The Dissertation Committee for Madhavi Gopal Valluri
certifies that this is the approved version of the following dissertation:

A Hybrid-Scheduling Approach for Energy-Efficient Superscalar Processors

Committee:

Lizy John, Supervisor

Jacob A. Abraham

Margarida F. Jacome

Stephen W. Keckler

Kathryn S. McKinley

A Hybrid-Scheduling Approach for Energy-Efficient Superscalar Processors

by

Madhavi Gopal Valluri, B.E., M.Sc(Engg), M.S.E

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2005

To my family

Acknowledgments

It gives me immense pleasure to thank the many people who made this thesis a reality for me.

This work would not have been possible without the relentless support and encouragement offered by my thesis supervisor Prof. Lizy John. I thank her for all the valuable insights and the constant guidance she provided during the course of my doctoral studies. I would also like to thank members of my dissertation committee, Prof. Jacob Abraham, Prof Margarida Jacome, Prof. Steven Keckler and Prof. Kathryn McKinley, for taking the time and effort to help me improve this dissertation. Prof. McKinley, with whom I have collaborated very closely, provided invaluable assistance during the final year of this work.

I thank all my friends within and outside UT for making my stay in Austin thoroughly enjoyable. I especially owe a great deal to many current and former LCA members for creating an excellent environment in the lab. Aashish, Ajay, Byeong, Hari, Hareesh, Lloyd, Rob, Sean, Shiwen, Yue, Juan, Tao, Ravi, Deepu, and Ramesh have all sat through countless practice talks and provided critical technical assistance. Several people outside of LCA have also helped me with my experimental setup. Particularly, Rodric Rabbah, Ramdas Nagarajan, Nitya Ranganathan and Satish Pillai have provided significant assistance with the compiler framework.

I have benefited immeasurably from my association with two amazing people. Randy and Heather, my colleagues at UT, have also become close friends over the years. With Randy, I have enjoyed numerous technical, non-

technical and even philosophical discussions over several hundred pots of tea. I am still amazed by Heather's incredible patience and positive attitude. She always found ways to help me even during her busiest times. I will cherish these friendships forever.

I am indeed fortunate to have wonderful parents and a terrific brother who have always believed in me. They have made me what I am today. I thank everyone in my immediate and extended family, including members of Indupinni's and my husband's family, for their affection and constant encouragement.

Finally, I would like to thank (although a simple thanks is not nearly enough!) my husband and best friend for his unconditional love and support. He was the pillar of strength that helped me persevere through all the vagaries of graduate school. I am truly grateful to him for the innumerable things he has done for me over the past few arduous years.

Madhavi Gopal Valluri

The University of Texas at Austin
May 2005

A Hybrid-Scheduling Approach for Energy-Efficient Superscalar Processors

Publication No. _____

Madhavi Gopal Valluri, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Lizy John

The management of power consumption while simultaneously delivering acceptable levels of performance is becoming a critical task in high-performance, general-purpose micro-architectures. Nearly a third of the energy consumed in these processors can be attributed to the dynamic scheduling hardware that identifies multiple instructions to issue in parallel. The energy consumption of this complex logic structure is projected to grow dramatically in future wide-issue processors.

This research develops a novel Hybrid-Scheduling approach that synergistically combines the advantages of compile-time instruction scheduling and dynamic scheduling to reduce energy consumption in the dynamic issue hardware. This approach is predicated on the key observation that all instructions and all basic-blocks in a program are not equal; some blocks are inherently easy to schedule at compile-time, whereas others are not. In this scheme,

programs are thus partitioned into low power "static regions" and high power "dynamic regions". Static regions are regions of the program for which the compiler can generate schedules comparable to the dynamic schedules created by the run-time hardware. These regions bypass the dynamic issue units and execute on specially designed low-power, low-complexity hardware.

An extensive evaluation of the proposed scheme reveals that the Hybrid-Scheduling approach wherein instructions are routed to a scheduling engine tuned to a region's characteristics can provide substantial reduction in processor energy consumption while concurrently preserving high levels of performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuing advances in the semiconductor technology have provided tremendous performance gains in general-purpose microprocessors. The remarkable improvements are not only due to the high clock rates afforded by smaller process technologies but also due to innovations in architectural techniques. The ability to pack more transistors on the die has allowed today's processors to include large multi-level caches for alleviating memory bottlenecks. In addition, modern microprocessors employ complex techniques such as out-of-order execution, dynamic register renaming, control speculation and data value speculation, to achieve high performance. These features extract instruction-level parallelism (ILP) in programs to substantially reduce program execution time. While these sophisticated architectural techniques and small feature sizes have fueled unprecedented levels of performance improvements over the years, they have at the same time introduced new challenges. Relative to previous generations, microprocessors today exhibit higher transistor switching activity rates and higher leakage currents. Increased activity on the chip is causing power dissipation to grow significantly from one generation of microprocessors to the next, despite using advanced processor technologies with reduced supply voltages. Table 1.1 shows the power dissipation trends of four generations of Alpha processors [2, 26, 43]. The table shows power dissipation increasing almost linearly with frequency, with power exceeding 100 Watts at 1 GHz in the Alpha 21364 [43].

