
            benefitFound = true ;
        end
    end
    if (benefitFound) then
        Place trace T at bestStartChunk ;
    end
end
```

Algorithm 4.2: Compute_Copy_Cost function that computes the cost of copying trace T

```
Input: Trace T and the set of intersecting traces IntersectingTraces
return edgewt: Sum of edge weights of T and the traces in IntersectingTraces from the TRG
edgewt = 0 ;
foreach (Trace T_i ∈ IntersectingTraces) do
    edgewt += Edge weight between T and T_i, from the TRG ;
end
```

tion 4.1. Intuitively, the edge weights measure the overhead due to conflicts in the SP when the nodes (traces) that share the edge are made to share SP chunks. The dynamic copies represent the minimum set of copies for a sample input. But in reality, the compiler may not be able to achieve this as it has to conservatively insert copies to ensure the legality constraint (see Section 4.3.3).

The placement algorithm uses the dynamic copy costs (edge weights in the TRG) to place each trace in the SP. The pseudo code for the trace selection/placement is shown

in Algorithm 4.1. Traces are considered for placement in the decreasing order of gain ($NetGain > 0$). Each trace is considered at a particular chunk using *ComputeCopyCost* function defined in Algorithm 4.2. and is greedily placed at the SP index with the maximum *netBenefit*.

Figure 4.4(d) shows how T1, T2, and T3 are placed in the SP. Initially, since the SP is empty, T2, the highest benefit trace with *netBenefit = Benefit*, is placed at chunk 0. T1 is placed at chunk 2 where there is maximum *net_benefit* and zero interference. Since the SP is now full, T3 has to overlap with either T1 or T2. The dynamic copy costs when T3 overlaps with T2 and T1 (edge weights between T3-T1 and T3-T2) are shown on the right in Figure 4.4(d). Since both the choices have positive *netBenefit*s, there is an advantage in placing T3 in the SP. The *netBenefit* when T3 overlaps with T2 is higher than T1, hence T3 is placed at chunk 0 overlapping with T2. A trace is tried at all possible starting chunks. If no benefit is seen, the trace is never placed in the SP and is always executed out of the main memory.

It should be noted that traces are obtained for the whole program. This allows selection and placement of traces across all procedures such that the conflicts are minimized. The placement heuristic is a greedy heuristic, giving preference to traces of highest benefit first. Alternately, an integer-linear programming based method could be employed to find an optimal selection and placement of traces. But this gets impractical for reasonably sized programs. The greedy heuristic, though not an optimal solution, works well in practice. The selection and placement algorithm is similar to code placement techniques for cache miss rate reduction where the code is reorganized to minimize conflict misses and improve

locality [33, 95].

4.3.3 Copy Placement

The goal of the copy placement phase is to insert *LC_COPY* instructions subject to the following issues:

- A selected trace should always be present in the SP when control enters the trace. If two traces T1 and T2 overlap in the SP, a copy of T2 could invalidate T1. Thus, after control leaves T2 and before T1 gets executed, T1 needs to be recopied.

- A copy of a trace should ideally occur only when it is needed. If a trace is already in the SP and has not yet been displaced, then it is pointless to recopy the trace. Thus, a copy should be inserted only when required. Since copies are stalling, redundant copies not only consume power but also affect performance.

The copy placement algorithm handles the two issues using a phased approach. Initially, copies are inserted on all edges of the CFG that enter a trace. This naïvely guarantees that the trace is copied into the SP before execution regardless of which other traces displace it. Thus, this ensures correct but inefficient execution. Following this, a phase of iterative copy hoisting and redundant copy elimination is performed. Iterative copy hoisting attempts to hoist copies from their initial locations up the CFG, across procedure boundaries, to infrequently executed blocks subject to legality constraints. The legality constraint is that there should be sufficient copies to ensure that the trace is copied prior to execution if displaced by another overlapping trace. Figure 4.5(a) shows the initial location of copies in the CFG. We use the convention *Cij* to denote the j^{th} copy for trace *i*. In

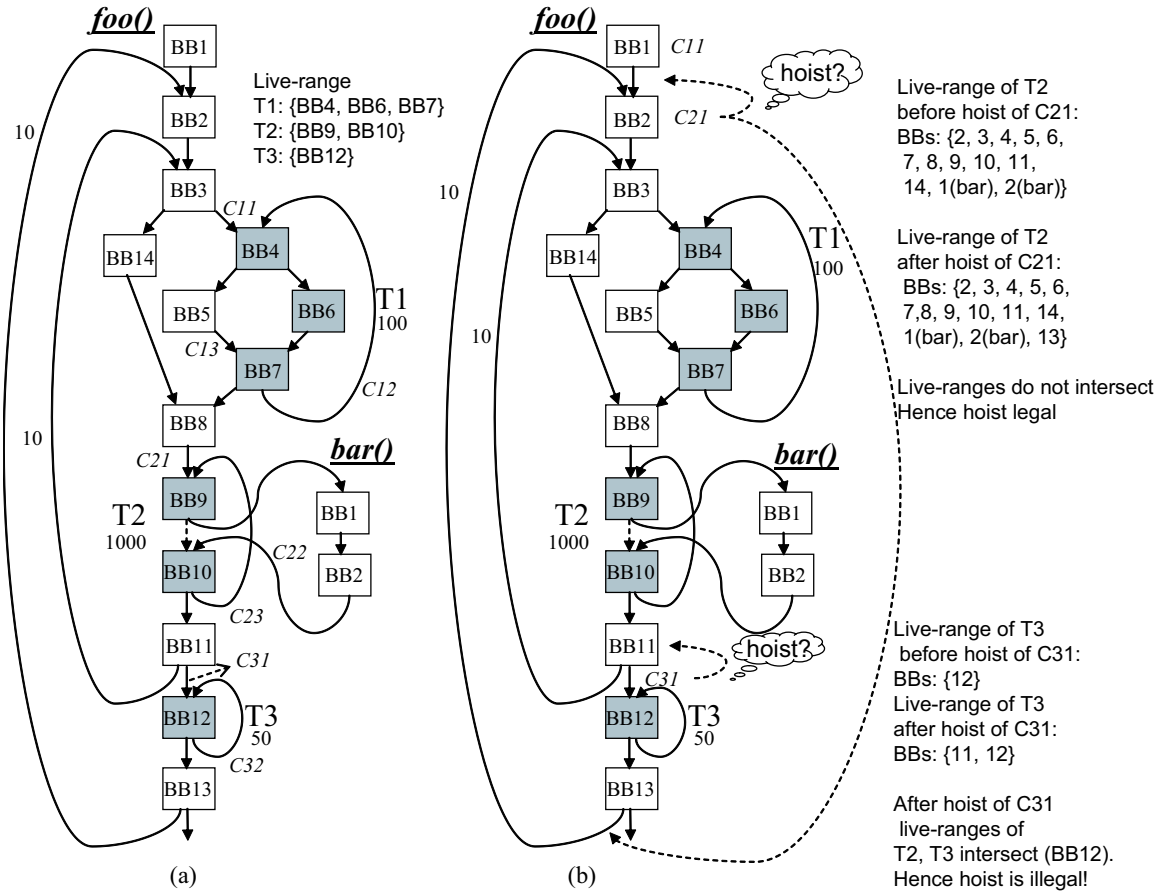


Figure 4.5: Copy placement example. (a) Initial copies inserted (b) Hoisting of copies and live-range computation

the previous section, the trace selection/placement algorithm overlapped traces T2 and T3. Naïvely, if all copies for traces T2 and T3 are moved to BB1, then copies would overwrite each other. Thus without recopies, this would cause illegal execution. Also, since no other traces overlap T1, copies for T1, C12 and C13, are redundant. Hence, two of the three copies are removed. T1 requires just a single copy in BB1.

Copy insertion and hoisting are performed on a global CFG of the entire application, including all procedures, represented within the underlying IR. The global CFG connects all procedures with their call sites. For indirect calls, edges are drawn conservatively from

the call site to all possible targets. Initial copies are placed on the edges of the CFG by creating extra pseudo BBs at the edges. In Figure 4.5(a), for sake of clarity, we show the initial copies on the edges.

Iterative copy hoisting and redundant copy elimination are performed in the following steps.

Live-Range Construction: Copies should be hoisted while being cognizant of the above mentioned legality constraint. To this end, dataflow analysis techniques are applied to the CFG. Intuitively, a trace needs to reside in the SP from the copy point until the point when control leaves the trace and never gets back to the trace. This is akin to live-ranges used in register allocation [15]. The live-range of a trace is defined as the set of blocks in the CFG starting from the point where the copy of the trace is defined until the last use of the trace.

To compute the live-range of a trace, we need to compute the blocks that are live-in to each trace and the blocks the copy reaches. Thus, traditional liveness and reaching-defs analysis [1] can be carried out for each trace to compute its live-range. Each trace identifier T_i is modeled as a variable. The copy for a trace "defines" the trace and is assumed to be the first instruction in the BB where the copy is placed. The "use" of a trace includes all BBs that comprise the trace. For a given trace, there can only be one copy of that trace in a block. While for a given block, there can be multiple copies corresponding to different traces. The copy for a trace can "kill" another copy for the same trace. The initial live-ranges consists of just the blocks in the traces and are shown in Figure 4.5(a) for traces T1, T2, and T3. Although not shown in figure, the live-ranges also include the pseudo blocks

on edges where the copies are present.

The live-ranges are initialized based on the defines and uses of traces For a block *b*, DEF(b) has a '1' for each trace that has a copy in the block. Similarly, USE(b) has a '1' for a trace if that block *b* is part of the trace. Once the DEF and USE vectors are computed, iterative data flow analysis [1] is used to compute the LIVEIN and LIVEOUT sets. The LIVEIN and LIVEOUT sets $\forall BB_i$ is defined as

$$LIVEIN(BB_i) = USE(BB_i) + (LIVEOUT(BB_i) - DEF(BB_i))$$

$$LIVEOUT(BB_i) = \bigcup_{BB_j \in succ(BB_i)} LIVEIN(BB_j)$$

where, succ(BB_i) is the set of successor blocks of BB_i in the CFG.

Similarly, to compute the reaching definitions of a trace, four bit-vector sets GEN, KILL, INDEF, and OUTDEF are defined for every BB. Each element in the bit vector, unlike liveness analysis, corresponds to the copy operations that defines traces. For every BB *b* and for every copy in that BB, GEN(b) has a '1' for that copy. For a block that has a copy of a trace T_i, KILL(b) has a '1' for all other copies of T_i in other blocks that are 'killed' by the copy present in *b* for T_i. Similar to liveness analysis, INDEF and OUTDEF sets can be defined based on GEN and KILL sets. The only difference being reaching definitions is a forward dataflow analysis problem while liveness is a backward dataflow analysis problem.

Once the liveness and reaching definitions are computed, the live-range of a trace can be defined as the intersection of liveness and reaching definitions. Formally, $\forall T_i$ in the CFG,

$$LiveRange(T_i) = \{BB_i \mid (T_i \in (INDEF(BB_i) \wedge LIVEIN(BB_i))\}$$

Algorithm 4.3: Pseudo code to hoist and eliminate redundant copies.

```
Input: CFG with copies inserted before entry of each trace
EliminateRedundantCopies() ;
CopyQ = List of copies for all traces sorted in decreasing frequency order ;
while (!CopyQ.isEmpty()) do
    C = CopyQ.pop() ;
    bb = BB containing the C ;
    PredSet = Predecessor BBs of bb ;
    while (!PredSet.isEmpty()) do
        Remove C from bb ;
        Hoist copies to BBs in Pred_Set ;
        Compute_Live_Ranges() ;
        if (LiveRangesIntersect()) then
            undo hoist ;
            CopyQ.remove(C) and finalize copy in bb ;
            break ;
        end
        else
            EliminateRedundantCopies() ;
            OldFreq = freq of C ;
            NewFreq = sum of frequencies of copies after C is hoisted ;
            benefit = NewFreq − OldFreq ;
            if (benefit ≥ 0) then
                hoist is successful ;
                insert the new copies in BBs contained in PredSet into the CopyQ ;
                break ;
            end
            else
                PredSet = new set of predecessors of PredSet ;
            end
        end
    end
    CopQ.remove(C) and finalize copy in bb ;
end
```

Once the live-ranges are constructed, the legality constraint can be defined as follows. If the live-range of two overlapping traces T_i and T_j intersect, then there is some path in the program flow where a copy of T_i would displace T_j before T_j is recopied.

Copy Hoisting: The iterative hoisting algorithm tries to move copies that are in BBs that are frequently executed, up the CFG to BBs of lower frequencies while maintaining the legality constraint. The algorithm has two goals while hoisting copies : (a) reduce the overhead of executing the copies, and (b) ensure that the copies are present at the appropriate points in the program such that the traces are copied prior to their execution.

The initial copy placement guarantees (b), but at the expense of executing copies even if the trace is not displaced by another overlapping trace.

These two goals conflict with each other. On the one hand, hoisting copies up the CFG to blocks of lower frequency is beneficial as it reduces the dynamic copy cost. But, on the other hand, the live-range of the trace corresponding to the copies grow longer as the copies are hoisted higher. This can interfere with the live-ranges of other traces that overlap with this trace, thus violating the legality condition. Alternately, it could prevent other copies from getting hoisted to ensure the legality condition. The copy hoisting algorithm addresses this problem by hoisting the most frequently executed copy only while it is the highest execution frequency. When its frequency decreases, hoisting is iteratively performed on the new highest frequency copy and so on.

The pseudo code for iterative copy hoisting is shown in Algorithm 4.3. The *EliminateRedundantCopies* function (using dominator analysis), eliminates unnecessary copies of a given trace. A copy is redundant if the BB in which the copy is placed is dominated by another block which contains another copy for the same trace. The elimination includes a check for intersecting live-ranges for legality. In Figure 4.5(a), copy *C11* dominates copies *C12* and *C13,* hence *C12* and *C13* are eliminated. The rest of the copies for all traces are sorted in decreasing order of frequency. The hoisting algorithm picks the copy with the highest frequency and iteratively tries to hoist it to its predecessor blocks. If the live-range intersects with another overlapping trace, the copy is required at the current block and is not hoisted further. If the hoist is legal and the sum total of frequencies for the set of copies after the hoist is lower than the sum of the frequencies for the set of copies before

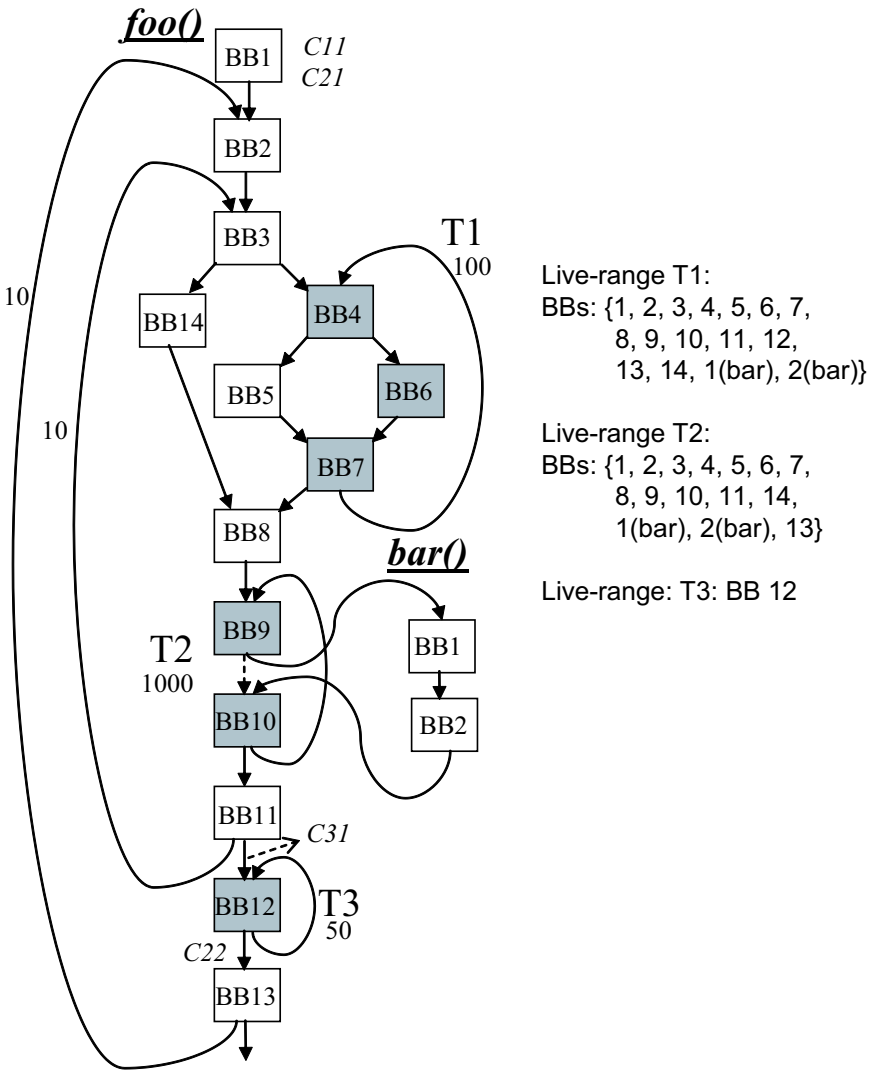


Figure 4.6: Final copy placement for example in Figure 4.5(a).

the hoist, the algorithm can claim benefit and confirm the hoist.

Figure 4.5(b) illustrates the hoisting algorithm. Assume that the copies have been hoisted to the currently shown positions from those shown in Figure 4.5(a). C21 was moved from its home location on the edge from BB 8 to BB 9 (Figure 4.5(a)) to its current position in BB 2 (Figure 4.5(b)) which is outside the inner loop and hence of lesser frequency. It should be noted that in the example, copies can be hoisted to edges. Subsequently, BBs are instantiated at the edges (not shown in example) to house these copies.

Assume that C11 has moved all the way up to the entry block BB1. Since T1 does not overlap with any other trace, this move is legal. C21 is the next copy for which hoisting is attempted to all incoming edges of its home block, BB2. The predecessors are the edges from BB1 to BB2 and the backedge from BB13 to BB2 (dotted lines). The live-ranges of T2 before and after the hoist of C21 are shown in Figure 4.5(b). The live-ranges do not intersect with other traces, thus the hoist is legal. Moreover, since the sum of the frequencies of the incoming edges is the same as the frequency of BB2, the hoist is considered beneficial. Next, C31, which is of the next highest priority, is hoisted from its home (edge from BB11 to BB12) to its predecessor block BB11. This causes the live-ranges of T2 and T3 to intersect. Since T2 and T3 overlap in the SP, this hoist is illegal.

The final copy placement and live-ranges are shown in Figure 4.6. The initial copies of traces T1 and T2 are performed in BB1. Before control enters T3, copy C31 is performed. After control leaves T3, T2 is copied back via copy C22. For a copy on an edge, a new basic block is created and inserted into the CFG. If a copy materializes on the return edge of the CFG (BB2 of *bar* to BB10 of *foo*), then the copy is inserted after the procedure call within the caller. While if a copy materializes on the call edge (BB9 of *foo* to BB1 of *bar*), it gets inserted before the procedure call in the caller.

Discussion: It should be noted that by using a global CFG, we are able to hoist copies across procedure boundaries. Considering each procedure independently restricts copy hoisting to the entry block of the procedure resulting in substantial copy overhead. Our original design was not inter-procedural and suffered large energy and performance penalties due to this problem. The global CFG also allows a copy to cross procedure calls,

thus reducing the copy overhead significantly.

There are two alternatives that could be considered to handle the hoisting problem. The problem could be formulated as a form of code motion and use techniques like lazy code motion [52]. However, lazy code motion is not ideal because each instruction is positioned in sequence at the point of highest profitability, thereby giving one copy complete priority over others. More importantly, if the live-range intersections are not taken into account while hoisting, the legality condition can be violated. Conversely, by hoisting the most profitable copy all the way up, its live-range is increased. This can prevent the hoisting of other copies as it would intersect with the live-range of the trace corresponding to the hoisted copy. Thus, interactions between multiple copies must be considered.

Alternately, one could place the copies in the prologue block of the 'main' procedure. This would cause the live-ranges of all traces to intersect. These live-ranges could then be 'split' by inserting copies at less costly points. But, live-range splitting heuristics employed in register allocation focus on reducing the register pressure so that a later coloring phase can allocate the live-ranges with reduced spill [12]. In our case, the traces have already been selected and placed if necessary in overlapping chunks in the SP. The copy placement and hoisting must guarantee the legality of the placement by introducing appropriate 'spills' (recopies) at reduced copy overhead.

4.4 Experimental Evaluation

4.4.1 Methodology

Our experimental framework consists of a port of the Trimaran compiler system [96] to the WIMS (see Chapter II) processor and a modified version of the WIMS processor

simulator to model a parameterized SP. Note that since the WIMS processor is single-issue, many of the VLIW transformations, except function inlining, in Trimaran were disabled for these experiments. The simulator uses a simple energy model attributing a fixed energy to each SP or memory access for instruction fetch and totaling it up across the run of an application. On the WIMS processor, instruction fetch energy accounts for approximately 30% of the overall system energy including the processor and memory.

For this study, a set of embedded benchmarks selected from the MediaBench and MiBench suites were chosen. For all experiments, compiler-directed dynamic placement (dynamic for short) is compared with compiler-directed static placement (static for short) that uses profile information to maximally pack the most frequently executed regions into the SP. In addition, a comparison against varying sizes of traditional instruction caches (icache for short) was also performed. For each of the experiments, two measures are presented. First, the instruction fetch energy consumption of each technique with respect to the baseline where all instructions are fetched from main memory. Second, the hit rate is the ratio of accesses (SP or icache) to total instruction references.

Two memory configurations were used - WIMS and CACTI. For the WIMS processor, SP size was varied from 32 to 4k bytes for the study. With each SP size, the energy presented in Table 4.1 (top) was assumed for each access. The energy consumption estimates were obtained from configuration specific data sheets from a popular memory compiler for a 0.18μm process. These values compare to 0.1384 nJ per read of a 16Kb bank from the regular on-chip memory. The access times for the on-chip main memory and SP are both one cycle.

size (bytes)	read (nJ)	write (nJ)
32	0.0506	0.0388
64	0.0527	0.0413
128	0.0568	0.0463
256	0.0651	0.0563
512	0.0698	0.0701
1024	0.0990	0.1174
2048	0.1020	0.1228
4096	0.1197	0.1416

size (bytes)	fetch (SP) (nJ)	fetch (Icache) (nJ)
64	0.1803	0.2961
128	0.1888	0.3059
256	0.1980	0.4732
512	0.2188	0.4966
1024	0.2404	0.5233
2048	0.2748	0.5655
4096	0.3277	0.6351

Table 4.1: Per access scratch-pad energy for the WIMS processor (top) and per access scratch-pad and icache energy using CACTI (bottom, .18μm) for different sizes.

To compare against an icache of equal size, CACTI [104] was used to obtain the energy numbers for different sizes of icache and SPs. Here, the icache and SP are assumed to be on-chip, while the main memory is off-chip. For SP, the energy for the tag and comparator circuits were subtracted from a correspondingly sized direct-mapped icache [8]. The SP and icache sizes were varied from 64 to 4k [2]. For the icache, we assumed 16-byte line size with 2-way associativity to get the energy numbers as shown in Table 4.1 (bottom). The Am41PDS3228D SRAM [4] was assumed to be the off-chip memory with 3.024nJ per access (16-bits). The icache hit rates were obtained using the Dinero-IV cache simulator [26]. While comparing static and dynamic with icache, all measurements were obtained using the CACTI energy numbers. For comparing static versus dynamic for the WIMS processor, all measurements were obtained relative to the WIMS energy model.

[2] CACTI did not support a 32-byte cache

For all studies, a single read/write from the cache to the main memory is assumed to be 2-bytes. In Table 4.1, although there is a difference in absolute energy values between the WIMS and the CACTI models, this is less important as we do relative comparisons within each class. The difference in energy numbers between WIMS and CACTI is due to an abstract energy model used by CACTI as opposed to real datasheet numbers used for WIMS. The loop cache energy for the dynamic scheme is obtained using the following equation:

$$SP\_Hit\_Rate * SPFetchEnergy + SP\_Miss\_Rate * MMFetchEnergy + SP\_stalls * (SPWriteEnergy + MMFetchEnergy) \tag{4.3}$$

where, SP_stalls is the number of cycles that the processor is stalled while executing the *LC_Copy* instruction[3]. The icache energy is computed as follows:

$$Hit\_Rate * IcacheReadEnergy + Miss\_Rate * NumAccesses * (IcacheWriteEnergy + MMFetchEnergy) \tag{4.4}$$

where, *NumAccesses* is the number of fetches required to bring a single cache-line into the icache during a miss.

4.4.2 Results

Comparison with static: The energy savings and SP hit rates for static and dynamic placement are compared across all the benchmarks for SP sizes of 64 and 256 bytes in Figure 4.7 for the WIMS processor. Considering first the 64 byte SP, dynamic is generally more effective at utilizing the SP. The largest energy benefits occur for cjpeg, unepic, and

[3]the rest of the terms are described in Section 4.3.2

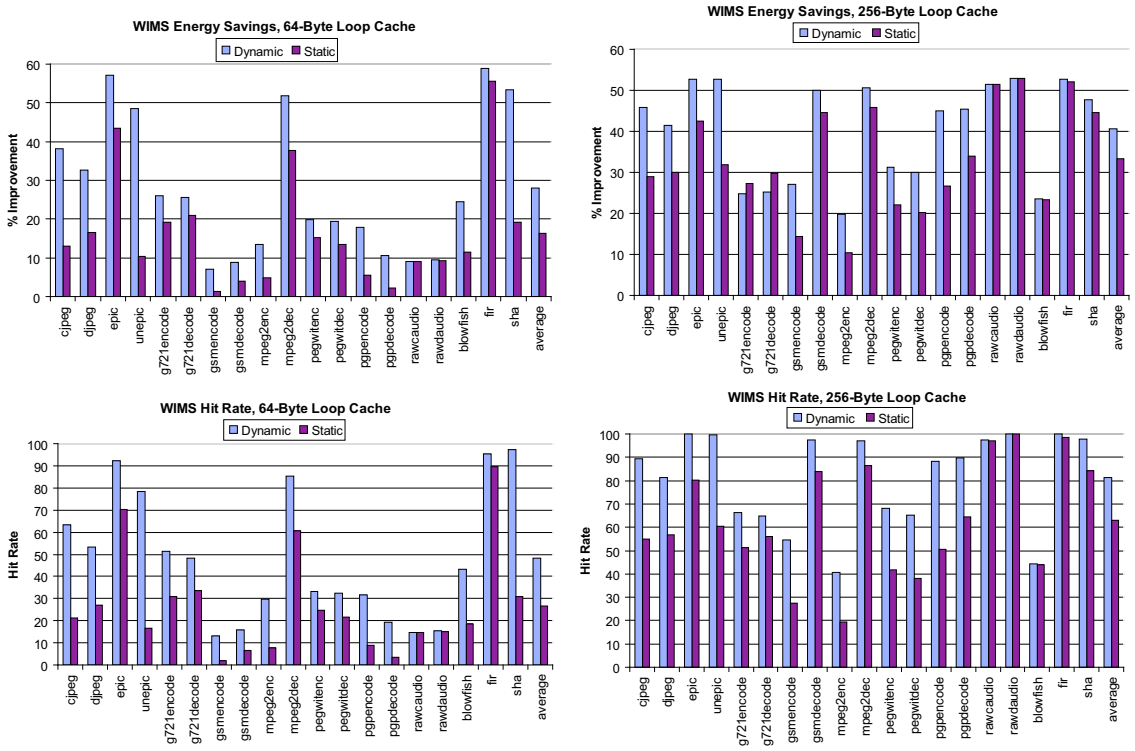


Figure 4.7: Comparing energy savings and hit rate of static and dynamic over on-chip main memory for the WIMS processor.

sha where the static placement savings are more than doubled with dynamic placement. These benchmarks achieve such large gains by increasing the hit rates in the SP by similar amounts due to more effective utilization of the SP. The epic application achieves the largest total energy savings of 58% with dynamic placement. However, static placement is also very successful with this benchmark, achieving 42%. Epic has a relatively small innermost loop where a large fraction of the execution time is spent, and the entire loop body can be placed in the 64-byte SP. Dynamic achieves a modest gain above that by relocating another loop body into the SP. Overall, dynamic placement achieves an average energy savings of 28% across the benchmarks compared with 17% for static placement.

Examining the 256-byte SP graphs, the energy savings and hit rates achieved with

static and dynamic placement are much closer. Clearly, as the SP size is increased, the importance of dynamic placement goes down as a larger fraction of the hot regions statically fit into the SP. Cjpeg, djpeg, unepic, gsmencode, mpeg2enc, and pgpencode are examples where dynamic is still very effective as these benchmarks have a large memory footprint. A small fraction of benchmarks, where the energy savings with dynamic placement exceeded static for the 64-byte SP, now achieve worse results with the 256-byte SP. Examples of this behavior are g721encode, g721decode, rawcaudio, and rawdaudio. These benchmarks are characterized by a modest number of conflicts between SP entries. The copies could not be hoisted out of frequently executed code regions due to interference, thus a large number of run-time copies must be performed. Rawcaudio and rawdaudio have small code size; thus, static is able to pack all the hot regions in the application into the SP without any run-time penalty. Dynamic, on the other hand, achieves the same hit-rate but at the expense of the one-time copy overhead. Overall, for both 64 and 256 byte SP configurations, by packing multiple hot regions, dynamic is effective at increasing SP hit rates.

The effect of varying SP size on four representative benchmarks on the WIMS processor is shown in Figure 4.8. Each graph contains 4 lines: SP hit rate for dynamic, SP hit rate for static, energy savings for dynamic, and energy savings for static. Note that hit rates (shaded light) use the left hand y-axis and energy (shaded dark) use the right hand y-axis. A number of interesting trends can be observed from these graphs. First, at smaller SP sizes, dynamic placement outperforms static placement by a large margin. For mpeg2dec, dynamic placement increases energy savings from 35% to 75% for a 32-byte SP and from 60% to 80% for a 64-byte SP. Similarly, for pgpdecode, energy savings increases from

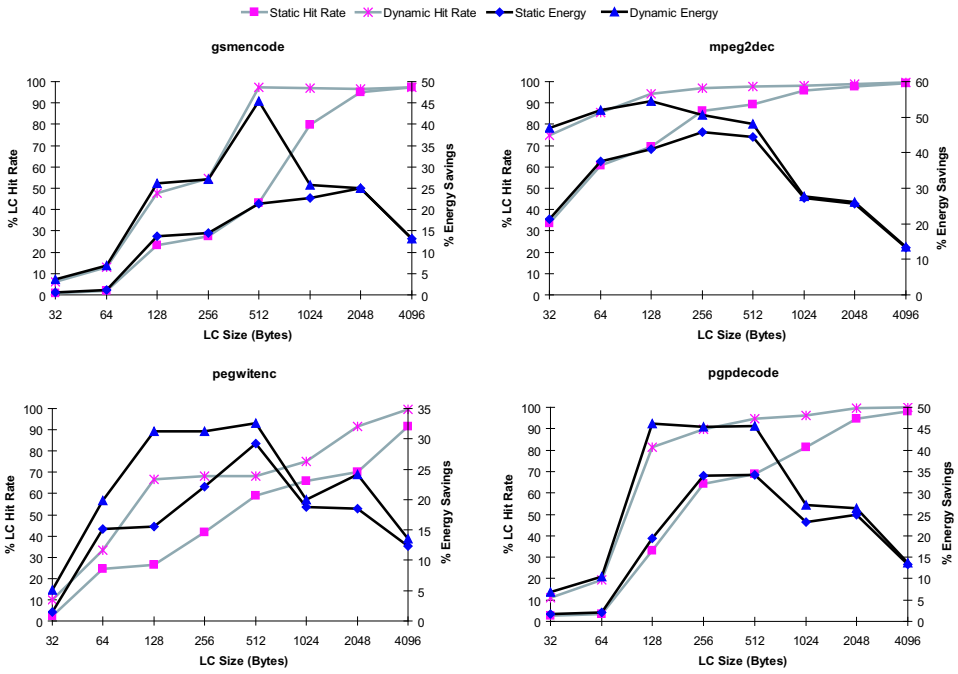


Figure 4.8: Effect of varying scratch-pad size on energy savings and hit rate over on-chip main memory for the WIMS processor.

38% to 92% for a 128-byte SP. For the other benchmarks, the differences are not as large, but the same trend occurs. The reason for the increased energy savings is the ability of dynamic placement to increase SP utilization. Most of these applications contain a number of hot code regions that collectively cannot fit in the SP using static placement. It is thus critical to relocate different regions of code into the SP at different points during program execution to take full advantage of the SP. The increased utilization is evident by the large increase in hit rate of dynamic over static for the smaller SP sizes.

A second trend seen in all the graphs is that energy savings goes down for larger SP sizes, particularly the 1k, 2k, and 4k configurations. The peak energy savings comes at around 128-512 byte SPs. The reason for this behavior is two fold. First, it becomes less

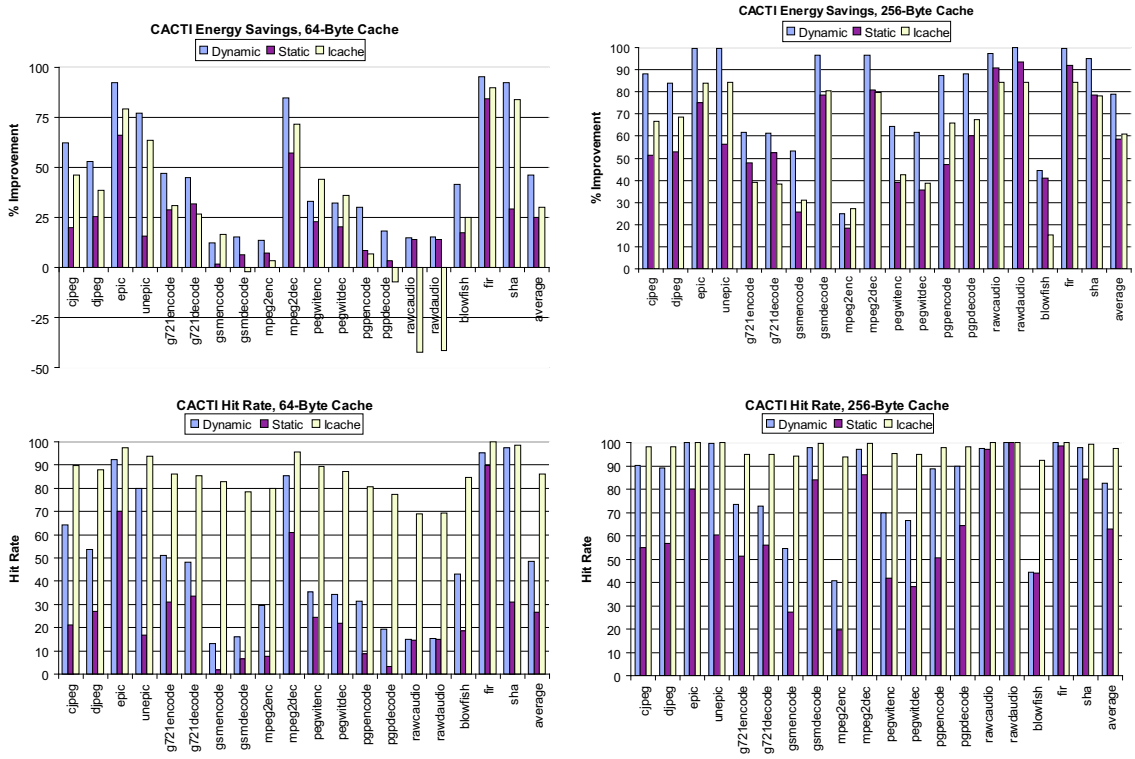


Figure 4.9: Comparing energy savings and hit rate of static, dynamic, and icache over off-chip main memory using CACTI.

beneficial to relocate instructions with larger SPs. For larger SPs, the energy characteristics are close to that of the on-chip memory, thus the potential savings becomes less. Second, the overhead of dynamic copying becomes larger, thereby taking away from the percentage savings. For the larger SP sizes, static performs a good job of packing a significant fraction of the hot code without any overhead. For dynamic, copy overhead causes the energy savings to depreciate.

Comparison with icache: Figure 4.9 shows the energy and hit rates for static, dynamic, and icache for 64 and 256-byte on-chip cache configurations using CACTI energy models. The main memory is assumed to be off-chip. The average energy savings for both static and dynamic are much higher than the WIMS processor. For the 256 byte SP, static

achieves 59%, while dynamic achieves 79% average energy savings. This is largely due to the costlier off-chip memory access. The off-chip main memory is over 20x more power hungry than the on-chip memory.

For all the benchmarks, icache is able to get higher hit-rates compared to static and dynamic schemes. On average, we observe 98%, 82%, and 62% hit-rate for icache, dynamic, and static respectively in the 256-byte cache configuration. In the 64-byte case, icache records a 70% improvement in hit-rate over dynamic. But this improvement comes at the expense of energy. Each access to the icache requires tag checks and hence is costlier than the tag-less SP. In addition, a miss for icache is much more expensive, as it has to fetch a cache-line of instructions (16-bytes) into the icache every time, which involves multiple accesses to both the main memory and the icache.