Table 1.1: Power dissipation trends in the Alpha processor family

	21064	21164	21264	21364
Transistor Count (Mill)	1.68	9.3	15.2	100
Die Size (cm^2)	2.33	2.99	3.14	3.5
Process Technology (um)	0.75	0.50	0.35	0.18
Power Supply V_{cc} (Volts)	3.3	3.3	2.2	1.5
Avg. Power Dissipation (W)	30	50	72	>100
Avg. Supply Current (Amps)	9.1	15.2	32.7	66.6
Design Frequency (MHz)	200	300	600	1000
Instruction Issue/Cycle	2	4	6	6
Execution Flow	In-order	In-order	Out-of-Order	Out-of-Order
L1I-L1D-L2 (KB)	8-8	8-8-96	64-64	64-64-1536

1.1 Why does power dissipation matter?

High power dissipation and energy consumption in the processor present several formidable challenges.

- Power dissipated in the processor is converted into heat. Heightened temperatures on the processor die can severely impact circuit reliability due to increased electro-migration and hot-electron degradation effects. Further, heat generated by the processor must be removed from the surface of the die. Increased heat thus complicates cooling and packaging. For high-performance processors, cooling costs are rising at $1-3 or more per Watt of heat dissipated [9, 53].

- High-performance general-purpose processors are being increasingly used in the portable devices such as laptops where battery life is at a premium. While semiconductors and devices track Moore's law and double in functionality every 18 months, the capacity of rechargeable batteries increases at a rate of only 5% to 10% per year [17]. The increase in battery capacity is thus outstripped by the increasing dynamic and leakage power in processors, leading to shorter battery lifetimes.

- Data centers typically collocate a large number of servers in one location. One of the essential tasks in such a facility is to ensure that the computing equipment is cooled down effectively. Heating, ventilation and air conditioning (HVAC) are thus critical components of all major data centers. Noted as one of the most costly operational expenses, HVAC has been estimated to account for 40% to 60% of power use in data centers, while servers account for only 20% [33].

- Increased power dissipation has significant impact on the environment. The worldwide total power dissipation of processors in PCs was 160 megawatts in 1992, growing to 9000 megawatts by 2001 [43]. In the same vein, 8% of US electicity in 1998 was attributed to the Internet and is expected to grow to about 30% by 2020 [43].

Due to all the above detriments, power dissipation and energy consumption have emerged as first-order constraints in the design of future microprocessors.

1.2 Power dissipation in superscalar processors

Figure 1.1 shows the distribution of power dissipated in various hardware structures for several recent superscalar microprocessors. The figure shows that dynamic scheduling hardware (also known as the out-of-order issue logic), accounts for a significant portion of the power budget. The energy consumption of the out-of-order issue logic accounts for nearly one-third of the overall energy in existing processors [11] [22] [26] and is projected to grow substantially in future processors [68][69].

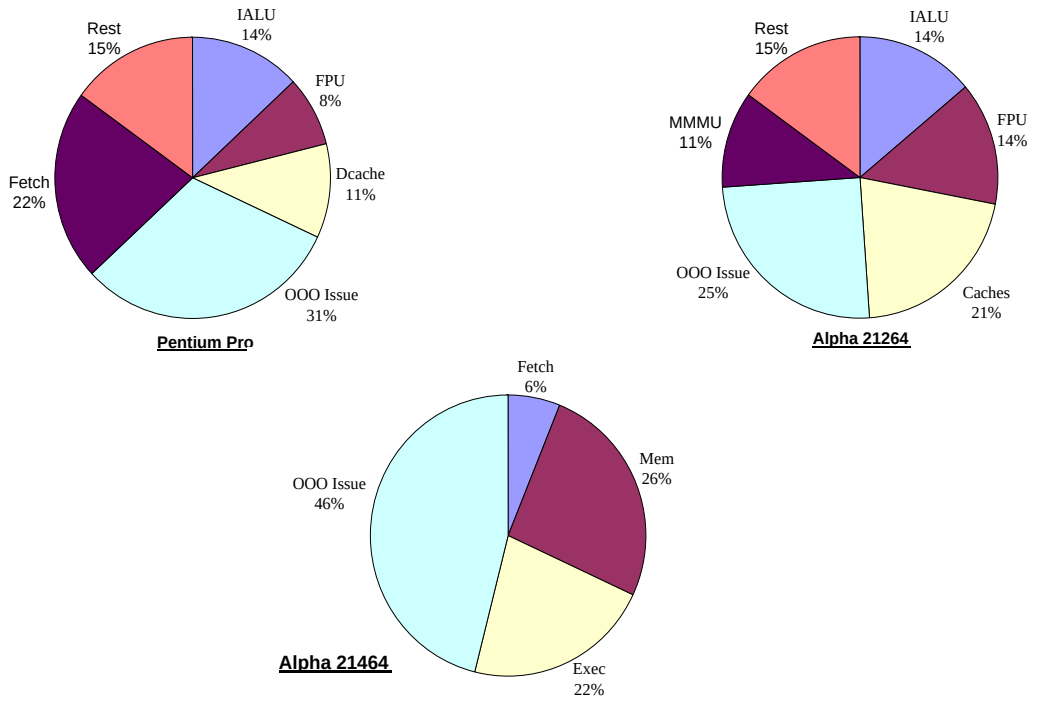


Figure 1.1: Energy breakdown in typical superscalar processors. The out-of-order issue hardware accounts for a nearly a third of the processor power budget.