The dynamic scheme is geared towards reducing the overall energy as opposed to raw hit-rate. The dynamic scheme copies a flexible number of chunks using a LC-COPY only when deemed beneficial by the compiler, thus leading to a more energy efficient SP utilization. A miss requires only a single access from the main memory. Dynamic is able to achieve 66% and 33% improvement in energy savings over icache for the 64-byte and 256-byte SP configurations respectively. In the 64-byte case, for rawcaudio, rawdaudio, pgpdecode, and gsmdecode, although icache registers over 70% hit rate, there is a decrease in energy savings over main memory. Overall, icache performs better than static, while dynamic performs better than the two.

Detailed comparisons: Table 4.2 (left) compares the code size and the size of the cache required for static, dynamic, and icache to cover 95% of the dynamically executed

benchmark	size(Kb)	static (b)	dynamic (b)	icache (b)
cjpeg	63	4096	1024	256
djpeg	68	2048	512	256
epic	11	1024	128	64
unepic	13	1024	128	128
g721encode	4	2048	2048	256
g721decode	4	2048	2048	512
gsmencode	21	2048	512	512
gsmdecode	19	1024	256	128
mpeg2enc	35	4096	1024	512
mpeg2dec	24	1024	256	64
pegwitenc	21	8192	4096	256
pegwitdec	21	4096	2048	512
pgpencode	102	4096	512	256
pgpdecode	102	4096	512	256
rawcaudio	.8	256	256	256
rawdaudio	.8	256	256	256
blowfi sh	5	1024	1024	1024
fi r	.5	256	64	64
sha	1	512	64	64
average	27.16	2277.05	882.53	296.42

benchmark	static	dynamic
cjpeg	512	256
djpeg	512	512
epic	512	128
unepic	512	128
g721encode	512	512
g721decode	512	512
gsmencode	2048	512
gsmdecode	512	128
mpeg2enc	512	512
mpeg2dec	256	128
pegwitenc	512	512
pegwitdec	512	512
pgpencode	512	128
pgpdecode	512	128
rawcaudio	256	256
rawdaudio	256	256
blowfi sh	1024	1024
fi r	64	64
sha	512	64
average	502.86	298.87

Table 4.2: The left table shows the code cache size (in bytes) for at least 95% hit rate for static, dynamic, and icache schemes for the CACTI energy models. The right table shows the scratch-pad size required for highest energy gains in the WIMS processor.

code for each benchmark. On average, all cache configurations require less than 10x the code size, which verifies the 90-10 rule. More interestingly, dynamic requires 2.5x less cache size than static, while icache, with a higher hit rate, requires 7.6x less cache than static. Table 4.2 (right) shows the SP size for maximum energy savings on the WIMS processor for each benchmark. On average, the SP size of dynamic is 1.68x less than the static scheme. Dynamic consumes more cache space than icache as it tries to reduce the overlaps of traces in the SP so as to decrease the recopy overhead. Table 4.3 give the size of the SP for maximum energy gains of dynamic over static. On average, at 110 bytes, dynamic shows over 17% improvement over static.

Finally, Table 4.4 quantifies the SP access behavior of the dynamic allocation scheme for a 256-byte SP. For each benchmark, column 2 gives the total number of traces in the code that were allocated to the SP. Column 3 shows the same metric expressed as a percentage of all the traces in the code. For example, for pgpencode, 155 traces were packed

benchmark	size	% gain
cjpeg	64	25.13
djpeg	64	16.02
epic	128	14.26
unepic	64	38.17
g721encode	32	12.46
g721decode	32	10.83
gsmencode	512	24.06
gsmdecode	128	9.35
mpeg2enc	128	12.19
mpeg2dec	32	25.79
pegwitenc	128	15.68
pegwitdec	128	15.35
pgpencode	128	30.89
pgpdecode	128	26.79
rawcaudio	128	0.10
rawdaudio	64	0.06
blowfi sh	64	13.03
fi r	32	19.18
sha	64	34.26
average	110.22	17.69

Table 4.3: The scratch-pad size at maximum energy gain of dynamic over static.

into the SP, which accounted for only 2.2% of all the traces in the code. Although not shown in the table, the 155 traces had a combined size of 5186 bytes, which is 20x the size of the 256-byte SP, but 5% of the total code size. Column 4 gives the dynamic execution frequency for these selected static traces. For pgpencode, 2.2% of the hot traces accounted for 87% of the dynamic execution frequency. Column 5 gives the mean number of trace overlaps per SP chunk. Note that a trace can overlap multiple chunks. For a 256-byte SP, there are 16 chunks assuming 16-bytes per chunk. Again for pgpencode, an average of 24.5 traces were overlapped per chunk. The dynamic scheme was able to successfully pack the most frequent hot traces and overlap them with the least number of conflicts. Finally, column 6 gives the percent overhead in cycles due to dynamic copying. Although the dynamic placement algorithm was driven by energy constraints, the performance degradation

benchmark	# traces	% traces	% dynamic. freq.	overlaps	% perf loss
cjpeg	79	1.65	89.72	13	-2.11
djpeg	58	1.11	80.36	8.81	-2.17
epic	60	7.25	99.74	8.12	-1.11
unepic	68	6.58	99.89	11.31	-1.06
g721encode	9	3.11	63.11	1.69	-8.55
g721decode	7	2.35	61.85	1.44	-7.56
gsmencode	45	3.66	58.88	9.38	-2.31
gsmdecode	14	1.04	97.58	3.31	-2.19
mpeg2enc	244	13.62	41.81	31.25	-2.39
mpeg2dec	51	3.16	96.32	7.19	-1.56
pegwitenc	42	4.65	72.15	7.31	-4.68
pegwitdec	45	4.89	69.42	8.31	-4.37
pgpencode	155	2.20	87.13	24.5	-2.21
pgpdecode	122	1.75	89.02	20.19	-2.58
rawcaudio	8	10.13	97.77	1.06	-0.01
rawdaudio	8	10.39	99.96	1.00	-0.01
blowfi sh	10	10.31	50.14	1.62	-1.05
fi r	11	33.33	99.84	1.31	-1.17
sha	9	15.25	98.24	1.44	-4.21
average	55.00	7.18	81.73	8.54	-2.57

Table 4.4: Benchmark characteristics on a 256-byte scratch-pad for dynamic allocation showing number of hot traces selected, fraction of the total number of traces, dynamic execution frequency, average number of trace overlaps per chunk in scratch-pad, and percent performance degradation due to copy overhead.

is only marginal, with an average of 2.57% loss due to copy stalls.

The dynamic placement algorithm was based on profiling the applications on a sample input. Although our final statistics were compiled using the same input, we did validate, by running on different input sets, that the control flow behavior of the benchmarks did not change significantly. The resulting performance/energy changed by less than 1%.

4.4.3 Comparison to ILP-based Solutions

While [87, 103] address compiler-directed dynamic placement, we believe the proposed ILP-formulation for optimal placement is impractical for moderate to large sized

applications. The ILP-solution attempts to model the problem by having variables for all possible traces at every edge on the CFG. Each such variable denotes whether the trace is placed in the SP or memory, and whether it needs to be recopied or not. The problem is formulated based on an earlier work on ILP-based optimal register allocation [6]. This works well as long as the number of constraints and variables are limited, but explodes when attempted for full inter-procedural analysis of large programs. Table 4.5 reflects our implementation of [103] and shows the number of variables and constraints, the time taken to solve for inter-procedural placement using a commercial ILP-solver, CPLEX, and the percentage energy improvement over the dynamic scheme. Such long run-times for a large number of variables have been acknowledged by the authors in their paper [103]. [87] and [103] try to reduce this complexity by limiting their analysis to within a procedure or loop boundaries. But this causes the SP contents to be flushed after procedure calls/returns. We observed excessive redundant copies to restore the SP contents without inter-procedural analysis.

Our heuristic based solution offers an alternative approach to ILP that does not achieve optimal results, but is practical and can handle full inter-procedural analysis of large programs with arbitrary control flow. For smaller loop-dominated benchmarks with multiple loop-nests and simple control-flow, the dynamic scheme was able to select and overlap an optimal set of traces. The iterative copy hoisting algorithm was successful in positioning the copies at optimal points in the code. As seen in Table 4.5, for all benchmarks the dynamic scheme performed close to optimal. The degradation for dynamic was observed due to the greedy nature of the trace placement and copy hoisting heuristics.

benchmark	# variables	# equations	time	% gain
fir	2034	2446	1 min	0
rawcaudio	1980	2516	1 min	2
rawdaudio	1404	1778	1 min	0
g721encode	2927	2584	1 min	11
g721decode	3238	2950	1 min	4
blowfish	8528	10169	1 min	4
sha	13598	14773	3 min	2
gsmencode	148538	142259	20 min	6
gsmdecode	150772	143394	20 min	2
epic	420170	515835	1 hr	1
unepic	227852	298993	1 hr	1
cjpeg	1210512	1496334	40 hr	3
djpeg	933002	1149342	40 hr	3
pegwitenc	1732012	2093478	60 hr	4
pegwitdec	1565644	1904755	60 hr	4
mpeg2enc	7666254	9850505	*	*
mpeg2dec	3534664	4382525	*	*
pgpencode	12673300	16033319	*	*
pgpdecode	10404038	13488579	*	*

Table 4.5: Size, time taken, and percent energy gain over the dynamic scheme for a full program ILP-formulation using CPLEX on a 1-GHz UltraSPARC-IIIi processor. '*' denotes failure to complete within 72 hours of run-time.

4.5 Conclusion

In this chapter, an approach for compiler-directed dynamic placement of instructions into a low-power code cache was proposed. Dynamic placement enables the compiler to use entries in the code cache to hold multiple hot code regions (traces) over the execution of an application, thereby increasing the code cache utilization by a substantial amount. These traces can have any complex control flow or embedded procedure calls. Dynamic placement is accomplished in two major steps. First, the code cache entries are allocated among all the candidate traces. A heuristic cost/benefit analysis compares the expected copy cost with the anticipated energy benefit. Second, copies are inserted followed by

an iterative process of hoisting and redundancy elimination using liveness analysis on a inter-procedural control flow graph to derive a cost-effective placement. Our investigation was carried out in the context of the WIMS (see Chapter II) microcontroller, a processor designed for embedded sensor systems where power consumption is the dominant design concern. Results show an average energy savings of 28% with dynamic placement compared with 17% with static for a 64-byte code cache. This is accomplished by increasing the code cache hit rate from an average of 26% to 49%. For a 256-byte code cache, a more modest increase in energy savings occurs; 41% with dynamic verses 32% with static. In comparison to a traditional instruction cache, dynamic placement achieves an average of 25% energy savings.

CHAPTER V

Compiler Managed Partitioned Data Cache Architecture

5.1 Introduction

Caches have been highly successful in bridging the processor-memory performance gap by providing fast access to frequently used data. They also save power by limiting expensive off-chip accesses. Data caches have proven to be effective as they help to dynamically capture both temporal and spatial locality without software intervention.

However, the use of caches in embedded domains has been limited due to energy inefficient tag checking and comparison logic [8]. Set-associative caches can achieve a high hit-rate and good performance, but it comes at the expense of energy overhead. Direct-mapped caches remove much of the logic overhead and thus consume much less power per access, but they incur more misses.

In this chapter, a hardware/software co-managed partitioned cache architecture is proposed that attempts to bridge the performance and energy gap between direct and set-associative caches. A partitioned cache consisting of multiple smaller direct-mapped partitions with the same combined size as a unified direct or set-associative cache is employed. Management of these partitions is controlled by the compiler using load/store instruction

set extensions. Using a whole program knowledge of the data access patterns, the compiler controls cache lookup and data placement by assigning individual load/store instructions to these partitions.

This software-guided partitioned cache architecture has many advantages. First, a smaller direct-mapped cache is more power efficient than either a unified direct-mapped or a set-associative cache, as the data and tag arrays are smaller. The software decides what partitions are activated, thus eliminating redundant tag/data array accesses to reduce power. Second, by managing the placement of data using the memory reference instructions, the compiler can enforce a better replacement policy. For example, data items that are accessed with a high degree of temporal locality can be placed in different partitions so as to avoid conflicts. But, references that are separated in time or whose live-ranges do not intersect can be made to share the same partition. Thus, this data orchestration can help reduce conflict misses.

Region-based caches [55] have been proposed, where multiple caches are used to capture heap, global, and stack accesses. But, unlike their approach, partitioned caches provide much finer grain data management and control. Instruction-level management generalizes to all types of data for all classes of applications, including heap dominated ones, where distinct heap objects can be placed in different partitions. Thus, through compiler-controlled management, the partitioned cache architecture can achieve the high performance of a set-associative cache while being within the energy envelope of a direct-mapped cache.

Further energy reduction is achieved by allowing each of the partitions to be configured

as a software-controlled scratch-pad. The data-arrays are exposed as part of the physical address space. By disabling the tag-arrays of selected partitions, a highly configurable data memory that can be tuned to the application's memory needs is provided.

5.2 Background

In this section, the need for hardware/software co-managed cache is motivated along with a discussion of how software can exploit partitioned caches for finer grained cache management.

Hardware/Software Co-Managed Caches: In a traditional cache design, the hardware is used to determine replacement and allocation policies. A standard hardware cache controller has two main responsibilities: 1) checking if the referenced data is present in the cache, and 2) on a miss, deciding where in the cache to allocate the requested data. Performing checks in hardware allows for fast and efficient access of the referenced data. It provides the appearance of a uniform address space by hiding the details of the underlying cache architecture from the programmer. The tags help in dynamically locating the cached memory references that can be hard to analyze statically.

However, making decisions in hardware usually forces a single, conservative allocation and replacement algorithm. Implementing a more flexible replacement policy entirely in hardware is usually expensive. Hardware-based schemes typically place data using the set-index field extracted from the address. This does not consider the access behavior pattern of the referenced data. For example, two data items referenced temporally adjacent to each other can be placed in the same set, thus causing conflicts. To reduce the effect of conflicts, set-associativity is employed such that conflicting data elements are placed

in different ways. But here again, a simplistic replacement policy, such as pseudo-LRU, is used which does not guarantee the absence of conflicts. Moreover, for set-associative caches, on each reference, every way within the set has to be probed to check for the presence of the data. Different applications or separate phases within the same application can have widely varying associativity requirements depending on their locality and working-set characteristics. Thus, many of these tag checks can be redundant resulting in wastage of power at no added performance advantage.

Making replacement and allocation decisions in the compiler can offer several benefits, as it can employ more intelligent heuristics to make replacement decisions at considerably lower costs. Also, the compiler has the added advantage of analyzing the application's future behavior through profiling of the application on a representative input set. Software control can thus customize the cache accesses based on the needs of the application. Software-only approaches like code/data re-organization [18, 76] can help avoid conflicts. But, the hardware, being transparent, is unaware of these transformations and thus performs wasted tag checks.

We employ a hardware/software co-managed caching policy where the hardware performs the critical cache lookup to reduce access cycle time, while providing hints through the software to guide the hardware towards efficient management. Exposing the operation of the cache to the software/compiler facilitates efficient use of this critical storage for both power and performance.

Partitioned Caches: Cache partitioning allows for coarser grained management by the software as compared to hardware-based replacement, while delegating critical finer

granularity operations, such as tag checks, to the hardware. Traditionally, caches are logically organized into ways, where the tag corresponding to each of the ways are compared in parallel to reduce access cycle time. The data is read from the matching way. Prior to matching, the parallel access has to pre-charge and read all the tag and data arrays, but select only one of the ways[1], resulting in wasted dynamic energy [66]. Ideally, in an n-way set-associative cache, using an oracle predictor, only one of the n-ways has to be read and the rest can be ignored. For a 4-way cache, accesses to three of four ways are redundant. Using CACTI [75], for a 1-Kb 4-way set-associative cache, an oracle predictor can save almost 64% of the cache access energy.

In vertically partitioned [106] or way-partitioned caches [19], each way is treated as a separate partition. Prior work uses vertical partitions for either energy savings through hardware control [106] or as coarser grained units that are managed as scratch-pads [19]. We generalize this idea by allowing the compiler finer grain control over the individual ways. In particular, on a cache access, the compiler decides the partitions or ways to be probed for the referenced data. Similarly, cache replacement is restricted to certain ways so as to reduce conflict misses. Although multiple ways can reduce conflicts, pseudo-LRU replacement algorithms are non-optimal, and thus cannot guarantee correct decisions. By pro-actively placing temporally co-located data references in different partitions, the compiler can avoid conflicts. Similarly, references that have poor temporal locality can be restricted to a single partition or even be allowed to bypass the cache by not assigning them to any of the partitions, thus preventing cache pollution. Smaller L1 caches with

[1]L1 caches generally employ parallel access to ensure the fastest access time. Lower levels of the memory hierarchy, i.e., L2/L3 caches, often serialize tag and data access to reduce unnecessary energy consumption as access latency is less important.

low associativity, high degree of conflict misses, and requiring a single cycle access are ideal candidates for way-based partitioning. It should be noted that partitioning need not be restricted to individual ways. It can logically be extended to include multiple ways per partition.

More importantly, in addition to reducing conflict misses, way- partitioned software-managed caches can restrict cache lookups to only selected ways that are guaranteed to contain the data. This provides a two-fold advantage. First, restricting memory references, based on its individual memory needs, to only a subset of the available cache can help save energy as only the assigned set of tags and their corresponding data arrays are activated. Second, per-access cycle time can be improved since only a limited set of ways are read on a single access. Ideally, the compiler can restrict each access to just a single way, thus matching the energy savings of an oracle predictor.

Way partitioning provides an ideal platform for the compiler to exercise fine-grained management of data placement at a particular level of the cache hierarchy by reducing conflicts while lowering the energy consumed. This work focuses on L1 caches, although the technique can be generalized to other levels of the cache hierarchy. Single level caches are common in embedded domains, where energy is a primary design constraint. We focus on software-based way-partitioning of L1 caches as they are typically small and capture majority of the memory references.