The dynamic issue hardware is useful for extracting ILP in applications. In the presence of complex and unpredictable control-flow in programs, even the best compilers frequently fail to expose sufficient ILP in the program. The dynamic scheduler analyzes a sequential stream of instructions and identifies independent instructions to issue in parallel. The hardware scheduler has run-time knowledge regarding program input data and control flow and can thus handle dependences that are unknown to the compiler. Consequently, schedules generated by the out-of-order issue logic are in general superior to compiler-generated schedules.

The logic that facilitates dynamic issue consists of several elaborate

hardware structures such as the *issue queue, reorder buffer, load-store queue* and *register renaming units* (shown in Figure 1.2).

The issue queue (also known as the issue window) is used for holding decoded instructions while they await their operand values. In each cycle, a few instructions whose dependences have been satisfied are selected from this queue and issued to the function units. Since state-of-the-art superscalar processors issue multiple instructions each cycle, issue queues are typically fully associative buffers with multiple entries and multiple ports. The issue queue also consists of *wakeup* and *select* logic blocks. These logic units are not only acknowledged to be large power sinks [69], but have also been identified as key structures limiting the operating frequency of the processor [46].

The wakeup logic is responsible for waking up instructions waiting for their operands to become available. In each cycle, completing instructions typically broadcast the identities of their destination registers (called *tags*) to all waiting instructions in the issue queue. Each instruction entry in the issue queue compares these results with the source operands of the instruction to determine its readiness to issue. The wakeup logic thus requires a large number of ports and complex circuitry to compare operands values. Increasing the number of ports on queue structures not only leads to considerable degradation of access latencies but also increases energy consumption substantially [8, 52].

The select logic is responsible for choosing instructions to execute from a pool of ready instructions. The complexity and power dissipation of this logic grows linearly with the size of the issue queue.

In superscalar machines with out-of-order issue logic instructions can be issued in an order other than the original program order and hence require a mechanism to the maintain the precise program state. Most processors utilize a *reorder buffer* (ROB) for this task. The ROB typically stores instruction

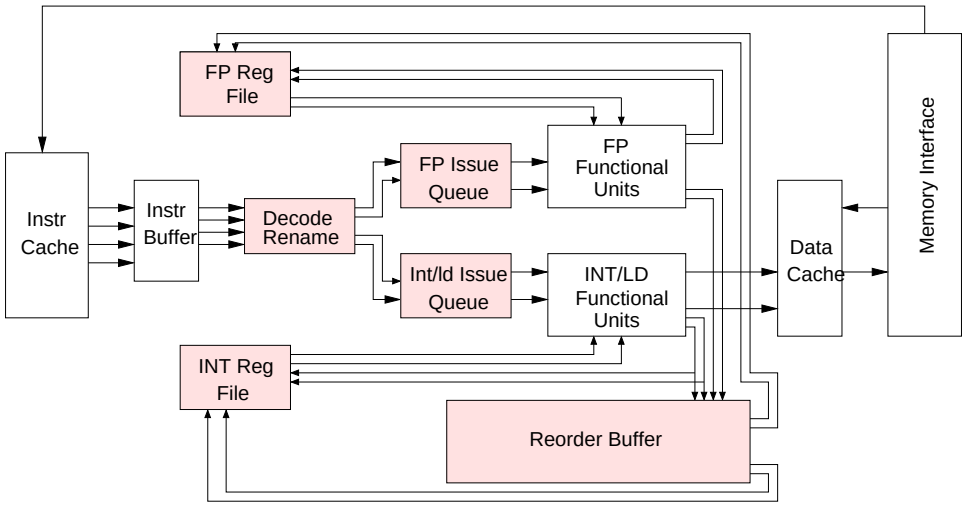


Figure 1.2: Typical out-of-order issue superscalar processor. Shaded portions represent logic units that facilitate dynamic issue.

identities such as the instruction PC (program counter) and operand tags. In some architectures, the speculative values of registers (*i.e.*, values of registers for instructions still in the imprecise state) are held in the ROB as well [70]. While instructions issue and even complete in any order, they commit their changes to the register file only in their correct program order. Much like the issue queue, the ROB is also a highly associative queue with multiple entries and multiple ports. Separate ports are required for (a) establishing the ROB entries, (b) reading out part of an ROB entry when the valid data value for the most recent entry for an architectural register is read out and (c) reading out all of a ROB entry at the time of committing an instruction.