5.3 Partitioned Cache Architecture

In this section, we look at the architecture and the ISA extensions required for a software-managed way-partitioned cache. The cache controller is altered to allow for a

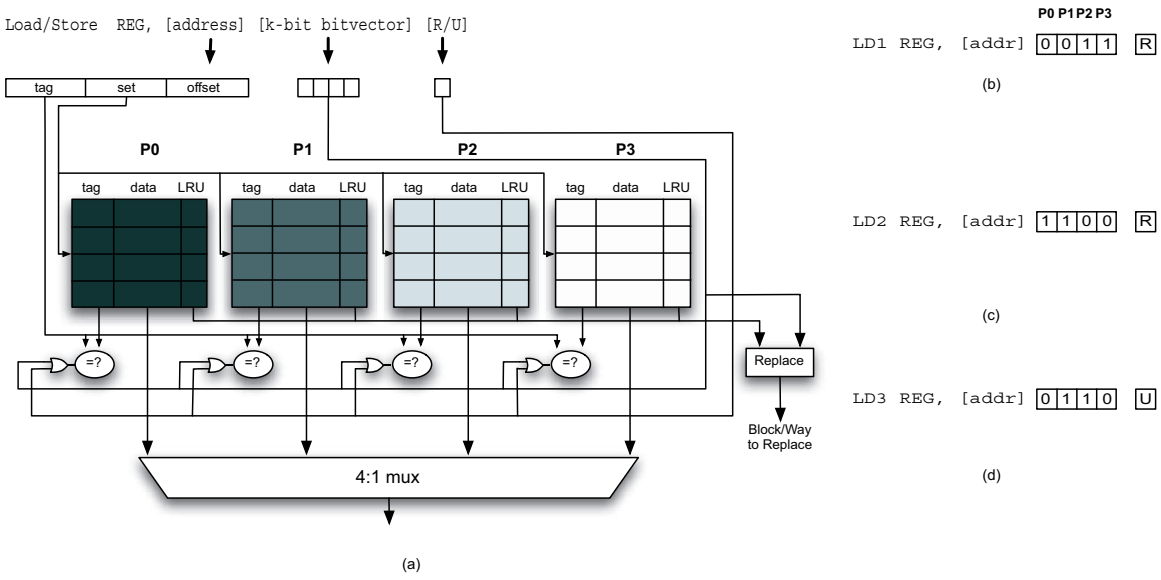


Figure 5.1: Hardware/software co-managed vertically partitioned cache

specified subset of the cache ways to be activated on a lookup. In addition, on a cache miss, the cache can be directed to only use a subset of the ways during replacement. Modified load/store instructions are used by the compiler to control the operations of the cache.

For a software-managed way-partitioned data cache, an important factor in determining the cache design is the granularity of partition assignment. Ideally, partition assignment should be made at the cache block level, as replacement decisions are made for each individual cache block. However, since memory blocks that a program accesses can vary for every run of the program, we instead assign partitions to the load or store instructions that access these blocks. By guiding the memory instructions, data placement in the cache is indirectly controlled. In order to guide load/store operations to their designated cache ways, we extended the instruction set architecture to include partition designations.

Figure 5.1 shows a 4-way set-associative cache. Each way is treated as a software-controlled cache partition. As shown on top of Figure 5.1, we conceptually extend the

load/store instructions with additional operand fields. For a *k*-partitioned cache, a *k* bit-vector immediate field is used to denote the partition (i.e., way) to which the instruction is assigned. For example, in Figure 5.1(b), *LD 1* is assigned to partitions 2 and 3. Similarly, *LD 2* in Figure 5.1(c) is assigned to partitions 0 and 1, and *LD 3* in Figure 5.1(d) is assigned to partitions 1 and 2.

Cache Replacement: On a miss in the cache, a block within the cache must be selected for replacement. On a miss, only the ways specified by this replacement bit-vector are considered for replacement. This allows the compiler control over the replacement decisions among the ways in a set. If an instruction is assigned multiple partitions, LRU or random policy can be used among the specified partitions. All load/store instructions need not be annotated with the partition bit-vector. For regular load/store instructions that are unannotated, all the partitions are considered for replacement. Thus, the flexibility of the underlying hardware allows the compiler to treat individual loads as needing only a single way up to all the ways of the cache, based on its access characteristics.

Cache Lookup: On a cache access, all ways that could possibly include the cache block must be probed. This is required to avoid any coherency and duplication of cache blocks. If two memory references sharing the same set of data objects are placed in different partitions, then all such partitions have to be checked for the presence of the data. Otherwise, one of the references, on a miss in its assigned partition, will not only duplicate the block cached in an another partition, but also read/write a stale value. The lookup can conceptually be accomplished by adding a second bit-vector for lookup to each instruction, which corresponds to all the partitions that could hold the referenced data object. During

lookup, only the specified ways have to be probed and the other ways need not have their tag/data arrays activated. Separating replacement from lookup provides the combined flexibility of improving performance by controlling data placement, while saving power by avoiding redundant tag checks.

While the lookup bit-vector would accomplish the task, its task can easily be folded into the original replacement bit-vector and a single extra bit-field. This bit-field, called the R/U-(**R**estricted/**U**nrestricted) bit, is added to each load/store instruction and shown in Figure 5.1. This field is used to restrict the tag checks to the partitions specified in the bit-vector. *R* means that only the specified partitions in the replacement bit-vector need be probed, while *U* forces all the partitions to be checked. Although a unified lookup/replacement bit-vector is less general than having a separate lookup bit-vector, it has been found to provide comparable benefits at reduced costs.

This lookup-optimization requires the compiler to guarantee that two memory instructions assigned to different partitions with the R-bit will never access the same data. More details on how this is done is described in Section 5.4. In the example shown in Figure 5.1, *LD 1* and *LD 2* are assigned to two different sets of partitions with their R-bit set. This restricts them to activate only their assigned ways. Ideally, we could have a set-associative cache, where each data object is assigned to different ways such that an access only activates the assigned way. This would enable better per-access energy savings than a direct-mapped cache while retaining the miss-rate of a set-associative cache as each of the assigned data objects do not conflict with each other.

ISA Support: The ISA extensions as described above require adding more encoding

bits, which might not be practical for embedded processors where code size is a primary design constraint. This overhead can be reduced by using a special purpose register called the cache access register (CAR) to hold the partition bit-vector. This allows extending the design to an arbitrary number of partitions. Further, we introduce two different types of load/store instructions. One that is partition cognizant, while the other that is not. The partition aware instruction implicitly sources the CAR. Extra move instructions are required to initialize the registers with a literal corresponding to the respective partition bit-vector and the R/U-bit. Partition unaware instructions do not use the CAR and perform lookup and replacement on all the ways just as in traditional designs. As shown in our experiments, the resulting code-size overhead is only 0.4%.

Besides the modifications to the cache controller to honor the partitions specified by the current instruction, no additional hardware beyond a standard cache is necessary. The tag-directory structure is retained. If partitioning is not supported, the assignment can be ignored and the default set-associative scheme can be used. For future generation processors that could be designed with higher or lower associativity, the compiler specified partitions can be virtualized by allowing multiple specified partitions to refer to a single way or a single specified partition to refer to multiple ways.

Memory Partitions as Scratch-pads: The partitions used in our architecture are not limited to a cache-based design. Partitions can be used within embedded processors as customizable scratch-pad (SP) memories. SPs are local on-chip SRAMs without the complex tag-checking logic of caches [8]. They can be superior to caches in terms of energy and provides real-time guarantees. Given this explicit instruction based control

of the individual ways, the compiler can optionally convert some of these ways to SPs by deactivating the corresponding tag-array [19]. This is useful in modern embedded processors, such as ARM, which have highly associative L1 caches. However, such high associativity is not always required and hence, some of these ways can be configured as software managed SPs. In fact, the Analog Devices DSP processor allows caches to be selectively re-configured as SPs. This helps to tune the memory system to the needs of the application. In Section 5.4, we provide more details on how way-partitioning can be used to decide how to split the ways between SPs and caches for each application to reduce the overall energy.

The objective of this work is to balance two opposing goals. On the one hand, cache access energy can be reduced by restricting memory references to as few ways as possible. Energy is reduced by limiting the number of tag and data arrays that are activated during each access. Moreover, controlled placement can help avoid conflict misses, thus eliminating off-chip accesses. However, this can potentially lead to capacity misses for references that have a moderate to large working-sets. The working-set of such references are retained by allowing them to perform data placement across multiple ways. Therefore, the compiler must balance these trade-offs so as to reduce conflicts while minimizing energy. Compiler managed way-partitioning provides better management over traditional cache designs by allowing customization of cache accesses based on the memory access requirements of the application. The following section describes the compiler algorithm to automatically assign partitions to the load/store instructions.

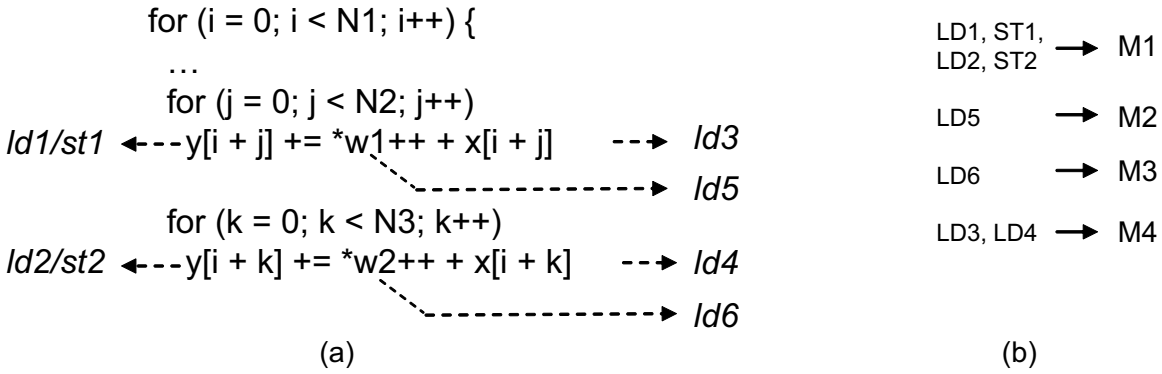


Figure 5.2: (a) Example kernel to illustrate compiler-managed cache partitioning. (b) Fused load/store instructions.

5.4 Compiler Partitioning of Memory Instructions

Software-managed cache partitioning gives the programmer or the compiler explicit control to manage the different partitions based on the memory needs of the application. In this section, we describe a compiler heuristic that automatically analyzes an application and assigns the important load/store instructions within the program to different cache partitions. The goal of the compiler is to assign these memory reference instructions to partitions (i.e., ways of the cache) so as to (1) reduce conflicts among data objects that are simultaneously accessed, and (2) restrict the instructions to just the appropriate number of partitions so as to satisfy the working-set of that reference. The first goal improves cache utilization by eliminating conflict misses that arise due to interferences among temporally co-located data reference streams. The second goal tries to reduce the number of redundant tag/data array checks that would otherwise have to be performed by the hardware. These goals provide the combined benefit of achieving the performance advantage of set-associative caches while reducing the energy consumed.

Compiler-managed partition assignment consists of two phases: cache estimation and

assignment. The cache estimation phase analyzes the memory access and usage characteristics of the application to estimate the cache configuration requirement of individual loads and stores. In addition, it also computes the degree of temporal interference among each of the loads/stores. In the assignment phase, a greedy heuristic is employed to assign these instructions to partitions. We employ a combined static and profile-driven compiler analysis. Pointer-analysis is used to control cache lookups for correctness, while memory address profiling is used to guide partition assignment. Profile-based analysis is more accurate in the presence of dynamically allocated data structures. Alternately, affine analysis [90] over array indices can be used to compute the memory reuse patterns and conflicts of data objects.

Figure 5.2(a) shows a sample C code segment with nested loops. This example is used throughout this section to illustrate the partitioning process. The load and store instructions that correspond to the accessed arrays are labeled in the source. Between the two innermost loops (with indices *j* and *k*), the arrays *y* and *x* are reused, while accesses to the *w* objects (*w1* and *w2*) are distinct. During execution of each innermost loop, the accesses to *y*, *x*, and the *w* objects are temporally co-located. So to prevent cache misses, each of these data objects have to be assigned such that they do not overlap with each other. Also, during the execution of the second loop, *w2* should not displace either *y* or *x*. But since *w1* and *w2* are distinct, they have no reuse and hence can overlap in the cache.

5.4.1 Cache Estimation

In order to assign partitions to the load/store instructions, the compiler first analyzes the application to identify three key attributes - (i) the data sharing pattern among differ-

ent load/store instructions, (ii) the cache configuration required to satisfy the working-set requirement of each load/store instruction, and (iii) the degree of conflict between every pair of load/store instructions.

Multiple instructions could share access to the same set of data structures. Grouping such references helps to restrict the number of partitions to which they are assigned. In addition, it guarantees that instructions that access the same set of data objects have the same partition assignment. The working-set size estimates how much cache should be allotted to the instruction, while the degree of conflict guides placement of instructions to different partitions. These attributes are mined independent of the address and the organization/replacement-policy of the target cache architecture.

The compiler initially profiles the code by running the application on a train input set. The profile run generates a trace of the load/store instructions executed during the run along with the addresses referenced in units of cache blocks. This memory address profile is processed on-line by analyzing a window of references that slides over the generated profile data. The window size is bound to two times the size of the cache in units of cache blocks.

Computing Data Sharing Among Load/Stores

Capturing commonality between instructions that access the same data can help limit cache accesses to just the required set of partitions. This involves the following - (i) using points-to analysis to guarantee that instructions that could potentially access the same data set are assigned to same partitions, and (ii) pro-actively merging such instructions so as to prevent them from being assigned to different partitions. This in turn reduces the number

of partitions that are to be activated at run-time.

Points-to Analysis: To restrict the load/store instructions to access a subset of cache partitions, the compiler has to guarantee that no two references assigned to two different partitions can access the same data item. This can lead to coherency issues as discussed in Section 5.3. Pointer analysis is used to avoid this problem. The pointer analysis phase within our compiler annotates every load/store instruction with a set of object identifiers that it potentially accesses [67]. The objects can be global/stack arrays/variables/structures or heap allocated objects.

Initially, each distinct object identifier and its associated load/store instructions are placed into a separate object set. If an instruction accesses multiple data objects, they could reside in different object sets and hence the corresponding sets are merged. This process continues until a set of completely disjoint object sets are obtained. All instructions within the same object set have to be assigned to the same set of partitions. Or more conservatively, if they are assigned different partitions, their U-bit (see Section 5.3) is set so that they check all available partitions on each reference for correctness, thus negating the energy benefits. Instructions within each object set are guaranteed not to access data objects of another set and hence can be assigned the R-bit. Each such fused set of instructions are now treated as a single new instruction which is then used during memory address analysis. By fusing these aliased instructions, the later placement phase is guaranteed to make the same placement decision for all actual instructions within this fused instruction.

Heuristic Fusioning: The pointer analysis phase is usually conservative and can potentially fuse more instructions than necessary. To avoid this, we instead perform heuristic

fusing. Heuristic fusioning is only used for the subsequent assignment phase. But the actual setting of the R/U-bit is done based on the conservative pointer analysis information to guarantee correctness (see later Section 5.4.2). Pointer analysis, to ensure correctness, may potentially tag two references as 'may alias', although at run-time they have a low probability of aliasing. Conservatively grouping such references forces them to be assigned to the same set of partitions leading to conflicts. Heuristic fusioning avoids this by separating such references such that the later assignment phase can assign them to different partitions. But, based on pointer analysis, the U-bit for both these references have to be set because, in the unlikely event of an alias, all other partitions have to be probed for any previously cached data. Although this fails to reduce redundant tag checks, it can help in improving performance by pro-actively reducing conflicts. Thus, the hardware supported decoupling of replacement from lookup allows the compiler to perform aggressive optimizations without being overly conservative.

Based on the profile information, all instructions that share more than a threshold (60% is used in our experiments) of the commonly accessed cache blocks are fused. This is a less aggressive, but more accurate form of fusing, which helps to group instructions that truly alias. These heuristically fused instructions are then considered as an aggregate instruction to be used in the later phases.

An important point to note is that pointer analysis cannot detect the case when two loads access different objects that fall within the same cache block. Data padding is used to prevent such false sharing. Since most objects occupy multiples of cache lines, the overhead due to padding is minimal.

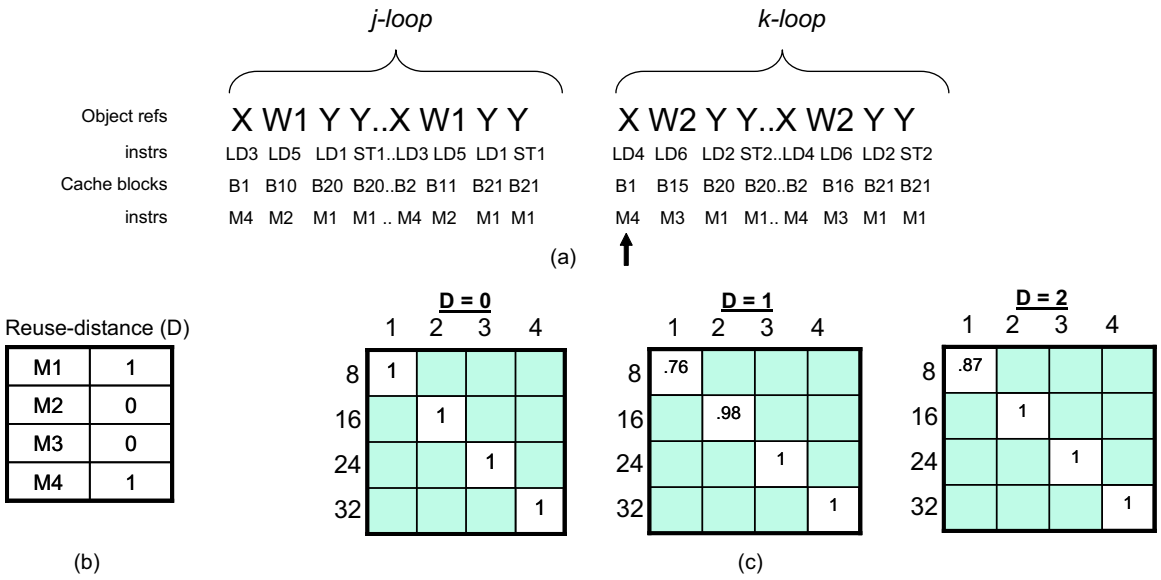


Figure 5.3: (a) Trace consisting of array references, cache blocks, and load/stores from the example in Figure 5.2. (b) Reuse distance (D) for each fused instruction. (c) Hit-rate estimate for different cache configuration using Equation 5.1.

We use the memory address profile shown in Figure 5.3(a) to illustrate the partitioning process. This is derived from the example illustrated in Figure 5.2(a). The *Bi*s represent the cache blocks that are accessed by the load/store instructions labeled in Figure 5.2(a). Along with the instructions, the corresponding data objects referenced are also shown. Note, while executing each innermost loop, the access to individual arrays can span multiple blocks. For example, access to array *x* in the *j*-loop spans blocks B1 and B2. Instructions LD3 and LD4 reference the same array *x* and so are grouped into a new fused instruction. In this example, both points-to analysis and heuristic fusioning leads to the same result and hence for ease of explanation we assume conservative fusing based on pointer analysis. The grouping of instructions is shown in Figure 5.2(b). Each fused instruction is shown as M_i and is also listed at the bottom of the memory address profile in Figure 5.3(a).