In several architectures, rather than storing the speculative values in the reorder-buffer, the values are stored in a separate physical register file [26, 29]. This register file also requires multiple ports, at least two read and one write port per instruction issued each cycle.

Modern superscalar processors also provide run-time memory disam-

biguation to enable out-of-order issue of load and store instructions. The out-of-order issue logic therefore also includes an associative Load-Store Queue (LSQ) to maintain load/store dependences.

Thus, the out-of-order issue logic consists of several multi-ported and associative complex hardware structures that are accessed multiple times each cycle. These power-hungry structures cause the out-of-order issue logic to be a chief energy sinks in modern superscalar processors. Furthermore, the power consumption of this hardware is projected to grow quadratically with increasing issue widths and window sizes in future processors [69].

1.3 Solutions to reduce power dissipation

In the past, high power dissipation in superscalar processors was addressed primarily at the circuit-level. Clock gating, power supply reduction, smaller process technology, low swing buses and state-of-the-art packaging are all examples of techniques that are traditionally applied to alleviate the power/energy bottlenecks [26]. However, conventional circuit-level techniques have reached their limits as the demand for processors with higher clock speeds and denser transistor counts continues to raise. Recently, power optimizations at the architecture and software-level (i.e., compiler, operating system, and application) have begun to receive increasing attention.

A host of architectural-level solutions that target different units have been suggested. These techniques range from alternate complexity-effective hardware structures [1, 69] to techniques that lower energy consumption by dynamically reconfiguring hardware resources in the processor based on run-time program requirements [14, 22, 25, 30, 47].

Compilers for desktop and server systems are traditionally not exposed

to the energy details of the processor. These compilers are tuned primarily for performance and occasionally for code size. However, with the heightening power dissipation problem, it has become to imperative to engage all components of the system to aid in reducing processor power and complexity. With a few exceptions, power-aware compiling for out-of-order issue processors is a largely unexplored area of research [31, 61, 63, 64, 65],

There are several key benefits that a compile-time approach can offer. Static techniques can often be applied orthogonally to hardware techniques, making it possible to simplify the logic beyond what is achievable with hardware techniques alone [63, 64, 65]. Additionally, the compiler has accurate information regarding the region to be executed in the near future and hence allows energy saving optimizations to be applied at finer granularities when compared to run-time hardware techniques [31, 61, 63, 64, 65].

1.4 The Hybrid-Scheduling solution

This dissertation presents a Hybrid-Scheduling scheme that uses both compiler analysis and micro-architectural innovation to lower energy consumption in superscalar processors. This cooperative hardware/software technique attempts to harness the work done by the compile-time instruction scheduler to reduce energy consumption in the power-hungry out-of-order logic units of superscalar processors.

The instruction scheduling phase in the compiler reorders and packages instructions into groups of parallel instructions. The function of the static scheduler is identical to that of the dynamic hardware scheduler. Generally, static schedules are of an inferior quality when compared to dynamic schedules since the compile-time scheduler is limited by unknown memory latencies,

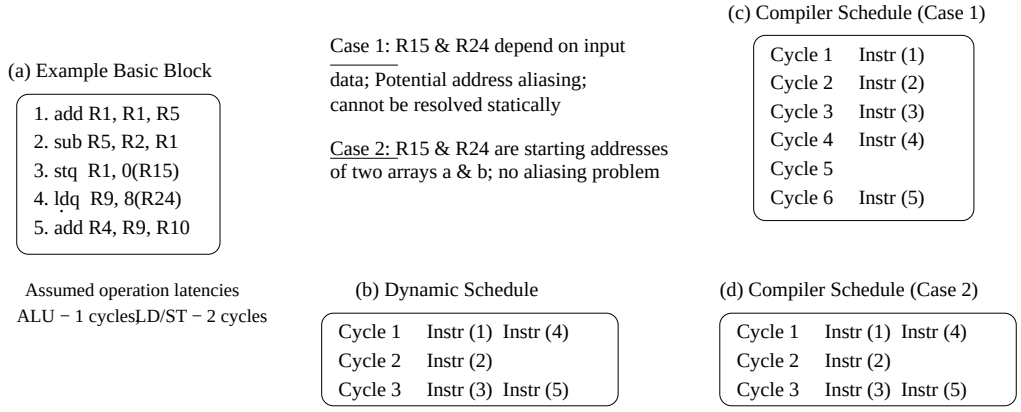


Figure 1.3: Illustrative example. (a) Example basic block (b) Dynamic schedule (c) Compiler schedule when the block has an unresolved memory dependence and (d) Compiler schedule when there are no unresolved memory dependences.

unknown control flow, limited architectural registers and unresolved memory aliases. However, close examination of several media, scientific and general-purpose programs, reveals that although portions of programs suffer from the above impediments, a significant number of program basic blocks are free of memory misses, false-dependences and unresolved alias edges [63][65]. Consequently, the static schedules of these blocks are near-optimal, and are not very different from their corresponding dynamic schedules. A simple example is shown in Figure 1.3.

Consider the case where the compiler is unable to resolve the addresses of two potentially independent load and store operations statically (case 1). The instruction scheduler adds a data-dependence edge between these operations and forces them to execute sequentially. The dynamic scheduling issue logic however has run-time knowledge and the ability to dynamically disambiguate the addresses and can schedule the load operation ahead of the store

if they are independent. For this case, the compiler-generated schedule (Figure 1.3(c)) is significantly worse than the schedule achieved by the dynamic issue logic (Figure 1.3(b)). On the other hand, consider the case wherein the compiler is able to resolve the addresses statically (case 2), there is no difference between the compiler-generated schedule (Figure 1.3(d)) and the dynamic schedule. Thus, in blocks where the compiler has perfect knowledge, instructions do not require dynamic reordering and can potentially issue in their statically-scheduled order.

In the Hybrid-Scheduling paradigm, programs are thus divided into low power "static regions", called *S-Regions* and high power "dynamic regions". The compiler, with the help of profile-guided hints, estimates the reorder requirements of each region and classifies basic-blocks/regions that can be scheduled in an efficient manner statically as S-Regions. S-Regions bypass the dynamic issue hardware in the processor and execute on specially designed low-power, low-complexity hardware. A high-level view of the proposed scheme is shown in Figure 1.4. The Hybrid-Scheduling architecture thus saves energy consumption while preserving high levels of performance by using aggressive and power-hungry scheduling hardware for application regions that warrant it, while facilitating low energy execution for structured regions that do not need such hardware.

This dissertation presents two variations of the Hybrid-Scheduling architecture. The generic Hybrid-Scheduling scheme caters to a diverse set of applications and exploits static schedules at a basic-block level to reduce energy consumption and complexity of the issue queue. The proposed Hybrid-Scheduling micro-architecture uses a novel issue queue structure that provides two separate issue queues for static and dynamic blocks. The *Dual-Mode* Hybrid-Scheduling architecture, on the other hand, provides two separate is-

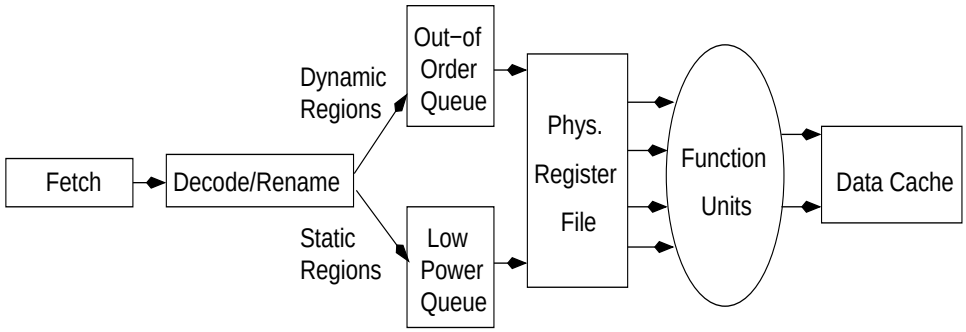


Figure 1.4: High-level view of the Hybrid-Scheduling scheme

sue modes. This scheme is particularly suitable for regular applications such as media and scientific, where, the execution time is typically dominated by regular loops. For well-understood structures such as loops, the compiler can exploit and enhance parallelism with the help of aggressive optimizations such as loop unrolling, software pipelining and trace scheduling. The Dual-Mode Hybrid-Scheduling scheme exploits compile-time schedules at these larger, loop-level granularities to reduce energy consumption in several microarchitectural components in out-of-order issue hardware including the issue queue, reorder buffer, rename logic, branch predictor and the instruction cache. With nominal increase in design complexity and chip area, the generic and Dual-Mode Hybrid-Scheduling micro-architectures can also potentially be combined for larger overall energy savings. Such a trade-off can be particularly useful in general-purpose desktop systems which cater to diverse application domains such as integer, media and scientific.

1.5 Thesis statement

By combining the advantages of both compile-time static scheduling and run-time dynamic scheduling, energy consumption in out-of-order issue superscalar processors can be significantly reduced while simultaneously delivering acceptable levels of performance.

1.6 Dissertation contributions

This dissertation makes several key contributions:

- An in-depth evaluation of the effectiveness of several state-of-the-art compile-time optimizations in reducing power is presented. This study provides insight into how typical performance optimizations impact power dissipation in superscalar processors. Further, an analysis of several existing power-aware compiler optimizations in the context of out-of-order issue processors is provided along with a discussion on the limitations of current techniques and areas for improvement.

- The Hybrid-Scheduling technique is based on the observation that not all instructions in a program require the same amount of dynamic reordering. The necessary compiler heuristics for evaluating the quality of static schedules are developed.