Algorithm 5.1: Estimating working-set size of load/store instructions

Input: Memory address profi le consisting of the load/store instruction and the cache block referenced $\{(L_k,B_1),(L_k,B_2),(L_k,B_3),...,(L_k,B_k)\}$
Output: Working-set size for each load/store instruction $\{L_k, Size\}$
for *(*$i = (k-1)$ **to** 1*)* **do**
    $dist$++ ;
    if *(*$B_i = B_k$*)* **then**
        break;
    end
end
if $(DistMap[L_k].lookup(dist))$ **then** $DistMap[L_k].value(dist)$++ ;
else $DistMap[L_k].insert(dist,1)$;

Estimating Cache Requirement For Load/Stores

The goal of the compiler is to allocate cache partitions to each load/store instruction so as to satisfy its working-set requirement. By assigning the right number of partitions, we eliminate capacity/conflict misses for all data accessed by that reference, while avoiding any redundant tag/data array accesses.

Working-Set Size Estimation: To estimate the number of partitions, first the working-set size of each load/store instruction, based on the concept of stack-reuse distance [62] (denoted as D), is computed. Estimating the average working-set helps decide the cache size to be allocated for every load/store instruction. The algorithm to estimate the reuse distance size is shown in Algorithm 5.1. For a specific instruction L_k, a list of cache block references listed in reference order is used. The size is estimated by looking at past references to unique cache blocks by that instruction. The memory address profile is scanned in reverse order starting from the current reference B_k. All unique references to B_i ($\neq B_k$) until the last occurrence of B_k is the number of cache blocks required (reuse distance) such that the current occurrence of B_k hits in the cache. The last B_k seen is then removed from the memory address profile, and the current B_k becomes the last occurrence for subsequent

passes. The working-set size of L_k is one more than the weighted average of such reuse distances.

Consider the example memory address profile shown in Figure 5.3(a). The current reference is shown by the arrow pointing to the fused instruction *M4* that references the block *B1*. Prior to this, *M4* references *B2* after referencing *B1* at the beginning of the memory address profile. Thus, for the current reference of *B1* to hit, *M4* requires at least two cache blocks. The reuse distance (D) for each reference M_i is shown in Figure 5.3(b).

Computing Number of Partitions For Each Load/Store: The number of partitions assigned should be such that its working-set requirement is satisfied. This helps to avoid conflict/capacity misses for the data accessed by that instruction. This requires an accurate estimation of the hit-rate of a given load/store instruction for different cache configurations based on its reuse-distance.

Given a reuse-distance *D*, cache size in terms of number of blocks *B*, and associativity *A*, the hit-rate can be approximately computed using the formula [11]:

$$\sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{D-a} \tag{5.1}$$

The reuse-distance D, can be interpreted as, for a given reference B, there is, on an average, D unique references to other blocks B_i between two unique references to B. The above formula assumes that the intervening references have an equal chance of being placed in any of the cache blocks within a set.

Using Equation 5.1, the hit probability for different cache configurations is shown in Figure 5.3(c). Each entry in the matrix represents the hit-rate for a given value of D. The rows of the matrix are the number of blocks in the cache, while the columns

represent the associativity (or the number of partitions) in the cache. Here, we are only concerned with the entries in the diagonal of the matrix. The top left entry represents a single partition (or way) with the total number of blocks same as that within a single way. As we proceed along the diagonal, the cache size is increased by adding more ways. The off-diagonal entries correspond to cases where partitions/ways can be combined to produce larger direct-mapped caches.

The number of partitions required for each load/store instruction with a given reuse distance D, is computed from the above matrices by picking the entry along the diagonal with the highest hit-rate. Although, we have shown a performance driven matrix, in reality, we use an energy matrix, where each entry corresponds to the energy consumed for that reference. The energy is computed using the formula $NumReferences * EnergyPerAccess(B,A) + (1 - HitRate) * NumReferences * EnergyPerMiss$, where *EnergyPerAccess* is a computed from CACTI [75] for a cache configuration with B blocks and A ways, while *EnergyPerMiss* is the energy required to fetch a cache line from L2 or off-chip memory on a miss. The *HitRate* is obtained from the performance matrix. Using this new energy matrix, the least energy consuming configuration is selected for each load/store instruction.

Computing Interferences between Loads/Stores

Section 5.4.1 computed the number of partitions required per load/store instruction. Since the cache is shared by multiple such instructions, it is important to place them in the cache so that they do not conflict with each other. References that overlap temporally are to be placed in different cache partitions, while references that are not simultaneously live

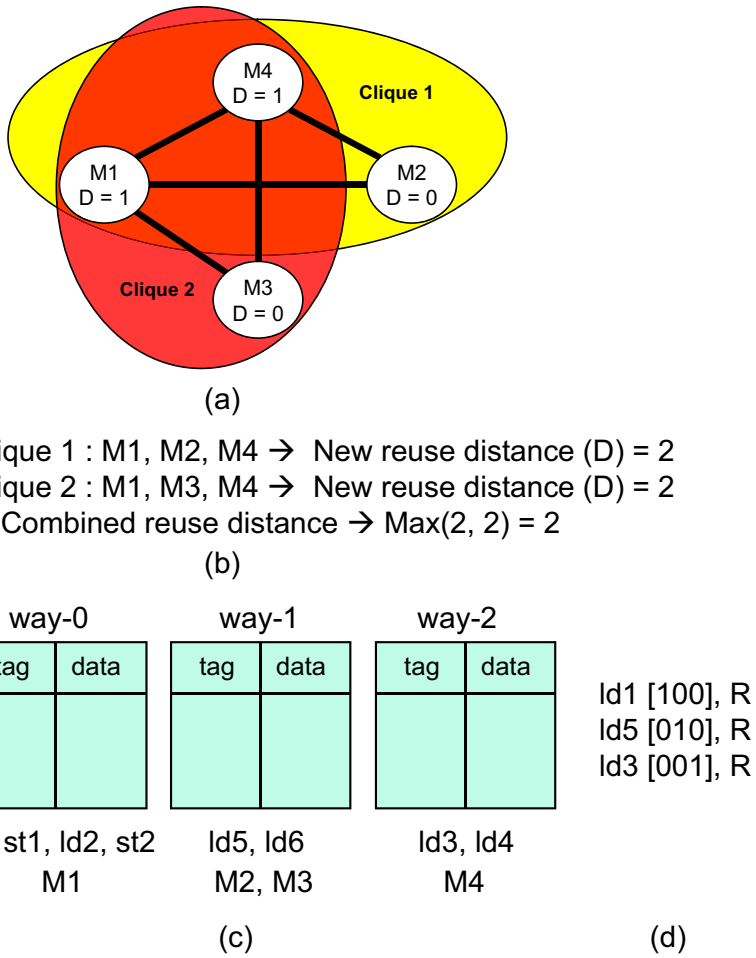


Figure 5.4: (a) Interference graph with cliques shaded. (b) The new reuse-distance for each of the cliques. (c) Assignment of loads/stores to partitions/ways. (d) Annotating with partition bit-vectors.

can share the same set of partitions.

This temporal interference between different load/store instructions is captured using a graph-based representation, aptly named the interference graph (IG). The nodes in the IG are the fused load/store instructions. An edge exists between the nodes if the degree of overlap exceeds some minimum preset threshold. To compute the interference, a single pass is made over the memory address profile. Every load/store reference that occurs between two consecutive occurrences of another load/store is recorded. The IG is shown in Figure 5.4(a) and is computed using the memory address profile shown in Figure 5.3(a).

It can be observed from the memory address profile that during the execution of the inner *j*-loop, references M1, M2, and M4 occur together temporally, while in the *k*-loop, M1, M3, and M4 overlap temporally. But references M3 and M2 occur at two different points in time and do not interfere with each other and hence do not have an edge.

Spatial Locality-Based Optimizations

So far we have tried to capture the temporal locality properties of the application through memory address profile-based analysis. The target architecture's default block size was used to capture the spatial locality for each reference. But, different applications have differing spatial locality characteristics. Thus to model spatial locality, accurate estimation of the block size of the application is required.

We saw earlier how the working-set is computed in units of cache blocks for each memory instruction. The cache configuration, derived from Equation 5.1, assumes that each block in the working-set has an equal probability of conflicting with every other block. This turns out to be overly pessimistic for references with high spatial locality. This in turn can force the reference to be assigned to multiple caches. Data references that exhibit a high degree of spatial locality can be assigned to a direct-mapped cache, as such a cache maps spatially adjacent references to adjacent cache blocks, thus avoiding conflicts. Thus, accurate estimation of block sizes can help restrict memory references to fewer partitions.

To compute the block size of the application, during a single pass of the memory address profile, we vary the block size starting from the smallest granularity, which is the block size of the target cache architecture to a maximum size, which is the size of a

single partition. For each such block size, the fraction of references that can be captured by a block of the new size is determined. The block size that maximizes this fraction is selected as the new block size. This new block size is computed prior to the earlier described phases and is used to estimate the reuse-distance, the working-set size, and the cache configuration as described above.

In Figure 5.3, using the default block size, M4 has a working-set of two cache blocks. But the two blocks that M4 accesses, B1 and B2, are spatially adjacent as they are part of the array *x* and hence will not conflict in a direct-mapped cache. By using twice the default block size, the new working-set is just a single block.

5.4.2 Cache Assignment

The goal of the assignment phase is to assign the load/store instructions to the partitions/ways in the cache. Instructions that exhibit a high degree of temporal overlap are placed in different partitions so as to prevent their data sets from conflicting.

In Section 5.4.1, we saw, based on the reuse characteristics of each load/store instruction, how the least energy consuming cache configuration was selected. If there were an infinite number of partitions, each such load/store instruction could be assigned such that its cache requirements are satisfied. But, since the number of partitions or ways are limited, some of these instructions must be assigned to the same set of partitions, thus causing them to overlap. Again, if these overlapped instructions do not interfere temporally, there would not be any conflicts. Ideally, during the assignment phase, when faced with a choice of placing a set of instructions to the same set of partitions, the memory address profile can be used to compute the combined working-set as described in Section 5.4.1. The least

number of partitions that can retain the combined working-set can then be estimated for this overlapped set of instructions. But this would involve multiple passes of the memory address profile and is clearly impractical. So we need a quick approximation of the combined working-set from just the working-set estimates of individual instructions.

To compute the combined working-set size for a set of load/store instructions, we use the interference graph computed in Section 5.4.1. If an edge exists between two nodes, they are assumed to interfere. Hence, the total number of cache blocks required to hold the working-set of both these references combined is the sum of the working-set size of each reference. But, if they do not interfere, the combined working-set size is the maximum of the working-set size of both the references.

Computing Working-Sets of Overlapped References Using Cliques: We use the graph-theoretic notion of a clique to realize the same. Given a set of potentially overlapping instructions, $M_1, M_2, ..., M_n$, we first consider a sub-graph within the interference graph consisting of just these nodes. All possible maximum cliques within this sub-graph are then enumerated. Each clique represents a set of instructions that occur together in time and hence can potentially conflict. All such instructions thus have to be assigned sufficient partitions to prevent conflicts. For each clique, the sum of the working-set size of each node in the clique is computed. The working-set size of the combined set $M_1, M_2, ..., M_n$ is therefore the max of the computed sums over all cliques. From this combined reuse-distance, Equation 5.1 can be used to find the most energy efficient cache configuration for this set of overlapped instructions.

For the interference graph shown in Figure 5.4(a), assuming that they all have to be

placed together, the cliques are enumerated in Figure 5.4(b) [2]. In this example, we assume that the block size is same as that of the target architecture and hence use the same reuse-distance for each instruction as listed in Figure 5.3(b). The reuse distance of the combined graph is two and is shown in Figure 5.4(b). Thus, if M1, M2, M3, and M4 are overlapped in the cache, atleast three partitions are required to retain their combined working-set.

Partitioning Algorithm: We use a simple greedy heuristic to place the instructions in different partitions. Each memory instruction in decreasing frequency order is placed starting from the first partition to the total number of partitions. At each candidate placement point, the most energy efficient cache configuration for that instruction is picked (Section 5.4.1). If there are previously placed instructions, the new working-set and the corresponding cache configuration is selected by enumerating all possible cliques for the set of overlapped instructions. Because of the overlaps, the number of partitions for the current instruction can be more than what would have been if only that instruction were to be considered in isolation. Among all such potential placement points, the position that results in the most energy efficient configuration is finally selected for placement for that instruction. This process is then repeated for all instructions.

In our example in Figure 5.2(a), M1 and M4 have a working-set size of one, based on the new block-size computed in Section 5.4.1. For D equal to zero, the number of partitions selected is one (from the matrix in Figure 5.3(c)). Since M1 and M4 interfere (from the IG in Figure 5.4(a)), to minimize conflicts, the greedy heuristic places them in different partitions. Alternately, if they were to be overlapped, to fit the combined working-set,

[2]Nodes that are part of a previously enumerated clique are included in a new clique as long as there is some new node that is not part of any other cliques.

both the references will have to be assigned to more than one partition, resulting in more tag checks than necessary. Similarly, M2 and M3 require a single partition and do not interfere. Their combined working-set fits within a single partition. Hence, M2 and M3 are placed in the same partition but disjointly from M1 and M4. The placement of the M_is and their corresponding load/store instructions are shown in Figure 5.4(c).

R/U assignment: The placement of load/instructions to partitions is followed by the assignment of the R/U-bit to the assigned load/store instructions. This bit specifies whether the placed instruction needs to check only the assigned partitions during cache lookup for the referenced data. If the R-bit is set, only the assigned ways are probed. If the U-bit is set, all the ways are probed. This is orthogonal to the assignment to partitions. The assignment uses the aliased information computed in Section 5.4.1 where, a set of potentially aliasing loads or stores are grouped into a single set. Two instructions from different sets are guaranteed not to interfere. If all instructions within the same points-to set have the same partition assignment, then it is safe to assign the R-bit to each of these instructions. If not, they are assigned the U-bit to avoid coherency and duplication issues. In Figure 5.4(d), since each of the M_is access distinct data objects, they are all assigned the R-bit. Thus, in this example, the compiler was able to restrict each reference to just a single way, thus providing the energy savings of an oracle way-predictor, while maintaining the hit-rate of a 3-way hardware cache by pro-actively avoiding conflicts through careful placement.

5.4.3 Partitioning for Scratch-pads

As described in Section 5.3, configuring the partitions as SPs and caches can help reduce energy by avoiding the tag-checks. For each application, we explore how many

partitions (or ways) are dedicated to SPs and caches [70]. We only consider placing global data objects (like arrays, scalars, etc.) as candidates for SPs. Heap allocated objects are hard to disambiguate and hence are placed in caches. But not all global data objects are good candidates for the SP. More specifically, if for example, a large array has a small working-set (say, due to tiling), then it is a better candidate for cache. This is because caches can capture the dynamic working-set, while for SPs the entire object has to be placed in the SP. We also assume that the SPs are static, i.e., its contents do not change at run-time. Caches allow two temporally disjoint objects to reside on the same cache space, thus leading to better utilization. To account for these two factors, prior to the partitioning phase, all global objects that have working-sets comparable to the object size, and that conflicts with the objects are to be placed in the cache, are placed in the SP.

5.5 Experimental Evaluation

5.5.1 Methodology

We use the Trimaran [96] compiler and simulator infrastructure for our experiments. The simulator is modified to add support for the trace analysis as described in Section 5.4. A parametrized cache simulator was built to model way-partitioning based on the annotated load/store instructions generated by the compiler algorithm illustrated in Section 5.4. We assume a RISC-based, single-issue machine similar to ARM to study the effects of partitioning. The compiler includes aggressive classical optimizations, including function in-lining and pointer analysis for load/store optimizations.

Benchmarks from the MediaBench benchmark suite are evaluated on varying L1 cache sizes and configurations. The partitioning is performed using the train input set run to com-

Benchmark	2-part	4-part	8-part	1-way	2-way	4-way	8-way	16-way
rawcaudio	1.3	1.2	1.3	3.8	1.4	1.4	1.4	1.4
rawdaudio	1.4	1.5	1.5	4.2	1.6	1.5	1.5	1.5
g721encode	0.0	0.0	0.0	1.9	0.0	0.0	0.0	0.0
g721decode	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0
mpeg2dec	19.0	20.3	19.2	43.1	21.4	26.1	35.7	42.2
mpeg2enc	6.7	4.8	4.6	13.8	7.1	5.2	5.7	8.6
pegwitenc	71.0	70.6	70.2	77.0	71.1	70.6	70.3	70.5
pegwitdec	94.9	94.3	93.8	99.9	95.1	94.5	94.1	94.3
pgpencode	21.1	20.1	20.1	23.1	21.1	20.5	20.1	20.1
pgpdecode	2.0	1.6	1.8	5.4	1.8	1.6	1.6	1.5
gsmencode	1.2	1.0	1.2	1.9	1.1	1.0	1.0	1.0
gsmdecode	1.2	1.1	1.0	1.8	1.2	1.1	0.9	0.9
epic	28.2	27.7	28.2	31.3	28.0	27.6	28.3	28.5
unepic	19.4	13.7	13.8	24.8	20.5	13.8	13.6	13.7
cjpeg	39.3	25.8	21.3	43.0	38.4	26.3	22.5	22.1
djpeg	48.5	30.4	24.7	75.0	48.5	30.9	25.9	25.7
Average	**22.2**	**19.6**	**18.9**	**28.2**	**22.4**	**20.1**	**20.2**	**20.8**

Table 5.1: Misses/1000 instructions with different 1-Kb partitioned cache configurations

pletion, while results are reported on a reference input set. The cache sizes are varied from 256-bytes to 8-Kbytes with 32-byte block size. Small cache sizes were chosen because the MediaBench applications have a small memory footprint and hence do not require large caches.

We focus on software-based way-partitioning, where individual load/store instructions are assigned to one or more ways. We evaluate three different way-partition configurations - *2-part*, *4-part*, and *8-part*, where *n-part* denotes a partitioning of the original cache of *n-ways* into *n-partitions* such that each partition is a single way that is direct-mapped and software managed. We compare against traditional hardware-based 1-, 2-, 4-, 8-, and 16-way set-associative cache configurations of different sizes. Since partitioning can default to all ways, the appropriate comparison has to be made between *2-*, *4-*, and *8-part* to the respective 2-, 4-, and 8-way set-associative cache, respectively. We assume

LRU replacement policy for the hardware-based cache configurations. For the partitioned cache, when an instruction is assigned to multiple partitions, LRU is used to select among the assigned partitions.

The basic motivation behind this configuration was to ideally restrict instructions to just one partition, while assigning conflicting instructions to different partitions. This allows only a single smaller direct-mapped way to be activated during each access while achieving the miss-rate equivalent to a *n-way* cache. The goal is to level or even outperform a set-associative cache while being below the access energy envelope of a direct-mapped cache as individual ways are smaller than a unified direct-mapped cache.