- A Hybrid-Scheduling micro-architecture is proposed and evaluated. The proposed Hybrid-Scheduling scheme utilizes a unique Reorder-Sensitive issue queue structure to exploit the varying dynamic scheduling requirement of basic blocks and lowers power dissipation and complexity of the dynamic issue hardware.

- A variation of the Hybrid-Scheduling scheme that is particularly suitable for regular applications such as media and scientific applications is presented. These applications typically contain large contiguous regions that can be classified as S-Regions. Considering S-Regions at larger granularities offers opportunities to simplify the hardware requirements dramatically. A complete description and evaluation of the compiler heuristics and micro-architectural details are provided.

- A combined compiler/simulator framework called SPHINX is developed for evaluating the proposed scheme. SPHINX integrates a detailed out-of-issue processor simulator that is largely based on Simplescalar's *sim-outorder* simulator with the Trimaran 2.0 [72] compiler framework. The tool also incorporates power models derived from Wattch [11] in the simulator to estimate power. The SPHINX framework thus provides a unified platform for exploring a range of software and hardware techniques for low power.

1.7 Organization of the dissertation

This dissertation is organized as follows:

Chapter 2 provides a detailed evaluation of the effectiveness of several several state-of-the-art compile-time optimizations in reducing processor power dissipation. This study identifies potential limitations and areas of improvement in existing optimizations.

Chapter 3 presents an an overview of the proposed Hybrid-Scheduling scheme. The chapter first describes the basic qualities of an S-Region and develops the compiler heuristics to quantitatively identify S-Regions. S-Regions

require inherently less complex resources. The hardware requirements of S-Regions are discussed next. Finally, some of the unique advantages of the Hybrid-Scheduling approach are summarized.

Chapter 4 describes the SPHINX compiler/simulator framework in detail. Chapter 5 provides an evaluation of the Hybrid-Scheduling scheme using the SPHINX framework. Power dissipation and performance results of the scheme along with a an evaluation of the compiler heuristics are presented.

Media and scientific applications are important applications in the desktop and server computing markets respectively. The execution time of these applications is dominated by a few regular loops. Chapter 6 presents a Dual-Mode Hybrid-Scheduling scheme in the context of regular applications. A detailed evaluation of the proposed micro-architecture and the compiler heuristics is also provided.

Chapter 7 briefly reviews previous contributions in areas related to the Hybrid-Scheduling scheme. Chapter 8 provides some directions for future research and finally, Chapter 9 presents the main conclusions of this dissertation.

Chapter 2

Impact of Compiler Optimizations on Power: A Preliminary Evaluation

This chapter provides a detailed evaluation of the effectiveness of several existing compile-time optimizations in reducing processor power dissipation. The objectives of this study are to identify potential limitations and areas of improvement in current optimizations. The first section of the chapter examines the impact of standard performance optimizations on processor power. Current compiler optimizations are tuned primarily for performance; they are not exposed to the energy details of the underlying processor. However, an interesting question is - if programs are compiled for performance, are they automatically compiled for low power? Further, there have been several compiler optimizations proposed for reducing power dissipation in VLIW (Very Long Instruction Word) and in-order issue processors. The second part of the chapter examines several of these optimizations for their applicability to modern out-of-order issue superscalar processors.

A key observation from this study is that optimizations that typically enhance parallelism in programs increase power dissipation despite an improvement in performance [62]. Further, the study also reveals that optimizations such as common-subexpression elimination and dead-code elimination, that

improve program performance by reducing the total number of instructions executed lower the total energy consumption as well [62].

2.1 Evaluating impact of performance optimizations on power

Many compilers today contain fairly advanced performance optimizations that are fully exposed to different features of the target machine such as available function units, issue width of the processor, available memory ports, etc. However, compilers are not provided with appropriate power dissipation models for different hardware structures, and hence, programs are typically not compiled for low power. Nevertheless, it is interesting to examine the impact of performance optimizations on processor power. This section presents a quantitative study of several state-of-the-art optimizations in DEC Alpha's *cc* and *gcc* compilers on power dissipation in superscalar processors. Standard optimization levels in these compilers along with several individual optimizations are evaluated. A brief description of the optimizations is presented first.

Standard optimization levels

-O0 No optimizations performed. Only variables declared register are allocated in registers.

-O1 Many local optimizations and global optimizations are performed. These include recognition and elimination of common subexpressions, copy propagation, induction variable elimination, code motion, test replacement, split lifetime analysis, and some minimal code scheduling.

-O2 This level performs inline expansion of static procedures. Additional

global optimizations that improve speed (at the cost of extra code size), such as integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches are also performed. Loop unrolling is not performed in *gcc* at this level.

-O3 Includes all **-O2** optimizations and also performs inline expansion of global procedures performed.

-O4 Software pipelining, an aggressive instruction scheduling technique used for exploiting instruction-level parallelism (ILP) in loops is performed. Vectorization of some loops on 8-bit and 16-bit data is also done. This level also invokes a scheduling pass which inserts NOP instructions to improve scheduling. This optimization level is supported only in the *cc* compiler.

The standard optimization levels in both the *cc* and *gcc* compilers include very similar optimizations. Additionally, in both compilers, the optimizations that increase the ILP in a program are in levels `-O2` and higher.

Individual optimizations

The individual optimizations examined are *basic-block scheduling, loop unrolling, function inlining, and aggressive global scheduling* in the *gcc* compiler. All the individual optimizations are applied in addition to optimizations performed at `-O1`.

-fschedule-insns This optimization performs basic-block list-scheduling. It is run after local register allocation.

-fschedule-insns2 This option invokes aggressive global scheduling before and after global register allocation. Postpass scheduling (when scheduling is done after register allocation) minimizes the pipeline stalls due to the spill instructions introduced by register allocation.

Table 2.1: Baseline processor configuration

Feature	Attributes
Issue Units	32 entry Register Update Unit 16 entry Load Store Queue Fetch/Decode/Issue/Commit width - 4
Cache Hierarchy	32KB 4-way L1 Dcache (1-cycle hit), 32KB DM L1 Icache (1-cycle hit), 512KB 4-way L2 (20-cycle hit)
Memory	40 cycles memory latency
Branch Pred.	2K combinational predictor
Function units	4 integer ALUs, 4 FP ALUs Units, 2 integer multiply units, 2 FP multiply units, 2 load/store units

-finline-functions Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

-funroll-loops Perform the optimization of loop unrolling for loops whose number of iterations can be determined at compile time or run time.

2.1.1 Experimental setup

This section quantitatively examines the performance optimizations on a typical out-of-order issue processor and isolates the optimizations that are particularly beneficial for mitigating processor power and energy consumption in contemporaneous superscalar processors.

The processor configuration used for this study is provided in Table 2.1 and the experiments are conducted within the Wattch 1.0 simulator framework [11]. Wattch is an architectural simulator that estimates CPU energy consumption. The power/energy estimates are based on a suite of parameterizable power models for various hardware structures in the processor and on the resource usage counts. The power models are interfaced with Sim-

pleScalar [13]. Sim-outorder, SimpleScalar's out-of-order issue simulator has been modified to keep track of which unit is being accessed in each cycle and record the total energy consumed for an application. The power models for the technology parameters of .35um process, 600MHz and 5V. With these parameters, the processor dissipates a maximum power of 78 Watts. Unused units dissipate 10% of their maximum power. This corresponds to the static power dissipated when there is no activity in unit. More details on the Wattch simulator can be found in Chapter 4 of this dissertation and also in the Wattch technical paper [11]. The different compiler optimizations are evaluated for six benchmarks. The benchmarks used are: three SpecInt95 benchmarks, namely *compress, go* and *li*, two SpecFp95 benchmarks *su2cor* and *swim*, and *saxpy*, a toy benchmark.

2.1.2 Results

The following subsections present a detailed analysis of the results obtained. First the influence of standard optimizations on energy and power is discussed, following which the effects of individual optimizations on power are outlined.

Influence of standard optimizations on energy

Table 2.2 shows the results obtained when the benchmarks are compiled with different standard optimizations levels. The results of all optimizations are presented relative to the results of optimization level `-O0`. For example, the percentage of instructions executed by a benchmark optimized with option `-O2` is given by:

$$\%\text{ of Insts Executed by } Prog_{O2} = \frac{\#\text{ of Insts Executed by } Prog_{O2}}{\#\text{ of Insts Executed by } Prog_{O0}} * 100$$

Table 2.2 shows that the number of instructions committed drops drastically from optimization `-O0` to `-O1`, and also drops significantly in codes optimized with `-O2` and `-O3`. There is however a very marginal increase in the number of instructions in *compress*. In codes optimized with the `-O4` option, the number of instructions increases due to the extra NOPs code generated for scheduling.

The reduction in number of instructions directly influences execution time. The performance improvement is significant in `-O1` when compared to `-O0`, sometimes as high as 73% (*swim*). Options `-O2, -O3` also lead to significant improvement over `-O1`. For example, an 8% improvement can be seen in *li* with `-O2` optimization. In some benchmarks like *saxpy* the improvement is only about 0.6%. Optimizations `-O2, -O3` improve performance in *compress* even though the number of instructions increases.

The energy consumed by a program is directly proportional to the number of instructions. In all benchmarks, the energy consumption decreases when the total number of instructions executed decreases. Note that although `-O2, -O3` improve performance in *compress*, the energy consumed is higher. This is primarily because of the higher number of instructions executed at these optimization levels, *i.e.*, the amount of "work done" is higher. Hence, if programs are required to be compiled for reducing energy consumption without sacrificing performance, optimizations such as common sub-expression elimination, induction variable elimination and unrolling that reduce the number of instructions executed should be applied.