5.5.2 Results

Impact on Cache Misses: Table 5.1 shows the misses/1000 instructions for different cache partitions on a 1-Kb cache. On an average, both *4-* and *8-part* partitions perform better than even a 16-way hardware managed cache, while the *2-part* partition performs slightly better than the corresponding 2-way cache. For mpeg2dec, *4-* and *8-part* outperforms the 4-way and 8-way cache, where we observe an increase in misses for higher associativity. This is due to the non-optimality of LRU. Overall, partitioning is able to perform as good as the corresponding set-associative cache.

For the *4-* and *8-part* caches, partitioning is able to remove around 80% of the conflict misses compared to a direct-mapped cache, which is close to a 8-way hardware-managed cache (85%). We observed that capacity misses for the partitioned cache were lower than in the corresponding set-associative cache. An *8-part* cache achieved a 16% reduction in capacity misses compared to an 8-way cache. This is because capacity misses are

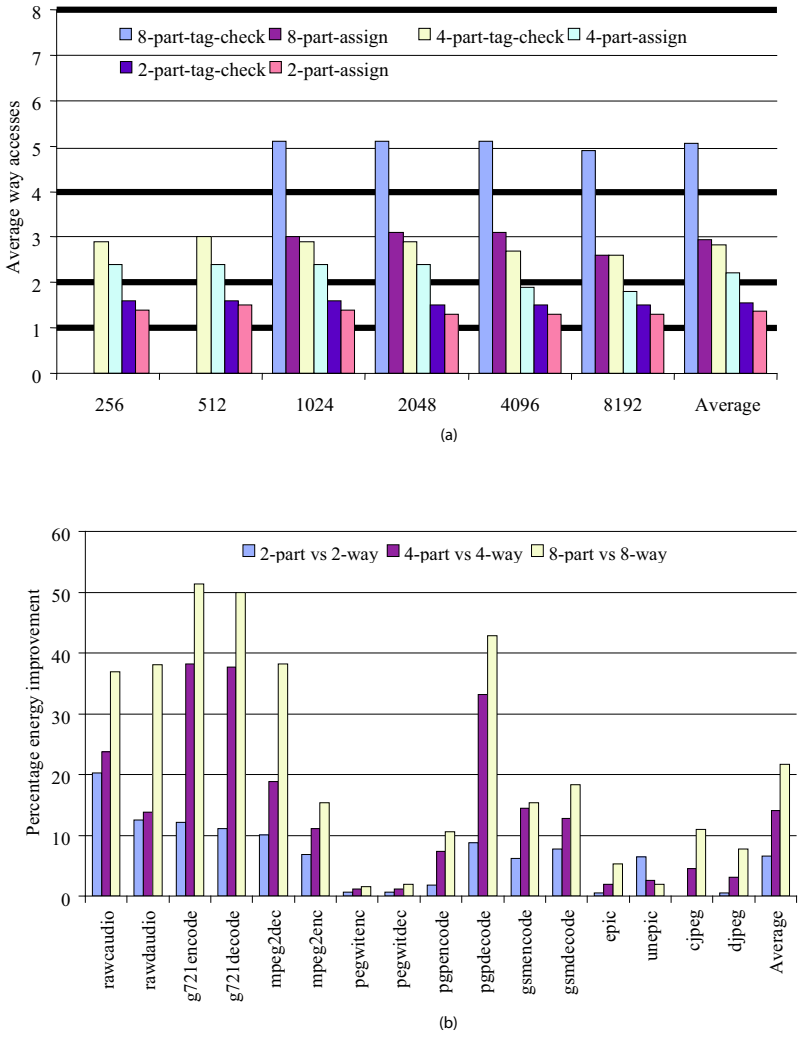


Figure 5.5: (a) Average data/tag-array accesses and partition assignment for different cache sizes and configurations and (b) Percentage reduction in energy for a 1-Kb cache

defined based on a fully-associative cache with LRU, which is not optimal. The compiler-directed data placement is able to eliminate the misses such that it equals or even betters the hardware-based strategy by pro-actively placing the conflicting data elements in different ways.

In our study, we also varied the cache sizes to study the impact of partitioning on different sized caches with different numbers of partitions. In general, we found that for smaller caches the miss-rate improvement is more for partitioned caches because of higher

conflict misses. Higher conflict misses for the base case provides opportunities for the compiler to reduce them by pro-actively avoiding conflicts among co-accessed data items using a whole program knowledge. Secondly, two and four partitions often perform as good as a 8-way set-associative cache. Since fewer partitions consume less energy while maintaining low miss-rates, they are recommended partition configurations.

Tag-checks & Way Assignment: Figure 5.5(a) shows how effective the placement is in restricting the number of ways assigned. It plots two metrics - (i) the average number of dynamic data/tag-array accesses and (ii) the average number of partitions that are assigned per instruction. The first metric is larger than the second as more ways must be probed to check for the presence of the referenced data. The lines shown in bold are the same metric for a hardware managed direct, 2-way, 4-way, and 8-way caches where both the number of tag-checks and replacements are same as the associativity of the cache. The partitioned cache can independently control both the metrics and hence their values are different.

On an average, for the *8-part* configuration, we observe a 36% reduction in cache accesses and a 63% reduction in assigned ways for MediaBench (Figure 5.5(a)). For the *8-part* configuration, the average number of assigned ways (second bar under *Average*) is around 2.9. This means that although there are 8-ways, on an average, it behaves like a 2.9-way associative cache. Other configuration show smaller reductions as they have lesser partitions.

As we scale the number of partitions and their sizes, the number of assigned ways decreases proportionally. This is due to the nature of our heuristics that adapt depending on the available cache partitions. Larger partitions satisfy the working-set needs of

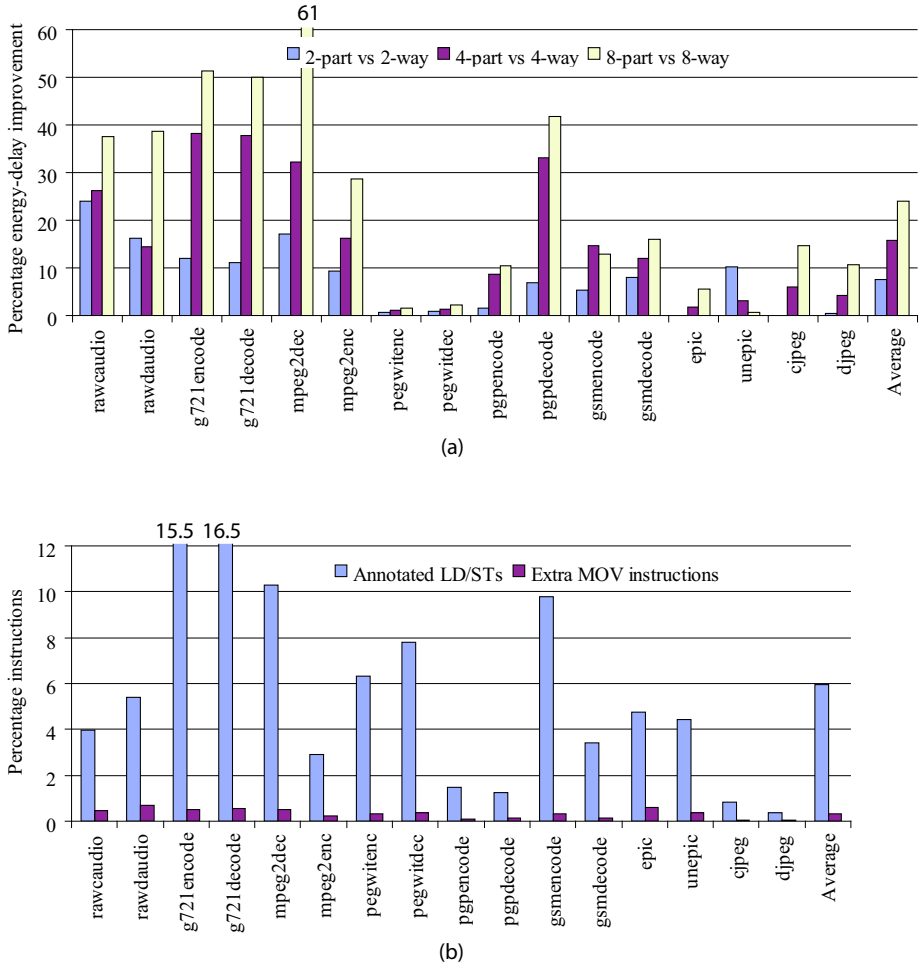


Figure 5.6: For a 1-Kb cache, (a) percentage reduction in energy-delay and (b) percentage annotated ld/st instructions and percent code size overhead.

each instruction and hence there is lesser need for them to “spread” across more ways (Section 5.4.1). Despite assigning to less ways, we are able to achieve miss-rates comparable to hardware-based set-associative caches that have to always activate all the available ways. The reduction in cache accesses provides ample opportunities for energy savings.

Improvement in Energy: We focus our energy results on the cache sub-system as our optimizations target only the caches. Figure 5.5(b) compares the percentage improvements in cache energy saved when compared to the corresponding hardware managed cache with

the same number of ways on a 1-Kb cache. The energy used is obtained using CACTI-3.2 [75] assuming the same physical configuration as shown in Figure 5.1. For all caches, we obtain the energy per access for a single way and scale it by the number of ways activated to compute the total energy [66]. The Am41PDS3228D SRAM [4] was assumed to be the off-chip memory with 3.024nJ per access (16-bits).

Since *8-part* eliminates all of the misses with a few number of ways, it achieves the highest relative energy-savings of around 20% (for MediaBench). The *2-part* is not able to eliminate as many tag/data-array checks when compared to *4-* or *8-part* caches and hence we observe a comparatively smaller relative energy improvement. For *g721encode*, we observe almost 50% savings in relative energy for a *8-part* configuration. On average, when all configurations are compared relative to a direct-mapped cache (not shown), *2-part* partition is the most energy efficient. This is because, the 2-partitions are able to remove most of the misses with an average of 1.6 tag-checks.

Improvement in Energy-Delay: Figure 5.6(a) compares the percentage improvements in the energy-delay product when compared to the corresponding hardware managed cache with the same number of ways on a 1-Kb cache. Since we model a single-issue machine, we use the simple performance equation: $Hits + Misses * MissPenalty$, where miss-penalty is assumed to be the off-chip latency of 25-cycles [82]. Energy-delay shows similar trends as energy, where relatively, *8-part* shows maximum savings.

Code Size: Figure 5.6(b) shows the percentage of load/store instructions that are annotated by the compiler on a *4-part* 1-Kb cache. The compiler annotates only the most frequently executed and profitable instructions, while the rest are assigned to all parti-

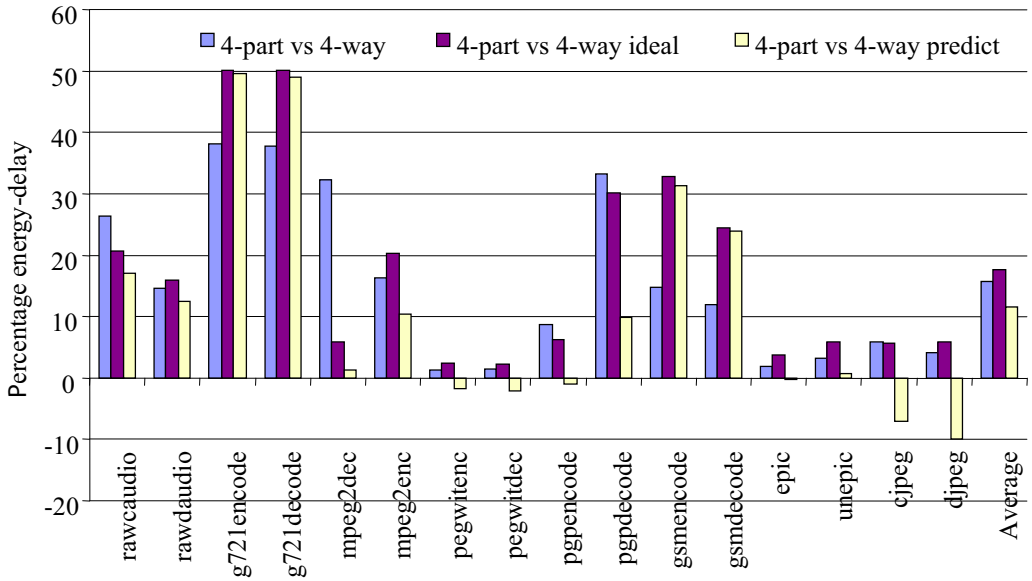


Figure 5.7: Percentage energy-delay reduction of partition data cache vs. way-prediction vs. oracle way-prediction for a 1-Kb cache 4-way associative baseline cache

tions. On average, only 6% load/stores are annotated. As described in Section 5.3, these annotated instructions require an extra-move instruction to initialize the CAR with the bit-vector corresponding to the assigned partitions. Since many instructions are assigned to the same set of partitions, common sub-expression elimination is applied to remove such redundant moves. The graph also shows that the static move instructions inserted average around 0.4%. Thus, the code size increase is negligible, which is critical for embedded processors.

Comparison to Way-Prediction: Way-prediction [13, 66] is a hardware-based technique that tries to reduce cache access power by predicting the way that contains the data. On a miss-prediction, the correct way is then accessed. The prediction is done in parallel to the tag accesses. On a miss, an extra cycle latency is incurred when accessing the cache. We compared our technique against the hardware-based technique and also against

an oracle way-predictor. A 1024 entry PC-indexed table was used for way-prediction. The results for a 4-way 1-Kb cache is shown in Figure 5.7. The oracle predictor was used to provide an upper bound.

On average, the partitioned cache performed 37% better than way-prediction and was within 11% of a prefect predictor. In some of the benchmarks (for example, rawcaudio), the partitioned data cache scheme outperformed the perfect predictor. This is because, in addition to reducing redundant cache accesses, by pro-actively placing conflicting data objects to different locations within the cache, the cache misses are reduced. On average, way-prediction achieved around 80% prediction accuracy. In other benchmarks, like g721encode and gsmencode, the way-predictor did better than partitioned data cache. Here, the way-predictor achieved accuracy close to the oracle predictor (99%). Moreover, these benchmarks had low miss-rates and hence the partitioned cache could not reduce any additional misses. The energy loss is mainly attributed to the overestimation of the number of partitions by the heuristic and is not a fundamental limitation of the architecture.

Partitions as scratch-pad: Finally, we explore the use of converting selected partitions as software-controlled scratch-pads, where the tag-array corresponding to each partition (or way) can be disabled. Figure 5.8 plots the relative improvement in energy for a *4-part* partitioned cache with respect to a 4-way cache. Since there are 4-partitions which can each be individually treated as a scratch-pad, we evaluate four different scratch-pad configurations - (i) all partitions are software-managed partitioned caches (*4-part*, sp0), (ii) 1 partition as scratch-pad while rest are partitioned caches (sp1), (iii) 2-partitions each for scratch-pad and partitioned caches (sp2), and (iv) 3 scratch-pad partitions and a *1-*

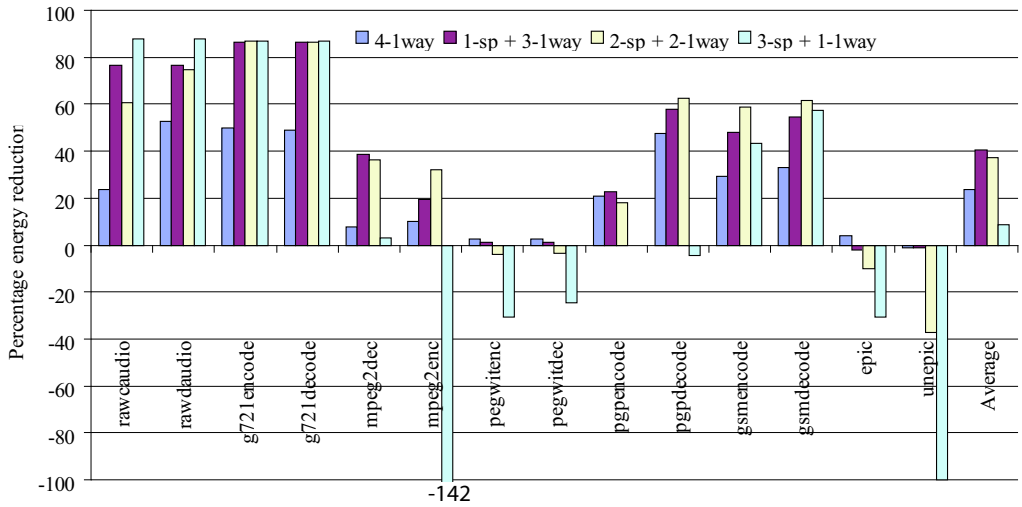


Figure 5.8: Percentage energy reduction for different scratch-pad (sp) configurations w.r.t a 4-way cache of 4-Kb

part cache (sp3). Since scratch-pads do not have the tag-access energy, they are more energy efficient than the correspondingly sized caches. We observe that 1 scratch-pad and *3-part* cache provide the highest energy benefits. Since only global data is placed in the scratch-pad, more caches perform better for benchmarks with heap accesses (epic, unepic). Otherwise, larger scratch-pads help retain more global objects, thus reducing tag-checks for such data (rawdaudio, g721encode). In general, scratch-pads require more memory as the entire data object has to be placed rather than just the working-set. *Cjpeg* and *djpeg* perform poorly with scratch-pads as majority of the data they use are heap allocated and hence require more caches.

5.6 Related Work

Multiple/Split Partitioned Caches: A variety of hardware cache organizations [34, 74, 77] consisting of multiple/split caches aimed at storing data based on spatial, tempo-

ral, or a combination of access pattern behavior have been explored in the past. All of these schemes employ hardware techniques to dynamically classify memory blocks into each of the special caches (partitions). The cache controller is modified to detect the access pattern and route the data to the appropriate cache partition. Recently, hardware-based programmable decoders have been suggested in [107] to reduce conflict misses in direct-mapped caches. Hardware-based dynamic partitioning of shared caches for multiple processes or threads [88] have also been proposed.

The use of compile time classification of memory reference instructions into spatial, temporal, and spatial-temporal has been explored in [78]. The classified data references are then cached into three separate organizations. At run-time, the cache controller places data in a given cache depending on the instruction. Spatial and temporal caches are very small and fully associative, while spatial-temporal caches are larger. Different block sizes are also used for each of the caches.

Although our partitioned cache approach is similar in spirit to earlier work on split caches, our scheme is more flexible in that we allow more generalized form of partitions. In addition, instead of a dedicated hardware controller deciding on what data needs to reside in which partition, the compiler, is used to make partitioning decisions. Most other partitioning schemes physically partition the caches with customized configurations which might not be applicable for all workloads. In comparison, our scheme can easily be defaulted to a traditional unified cache by using regular load/store instructions.

The partitioned cache techniques presented in [42, 72] differ from the scheme we propose in several aspects. Their hardware scheme does not handle coherency issues, whereas

we selectively probe all cache partitions to detect and resolve coherency and data duplication. Our scheme also has the ability to specify multiple non-contiguous partitions with possibly a global replacement among the different partitions. In addition, we only need to partition a select set of load/store instructions and can easily default to a traditional cache based on the needs of the application. [42, 72] use hardware-based partition descriptor tables to record the size and offset of the partitions in the original cache. The PC of the load/store instruction is used to index into another table to identify the assigned partition. This can affect hit time. They limit their analysis to loops with affine accesses and do not handle multiple partitions for a single load/store.

Coarser-Grained Partitioning: While our method uses a fine-grained approach to partitioning, more coarser-grained techniques have been studied in the past. A hardware/compiler scheme is used in [97] to classify an instruction as cacheable or non-cacheable based on the miss rate. In region-based caching [55], caches are partitioned depending on whether an access is to the heap, stack, or the global address space. Minimax caches [101] partition scalars and non-scalars to different caches to reduce conflicts. Page-based partitioning has been proposed in [82], where a smaller direct-mapped cache is placed next to a larger main cache. To avoid conflict misses, page coloring schemes have been proposed in the past [81]. But, these require additional OS support and are dependent on whether the cache is virtually or physically indexed. Moreover, they are targeted towards reducing just the conflicts within a cache and not towards reducing the number of tag lookups. Our scheme can control data placement irrespective of the underlying addressing schemes. It allows more fine-grained partitioning and control and can emulate

the above strategies by annotating the instructions based on their broader classification. Instruction-driven control can generalize to all kinds of data access patterns.

Hardware/Software-Based Cache Management: FlexCache [64] uses software-based cache management by grouping references to hot pages to avoid redundant tag-checks. This can affect hit-case latency, which is reduced using ISA extensions and special registers to store the address translations. They do not target partitioned caches, which combine the benefit of both traditional caches and software management.

Way-partitioning [19] partitions the ways (columns) of a cache such that the replacement decisions are restricted to certain ways. A bit-vector is used to restrict the allowable ways and use the reserved partitions as scratch-pad memories. They do not make fine-grained replacement decisions on a per instruction basis, which allows us to tackle both energy and conflict misses. Their technique performs partitioning at the page-level by modifying the TLB. Moreover, they do not allow restricting lookups to the assigned partitions. Thus, our technique generalizes on their method.

Cache management through compiler specified hints has been proposed in [79] to decide what data is to be retained/evicted. But, their focus is on reducing conflicts and are not always applicable in general, especially in the presence of dynamically allocated data. The hardware still performs energy inefficient tag checks. Our work instead tries to efficiently use the available ways in a set-associative cache for energy improvements while maintaining performance through a mix of static and profile-driven analysis. The proposed solution can be applied over the existing code/data re-organization techniques [18, 76] to further improve performance.

Hardware/Software Techniques Towards Cache Energy Savings: To reduce the energy consumed in set-associative caches, recently, pseudo set-associative caches have been proposed [13, 39, 40, 66]. The basic idea is to probe each of the ways sequentially or use some form of hardware way prediction. For the common case, where the first access results in a hit, there can be substantial savings in energy and access time. Unlike our method, the probing is done in hardware with no compiler control. Our technique is more general, as it can selectively activate different sets of tag/data-arrays for different references in the application without incurring any cycle time overheads. In fact, for instructions that need to access multiple ways, we can allow techniques similar to theirs to further reduce power at the expense of increased cycle time. Dynamically reconfigurable caches have been proposed in [2, 7, 106] where selected portions of the cache can be disabled for energy savings and for dynamically tuning the memory configuration depending on the application's needs. Compiler-based techniques to reduce tag energy have been proposed in [105]. These techniques do not try to reduce conflict misses. Alternately, banking [32] can be deployed to reduce cache access energy, but this is orthogonal to partitioning and can be applied to individual partitions if desired.

5.7 Conclusion

In this chapter, we presented a novel compiler-managed partitioned cache architecture, where individual ways within the cache are explicitly controlled by load/store instructions. These load/store instructions provided hints to the hardware to control placement within individual ways, while regulating the ways that need be probed during cache access. This primary benefit was in avoiding redundant tag/data array checks, thus reducing energy,

while maintaining or even improving performance through intelligent placement of data. In addition, a compiler algorithm that uses whole program knowledge to assign instructions to the partitions with negligible increase in code size is presented. An average 24% energy savings was achieved with four 1-Kb direct-mapped caches when compared to a 4-way set-associative 4-Kb cache.

CHAPTER VI

Conclusion

6.1 Summary

Power is one of the primary design parameters in the design of embedded systems. Embedded processors typically operate under tight environmental constraints where they often do not have access to a constant source of power. Hence, they have to operate at a really low power budget so as to prolong the limited battery life. Many of the current generation embedded processors perform computationally intensive tasks, such as image and video processing with real-time constraints. Thus, it is imperative that we focus on energy saving techniques that do not sacrifice on performance.

Memory power has become one of the dominant contributors to the overall system power. Current techniques to solve the memory power problem can be broadly classified into hardware and software techniques. In general, hardware-only solutions tend to be expensive and are limited in their scope of optimization. But on the positive side, they can adapt to the dynamically varying program conditions and have less performance overhead. Software-only solutions, on the other hand, are usually conservative so as to guarantee correctness. Moreover, they are limited to statically analyzable code and data access patterns,

like loop-nests and arrays. They do not adapt well to dynamically allocated data that are accessed through pointers. But, they offer the advantage of having a whole program knowledge to pro-actively optimize for the entire application.

This thesis demonstrated synergistic hardware/software techniques that address the memory power problem. A combined hardware/software solution not only provides the advantages of both hardware and software-only solutions, but also allows aggressive software optimizations. Through ISA extensions, the compiler provides hints to the hardware for better program management, while the critical operations are delegated to the hardware so as to not compromise performance.

Register files and on-chip memories form a hierarchy of storage structures that are faster and low power. By maximizing the utilization of these on-chip structures, off-chip accesses can be avoided, thus resulting in substantial savings in memory power. Any hardware/software technique should therefore try to efficiently utilize these structures so as to save power without sacrificing on performance. We have proposed and evaluated three different hardware/software techniques that effectively utilize the on-chip register files, instruction, and data memories. These techniques were evaluated within the context of the WIMS microcontroller that is used in low-power embedded sensor-based systems.

First, we looked at a windowed register file architecture that provides the appearance of a large register file without compromising on the instruction encoding space. A novel graph partitioning compiler algorithm was developed that partitions the virtual registers in a procedure into multiple register windows, thus reducing the overall spill code while minimizing the overhead due to inter-window moves and window swaps. We evaluated our

design over a wide range of processor and window configurations. Increasing the number of windows from 1 to 2 yielded an average performance improvement of 10% for the 4-register case and 11% for the 8-register case on the WIMS processor. The corresponding experiment on a 5-wide VLIW machine, achieved an average performance improvement of 21% and 25% for the 4 and 8 register configurations, respectively. An average power reduction of 25% for the 2-window 8-register over the 1-window case was observed on the WIMS processor.

Second, we evaluated a compiler-directed scheme to effectively manage a low power scratch-pad memory. Unlike traditional instruction or data caches, scratch-pad memories lack the complex tag checking and comparison logic, thereby proving to be efficient in area and power. We proposed a compiler-managed dynamic placement algorithm, wherein multiple hot code sequences, or traces, are made to overlap each other in the code cache at different points in time during execution. Special copy instructions are provided to copy the traces at run-time. The compiler, based on a power estimate, initially selects the most frequent traces across all procedures into the code cache. Through iterative code motion and redundancy elimination, copy instructions are inserted in infrequently executed regions of the code to copy traces into the code cache. For a 64-byte code cache, the compiler-managed dynamic scheme achieves over 64% energy improvement over the static-based solution in a low-power embedded microcontroller.

Finally, we evaluated a partitioned data cache architecture, where a unified set-associative cache was replaced by multiple direct-mapped caches with the same combined size. Traditional data caches achieve high performance, but at the expense of increased dynamic

power due to often redundant tag and data-array checks. We evaluated a combined hardware/software scheme where, each of the smaller cache partitions are exposed to the compiler through load/store instruction extensions. This allows the compiler to make better placement decisions to reduce cache misses. More importantly, the compiler also orchestrates which data partitions are activated on a lookup, thus removing unnecessary tag and data-array accesses. An average 24% energy savings was achieved with four 1-Kb direct-mapped caches when compared to a 4-way set-associative 4-Kb cache.

6.2 Putting it All Together

The register window and software-managed scratch-pads are implemented within the WIMS microcontroller. Although the partitioned data cache work is more exploratory and no implementation exists in any current generation microprocessors, we evaluated the projected savings on the WIMS processor due to partitioning of the data caches.

6.2.1 Methodology

To evaluate the combined benefits of all three optimizations, similar experimental setup as detailed in the earlier chapters was used. For the register window case, the instruction-level energy model of the WIMS processor was used to calculate the energy savings of scaling the number of register windows from 2 to 4. The base case assumed was the WIMS processor with a single register window of 8-registers. The WIMS processor has an all on-chip memory that is partitioned into 4-banks of 16-Kb each. So any spill loads/stores saved reduces the number of on-chip memory accesses.

To evaluate the loop-cache configuration, the read/write access energy for the on-chip

loop-cache and the on-chip memory was used. Any access that is not directed to the loop-cache goes to the on-chip memory. The size of the loop-cache was varied from 64-bytes to 512-bytes. Both static and dynamic loop-cache allocation schemes were evaluated. For larger loop-cache sizes and for applications with a not too large instruction memory footprint, the simpler static scheme performed better than the dynamic scheme. The loop-cache on the WIMS processor is just a small bank of on-chip memory. Since the WIMS processor has an all on-chip memory, the energy difference between an access to the loop-cache and the main memory banks are relatively small. To measure the overall system energy savings for the loop-cache, the percentage contribution of an on-chip memory access was computed and was found to be close to 20% of the system energy. This is not surprising given that an instruction is accessed almost every cycle.

Finally, for the partitioned data cache scheme, an on-chip 1-Kb direct mapped data cache and a 4-Kb 4-way set-associative cache were assumed to be the baselines, with the miss being directed to an off-chip memory. The Am41PDS3228D SRAM [4] was assumed to be the off-chip memory with 3.024nJ per access (16-bits). The number of partitions were varied from 2 to 8. As mentioned earlier, the current generation WIMS processor does not include a data cache, so the savings obtained is just an estimate. CACTI [75] was used to model the energy for different cache configurations. The overall contribution to the system energy by the direct-mapped data cache was found to be around 15%.

6.2.2 Results

The combined energy savings due to register windows, scratch-pads, and partitioned data caches are shown in Figure 6.1. The results are shown for a 2-window 8-register con-

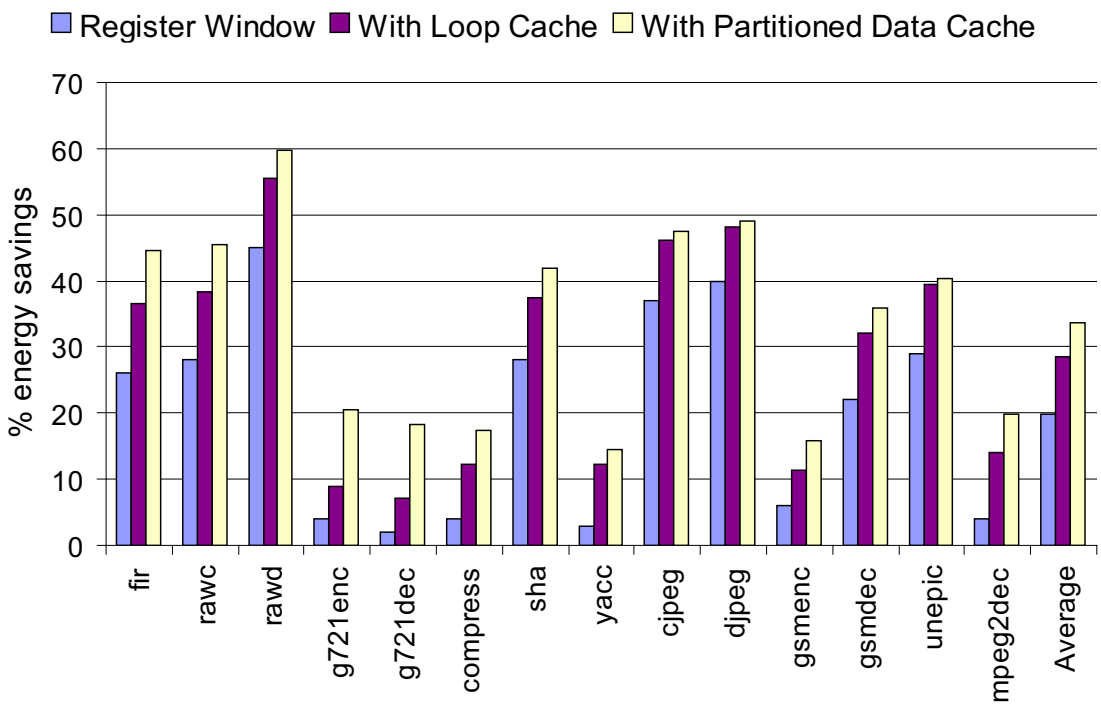


Figure 6.1: Combined energy savings on the WIMS processor using 2-windows of 8-registers, a 512-byte loop cache, and a 1-Kb partitioned data cache.

figuration with a 512-byte scratch-pad and a 1-Kb data cache that is partitioned into multiple 256-byte direct mapped caches. The baseline is a 1-window 8-register machine with no on-chip scratch-pad, but an on-chip instruction memory, and a 4-way set-associative data cache. On average, we observe close to 30% savings in dynamic energy savings when all three optimizations are applied.

The major contributor to the overall system energy savings is the register window scheme. This is primarily due to savings in the number of spill instructions and the resulting reduction in the number of main memory accesses. The contribution to the overall system energy savings due to the software-managed loop-cache is comparatively lower. But, here the savings is measured against an all on-chip main memory. Since WIMS has a

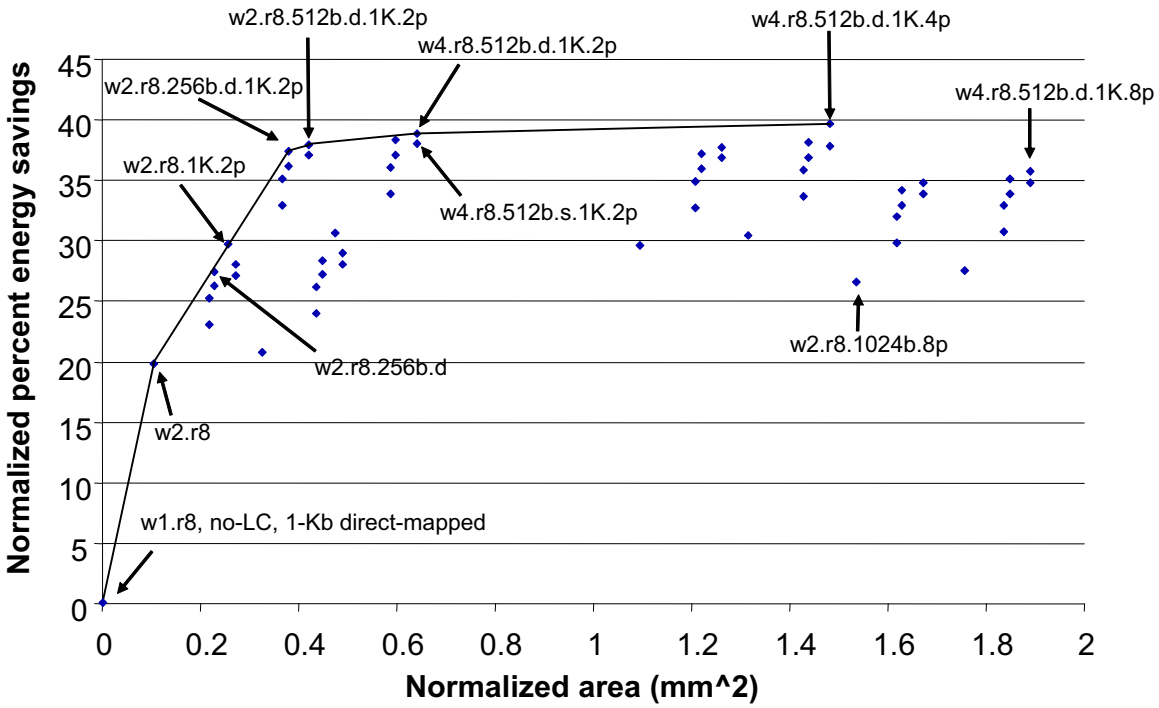


Figure 6.2: Pareto optimal graph with area overhead incurred on the x-axis and the energy savings obtained on the y-axis for different window, loop-cache, partitioned data cache configurations.

64-Kb on-chip memory that is partitioned into 4 banks of 16-Kb, the energy savings due to a 512-byte on-chip loop cache is not pronounced. But for most embedded processors that use off-chip memories, the corresponding savings would be significantly higher.

Similarly, the contribution by the partitioned data cache to overall system energy is comparatively low. For the data cache, we assume that a miss results in an off-chip access. Since a 4-way cache has a high hit-rate, the number of off-chip accesses are low. Much of the energy savings is due to savings obtained by avoiding redundant tag/data-array accesses. The contribution by the 4-way 1-Kb data cache to the overall system energy is found to be around 20%. As shown in Chapter V, although we are able to save around 18% of the cache access energy, the overall contribution to system energy savings is smaller.

Figure 6.2 shows the Pareto optimal graph for different configurations of register windows, loop-cache, and partitioned data caches. The baseline for all points is 1-window of 8-registers, on-chip instruction memory, and a 1-Kb direct-mapped cache. For reference, the base WIMS processor with a 512-byte loop-cache, 64-Kb on-chip memory, and 2-windows of 8-registers with support for peripherals, on-chip ADC is around $12.5mm_2$. The notation w2.r8 and w4.r8 denotes 2- and 4-windows of 8-registers each. 512b.d denotes a 512-byte loop cache that is dynamically allocated, while 1K.2p and 1K.4p refers to a 1-Kb cache with 2 and 4 partitions. As we add more features, such as larger loop-caches, more register windows, or more number of cache partitions, the relative energy savings is higher. But, we also observe an increase in the chip area due to these structures. At one extreme is a 2-window 8-register configuration which has the least energy savings and lowest area overhead, while at the other end is a 4-window 8-register configuration with a 512-byte dynamically managed loop-cache, and a 4-partition cache configuration. The knee of the Pareto optimal curve is a 2-window 8-register configuration with a 256-byte loop cache and a 2-partition data cache.

6.2.3 Discussion

Due to high register pressure, scaling to two windows provided the highest overall benefit. Beyond two windows, the relative energy benefit does not justify the area overhead of adding more registers. Of course, this depends on the nature of the application. For a wider issue machine, where there is even more register pressure, scaling to four windows might be more beneficial.

Although the loop-cache provided less significant energy savings, many embedded

processor do include an on-chip loop-cache to avoid the power hungry off-chip accesses without incurring the area/energy penalty of a traditional instruction cache. Overall, smaller loop-cache sizes of the order of 256 to 512-bytes provided the maximum energy savings. For smaller loop-cache sizes, the dynamic scheme clearly out-performs the static scheme, although for larger sizes, static seems to be an attractive option. In fact, one scheme does not nullify the usage of the other. Thus, depending on the nature of the application and the size of the loop-cache, a combined scheme might provide more benefit than either one used in isolation.

For the partitioned data cache, the four partition configuration provided the maximum energy savings. This provides just sufficient associativity to reduce the conflict misses, while allowing the compiler enough flexibility in isolating accesses to different partitions so as to pro-actively reduce conflicts while eliminating redundant cache accesses. Scaling to eight partitions actually resulted in reduced energy savings with higher area overhead. Although this exploration did not include the use of the cache as a scratch-pad, from the results in Chapter V, for more regular benchmarks, configuring selected partitions as scratch-pads provides added energy savings. Thus, partitioning provides the flexibility of tuning the memory sub-system depending on the needs of the application to improve both power and performance.

To conclude, all three schemes must be collectively applied to reduce the overall memory power. Depending on the available area budget, the sizes of each of the hardware structures can be varied. Given a processor with one or more of these features, the compiler techniques developed in this dissertation can be used to effectively utilize these storage

structures to save energy and also to improve performance.

6.3 Future Directions

The work presented in this dissertation can be extended in numerous directions. One possible variation of the register windowing algorithm presented in Chapter III is to reorder the instructions so as to reduce the number of window toggles. A compiler could perform a separate pass prior to window assignment to schedule the instructions while honoring the dependencies such that, the instructions that could potentially be assigned the same window are grouped together.

The windowed register file architecture limits register accesses to a single window at a time. This prevents any accesses to variables that are stored in the other window, unless they are explicitly moved to the current active window. The disadvantage of such a scheme is that if the current active window is saturated, it can introduce extra spills to accommodate the variable from the other window. An architectural variation to avoid this is to use a sliding window just as in the SPARC architecture. This will allow better sharing of registers across windows. By sliding the active window, registers from both the windows can be accessed without any move or spill overhead. This reduces the number of moves, because often, a single instruction needs to source its variables from multiple windows.

Although we implemented the windowing architecture to target just a single register file, in the WIMS processor, both the address and the data register files are windowed. But, both these files have separate bits in the MSR to control the active window. Having separate toggle instructions for data and address registers can affect performance. The

older generation WIMS processor had a single toggle bit that toggled both the data and address registers simultaneously. Although this can constrain the compiler, it can reduce the toggle overhead. A possible extension to the compiler algorithm would be to handle both data and addresses simultaneously so as to reduce spill pressure on both the register files, while minimizing the combined toggle overhead.

On the memory side, a hybrid scratch-pad plus cache solution can help get the combined benefits of both scratch-pads and hardware-managed caches for both data and instructions. For statically analyzable code and data access patterns, a dynamically managed scratch-pad can be used to improve both power and performance. The hardware-managed cache is useful for hard to analyze code like dynamically allocated data or third party libraries, OS etc.

Beyond compilers, virtual machines or dynamic compilation systems can provide a system level solution to better manage the underlying hardware. Compilers usually are restricted to optimizing a single application. But, a typical system will have multiple processes/threads that constantly interact or contend with each other for limited hardware resources. One of the limitation of the loop-cache system is that the compiler assumes that the loop-cache is dedicated to a single application So on a context switch it is required that the contents of the loop-cache be saved and restored. This might be too much energy and performance overhead. One naive solution is to partition the loop-cache and dedicate parts of it to critical portions of the application like the interrupt handler routines or the OS. But this has the obvious disadvantage of providing too little or too much to certain applications. This possibly diminishes the advantage of a dynamic code placement system.

Perhaps a better solution would be to have a run-time system that can handle multiple applications. A run-time system, has a whole system view that can help manage the resources across applications. In fact, future systems will have close levels of interaction between the compiler and the run-time system. The compiler can convey the application's resource requirements to the dynamic system, while the run-time system in turn, uses this information to optimize across applications. One excellent example of such an interaction is the partitioned data cache, where the compiler analyzes an application for its memory requirements at different points in the program, and assigns cache partitions virtually. The run-time system can then use this information to allocate physical partitions to different applications, taking into account the current state of the system.

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] D.H Albonesi. Selective Cache Ways: On Demand Cache Resource Allocation. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 248–259, November 1999.

[3] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 281–290, September 2002.

[4] AMD. *Am41PDS3228D SRAM*, 2004.

[5] Analog Devices. *ADSP-219x/2191 DSP Hardware Reference Manual*, July 2001. http://www.analog.com/Analog_Root /static/library/dspManuals/ADSP-2191_hardware_reference.html.

[6] A.W Appel and L. George. Optimal spilling for cisc machines with few registers. In *Proc. of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 243–253, June 2001.

[7] Rajeev Balasubramonian et al. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 245–257, December 2000.

[8] Rajeshwari Banakar et al. Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proc. of the Tenth International Conference on Hardware/Software Codesign*, pages 26–32, May 2002.

[9] N. Bellas et al. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. In *Proc. of the 1999 International Conference on Computer Design*, page 378, October 1999.

[10] L. Benini et al. Asymptotic Zero-Transition Activing Encoding for Address Busses in Low-Power Microprocessor-Based Systems. In *Proc. of the 7th Great Lakes Symposium on VLSI*, page 77, March 1997.

[11] Mark Brehob and Richard Enbody. An Analytical Model of Locality and Caching. Technical Report MSU-CSE-99-31, Michigan State University, September 1999.

[12] Preston Briggs. Register allocation via graph coloring. Technical Report TR92-183, Rice University, Houston, April 1992.

[13] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Caches. In *Proc. of the 2nd International Symposium on High-Performance Computer Architecture*, page 244, February 1996.

[14] Center for Wireless Integrated Microsystems (WIMS). *An NSF Engineering Research Center*, 2000. http://www.wimserc.org.

[15] G.J Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–101, June 1982.

[16] P.P. Chang, S. A. Mahlke, W. Y. Chen, N. F. Warter, and W. W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proc. of the 18th Annual International Symposium on Computer Architecture*, pages 266–75, 1991.

[17] Bruce Childers and Tarun Nakra. Reducing Memory Bus Transactions for Reduced Power Consumption. In *Proc. of the 2000ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 146–161, June 2000.

[18] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proc. of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[19] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devdas. Application Specific Memory Management in Embedded Systems Using Software-Controlled Caches. In *Proc. of the 37th Design Automation Conference*, pages 416–419, 2000.

[20] Jeonghun Cho, Yunheung Paek, and David Whalley. Register and memory assignment for non-orthogonal architectures via graph coloring and mst algorithms. In *Proc. of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 130–138, June 2002.

[21] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[22] S. Cotterell and F. Vahid. Tuning Loop Cache Architectures to Programs in Embedded System Design. In *Proc. of the 13th International Symposium on on System Synthesis*, pages 8–13, October 2002.

[23] José-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, and Nigel Topham. Multiple-banked register file architecture. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, June 2000.

[24] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, February 1998.

[25] D. Dobberpuhl. The design of a high-performance low-power microprocessor. In *Proc. of the 1996 International Symposium on on Low Power Electronics and Design*, pages 11–16, 1996.

[26] Jan Elder and Mark D. Hill. Dinero IV trace-driven uniprocessor cache simulator, 2003.

[27] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard Laboratories, December 1998.

[28] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, December 1997.

[29] Marcio M. Fernandes, Josep Llosa, and Nigel Topham. Allocating lifetimes to queues in software pipelined architectures. In *Proc. of the 3rd International Euro-Par Conference*, pages 1066–1073, August 1997.

[30] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. of the 19th Design Automation Conference*, pages 175–181, 1982.

[31] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[32] Kanad Ghose and Milind B. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *Proc. of the 1999 International Symposium on on Low Power Electronics and Design*, pages 70–75, August 1999.

[33] Nikolas Gloy et al. Procedure Placement Using Temporal Ordering Information. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 977–1027, December 1997.

[34] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the 1995 International Conference on Supercomputing*, pages 338–347, July 1995.

[35] A Gordon-Ross et al. Exploiting fixed programs in embedded systems : A loop cache example. *Computer Architecture Letters*, 1(1):2, 2002.

[36] Matthew Guthaus, Jeffrey Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, and Richard Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the 4th IEEE Workshop on Workload Characterization*, pages 10–22, December 2001.

[37] B. Hendrickson and R. Leland. *The Chaco User's Guide*. Sandia National Laboratories, July 1995.

[38] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–23, October 2000.

[39] M. Huang et al. L1 Data Cache Decomposition for Energy Efficiency. In *Proc. of the 2001 International Symposium on on Low Power Electronics and Design*, pages 10–15, August 2001.

[40] K. Inoue, T. Ishihara, and K. Murakami. Way Predicting Set-Associative Caches for High Performance and Low Energy. In *Proc. of the 1999 International Symposium on on Low Power Electronics and Design*, pages 273–275, August 1999.

[41] Intel Corporation, Santa Clara, CA. *Intel IA-64 Software Developer's Manual*, 2002.

[42] J. Irwin et al. Predictable Instruction Caching for Media Processors. In *IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, pages 141–150, July 2002.

[43] Illya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Data reuse analysis techniques for software controlled memory hierarchies. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 202–207, February 2004.

[44] M. Kandemir et al. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):243–260, February 2004.

[45] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, September 1998.

[46] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1993.

[47] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–207, February 1970.

[48] Hansoo Kim. *Region-Based register allocation for EPIC Architectures*. PhD thesis, Department of Computer Science, New York University, 2001. www.crest.gatech.edu/publications/hansooth.pdf.

[49] N. Kim et al. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, February 2004.

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

[51] K. Kiyohara, S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, S. Anik, and W. W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 247–256, May 1993.

[52] J. Knoop et al. Lazy code motion. In *Proc. of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[53] Arvind Krishnaswamy and Rajiv Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Proc. of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 55–63, June 2002.

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

[55] Hsien-Hsin S. Lee and Gary S. Tyson. Region-based Caching: an Energy Efficient Memory Architecture for Embedded Processors. In *Proc. of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 120–127, November 2000.

[56] L. Lee et al. Instruction Fetch Energy Reduction Using Loop Caches for embedded applications with Small Tight Loops. In *Proc. of the 1999 International Symposium on on Low Power Electronics and Design*, pages 267–269, August 1999.

[57] L. Lee et al. Low-Cost Embedded Program Loop Caching - Revisited. Technical Report CSE-TR-411-99, Univ. of Michigan, October 1999.

[58] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. of the 35th Design Automation Conference*, pages 516–521, 1998.

[59] Rainer Leupers and Daniel Kotte. Variable partitioning for dual memory bank dsps. In *Proc. of the IEEE International Conference on Acoustics Speech and Signal Processing*, pages 1121–1124, May 2001.

[60] Eric Marsman, Robert Senger, Michael McCorquodale, Mathew Guthaus, Rajiv Ravindran, Ganesh Dasika, Scott Mahlke, and Richard Brown. A 16-bit, Low-Power Microcontroller with Monolithic MEMS-LC Clocking. In *Proc. of the International Conference on Circuits and Systems*, pages 624–627, May 2005.

[61] Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[62] R. L Matsson, J. Gecsei, D. Slutz, and I.L Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[63] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993.

[64] C. Moritz, M. Frank, and S. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *Proc. of the Workshop on Intelligent Memory Systems*, pages 8–14, November 2000.

[65] Motorola. *CPU12 Reference Manual*, June 2003. http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf.

[66] M.Powell, S.Yang, B.Falsafi, K.Roy, and T.N.Vijaykumar. Gated-VDD: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. of the 2000 International Symposium on on Low Power Electronics and Design*, pages 90–95, July 2000.

[67] E. Nystrom, H-S Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, August 2004.

[68] Ozcan Ozturk and Mahmut Kandemir. Nonuniform Banking for Reducing Memory Energy Consumption. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 814–819, March 2005.

[69] Krishna V. Palem and Rodric M. Rabbah. Design Space Optimization of Embedded Memory Systems via Data Remapping. In *Proc. of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 28–37, June 2002.

[70] P.R Panda, F. Cathoor, N.D Dutt, et al. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, April 2001.

[71] Preeti Ranjan Panda, Nikil Dutt, and Alex Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, Norwell, MA, 1999.

[72] Peter Petrov and Alex Orailoglu. Performance and Power Effectiveness in Embedded Processors - Customizable Partitioned Caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(11):1309–1318, November 2001.

[73] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[74] Milos Prvulovic et al. The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation. In *Proc. of the IEEE TCCA Newsletter*, pages 3–10, July 1999.

[75] G. Reinman and N. P. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical Report WRL-2000-7, Hewlett-Packard Laboratories, February 2000.

[76] G. Rivera and C.-W Tseng. Data transformations for eliminating conflict misses. In *Proc. of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, June 1998.

[77] Jude A. Rivers and Edward S. Davidson. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In *Proc. of the 1996 International Conference on Parallel Processing*, pages 154–163, August 1996.

[78] J. Sanchez and A. Gonzalez. A Locality Sensitive Multi-Module Cache with Explicit Management. In *Proc. of the 1999 International Conference on Supercomputing*, pages 51–59, June 1999.

[79] Jennifer Sartor, Subramanian Venkiteswaran, Kathryn McKinley, and Zhenlin Wang. Cooperative Caching with Keep-Me and Evict-Me. In *Proc. of the* 9^{th}*Annual Workshop on Interaction between Compilers and Computer*, pages 46–57, February 2005.

[80] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[81] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing cache misses using hardware and software page placement. In *Proc. of the 1999 International Conference on Supercomputing*, pages 155–164, June 1999.

[82] Aviral Shrivastava, Illya Issenin, and Nikil Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 90–96, September 2005.

[83] Abraham Silberschatz et al. *Operating System Concepts*. John Wiley and Sons, Inc, Indianapolis, IN, 2001.

[84] M. Smelyanskiy, G.S. Tyson, and E.S. Davidson. Register queues: A new hardware/software approach to efficient software pipelining. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 3–12, October 2000.

[85] SPARC International Inc. *The SPARC Architecture Manual, Version 8,* 1992. www.sparc.com/standards/V8.pdf.

[86] S. Steinke et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of the 2002 Design, Automation and Test in Europe*, page 409, March 2002.

[87] S. Steinke et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. of the 13th International Symposium on on System Synthesis*, pages 213–218, October 2002.

[88] G.E Suh, L. Rudolph, and S. Devdas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, April 2004.

[89] MIPS Technologies. *MIPS32 Architecture For Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture*. MIPS Technologies, March 2001.

[90] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory level. *IEEE Transactions on Computers*, 48(2):150–158, 1999.

[91] Tensilica Inc. *Xtensa Architecture and Performance*, September 2002. http://www.tensilica.com/xtensa_arch_white_paper.pdf.

[92] Texas Instruments. *TMS320C54X DSP Reference Set*, March 2001. http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.

[93] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, June 2004. http://focus.ti.com/lit/ug/spru189f/spru189f.pdf.

[94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[95] Hiroyuki Tomiyama and Hiroto Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[96] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[97] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[98] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 276–286, September 2003.

[99] Uh et al. Techniques for Effectively Exploiting a Zero Overhead Loop Buffer. In *Proc. of the 9th International Conference on Compiler Construction*, pages 157–172, March 2000.

[100] Osman S. Unsal et al. Cool-Cache for Hot Multimedia. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 274–283, December 2001.

[101] Osman S. Unsal et al. The Minimax Cache: An Energy-Efficient Framework for Media Processors. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, pages 131–141, February 2002.

[102] T. VanderAa et al. Instruction buffering exploration for low energy VLIWs with instruction clusters. In *Proc. of the 9th Asia and South Pacific Design Automation Conference*, pages 824–829, January 2004.

[103] Manish Verma et al. Dynamic overlay of scratchpad memory for energy minimization. pages 104–109, September 2004.

[104] S.J.E. Wilton et al. CACTI: An enhanced cache access and cycle time model. *Journal of Solid State Circuits*, 31(5):677–688, May 1996.

[105] Emmett Witchel, Sam Larsen, Scott Ananian, and Krste Asanovic. Direct Addressed Caches for Reduced Power Consumption. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 124–133, December 2001.

[106] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T.N Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, pages 151–161, February 2002.

[107] Chuanjun Zhang. Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 155–166, June 2006.

ABSTRACT

HARDWARE/SOFTWARE TECHNIQUES FOR MEMORY POWER

OPTIMIZATIONS IN EMBEDDED PROCESSORS

by

Rajiv A. Ravindran

Chair: Scott Mahlke

Power has become one of the primary design constraints in modern embedded microprocessors. Many embedded applications perform computationally demanding tasks, such as signal processing and encryption, that require high performance processors. Hence, traditional power savings techniques that sacrifice performance for power are not always applicable. It is well known that the memory sub-system is responsible for a significant fraction of the overall power dissipation. On-chip memory structures, such as register files, caches, and scratch-pads, provide fast and energy efficient access to program and data by reducing the slower and power hungry off-chip accesses. Memory power, therefore, can be reduced by employing techniques to effectively utilize these storage elements. In this dissertation, three different compiler orchestrated, but hardware-assisted techniques, are proposed that target these on-chip storage elements while remaining performance neutral.

A windowed register file architecture that provides a large number of physical registers without compromising on the instruction encoding is proposed. A novel graph partitioning compiler algorithm was designed that partitions virtual registers within a procedure across multiple windows. This reduces memory demand by avoiding spills, thus improving both power and performance.

Scratch-pad memories, unlike traditional hardware caches, lack the complex tag checking and comparison logic, thereby proving to be efficient in area and power. A compiler managed dynamic instruction placement algorithm was designed wherein, multiple hot code sequences are made to overlap each other in the scratch-pad at different points in time during execution through specially provided copy instructions.

Finally, data caches have been effective in dealing with more irregular data access patterns. But, they employ hardware-based lookup and replacement schemes that have high energy overheads. A partitioned data cache architecture is proposed in which, enhanced load/store instructions are used to control fine-grain data placement and lookup within a set of cache partitions. This fine-grain control can avoid conflicts, thus providing the performance benefits of highly associative caches, while saving energy by eliminating redundant tag- and data-array accesses.

These techniques are evaluated within the context of a low-power WIMS microprocessor resulting in a combined system energy savings of around 32%.