Table 2.2: Effects of standard optimizations in cc and gcc on power dissipation and total energy consumption

Benchmark	opt level	Energy	Exec Time	Insts	Avg Power	IPC
compress	O0	100.00	100.00	100.00	100.00	100.00
	O1	74.48	81.55	81.52	91.33	99.96
	O2	75.13	81.44	82.04	92.25	100.73
	O3	75.13	81.44	82.04	92.25	100.73
	O4	79.01	82.77	86.11	95.45	104.03
go	O0	100.00	100.00	100.00	100.00	100.00
	O1	66.20	64.13	68.94	103.23	107.50
	O2	62.62	61.31	63.01	102.14	102.78
	O3	62.62	61.31	63.01	102.14	102.78
	O4	63.67	62.19	63.75	102.38	102.51
li	O0	100.00	100.00	100.00	100.00	100.00
	O1	81.32	83.66	83.18	97.20	99.42
	O2	79.60	75.97	82.97	104.78	109.21
	O3	79.60	75.97	82.97	104.78	109.21
	O4	85.71	77.89	90.96	110.05	116.78
saxpy	O0	100.00	100.00	100.00	100.00	100.00
	O1	97.38	100.24	92.49	97.15	92.27
	O2	97.69	99.38	92.49	98.30	93.07
	O3	97.69	99.38	92.49	98.30	93.07
	O4	98.31	99.27	92.84	99.02	93.51
su2cor	O0	100.00	100.00	100.00	100.00	100.00
	O1	42.09	51.04	33.21	82.46	65.06
	O2	40.99	47.52	33.10	86.28	69.67
	O3	40.99	46.37	33.10	87.65	71.38
swim	O0	100.00	100.00	100.00	100.00	100.00
	O1	30.10	36.64	20.01	82.15	54.63
	O2	28.93	34.01	19.05	85.06	56.01
	O3	28.93	34.01	19.05	85.06	56.01

Influence of standard optimizations on power

Table 2.2 shows that although the number of instructions and the number of execution cycles for a program reduces at higher optimization levels, the num-

ber of instructions do not reduce sufficiently to keep the instructions per cycle (IPC) constant. IPC of programs typically increases at optimization levels `-O2` and higher. The primary reason is that most optimizations that increase IPC such as instruction scheduling and loop unrolling are included at these levels. Power dissipated is the amount of work done in one cycle which is directly proportional to the IPC. Hence, optimizations that increase IPC, directly increase the power dissipated. Instruction scheduling and other `-O2, -O3` optimizations are suitable for performance improvement but are not suitable for systems where power dissipation is a concern. These are some optimizations that can be significantly improved exposing the power consumption details of the underlying processing units. Note, IPC in `-O0` is high because of the poor quality of code produced. Unoptimized codes usually feature a large number of independent (although superfluous) instructions. These instructions increase the IPC of the program without accomplishing useful work. Another interesting observation from Table 2.2 is that since optimizations such as common subexpression elimination improve code by reducing instructions rather than increasing available parallelism, IPC does not increase in `-O1` codes.

Influence of individual optimizations on energy and power

Tables 2.3 to 2.8 show the results for experiments on different individual optimizations and their impact on processor power/energy. The results are shown for each benchmark separately. The tables show the performance, power, and energy for each of the optimizations relative to performance, power, and energy of code with `-O0`. Since the optimizations are applied over the `-O1` option, all results are compared to results of the programs with optimization level. The effects of the instruction scheduling are discussed first.

Table 2.3: Individual optimizations on Compress

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.0	100.0	100.0	100.0	100.0
O1	67.66	74.68	60.46	90.60	80.95
inline-func	67.69	74.68	60.46	90.63	80.95
sched-instr2	68.82	74.94	63.21	91.82	84.35
sched-instr	66.66	73.47	59.83	90.72	81.43
unroll-loops	66.84	74.19	59.90	90.09	80.74

Table 2.4: Individual optimizations on li

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	70.91	74.67	66.18	94.96	88.63
inline-func	71.02	73.14	68.00	97.11	92.97
sched-instr2	69.56	66.65	68.33	104.36	102.52
sched-instr	69.56	66.65	68.33	104.36	102.52
unroll-loops	66.05	59.91	68.19	110.24	113.81

Table 2.5: Individual optimizations on saxpy

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	96.78	98.56	96.21	98.19	97.61
inline-func	96.78	98.56	96.21	98.19	97.61
sched-instr2	97.07	97.14	96.27	99.93	99.11
sched-instr	96.79	98.52	96.15	98.24	97.60
unroll-loops	96.87	98.72	95.97	98.13	97.21

Table 2.6: Individual optimizations on su2cor

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	42.09	51.04	33.21	82.47	65.07
inline-func	42.06	51.01	33.21	82.46	65.11
sched-instr2	42.49	50.36	34.02	84.38	67.55
sched-instr	40.90	47.79	33.30	85.58	69.67
unroll-loops	40.17	48.35	31.17	83.08	64.46

Table 2.7: Individual optimizations on swim

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	30.06	36.64	20.02	82.02	54.64
inline-func	30.06	36.64	20.02	82.02	54.64
sched-instr2	30.91	36.39	20.53	84.92	56.41
sched-instr	29.83	35.11	20.32	84.95	57.86
unroll-loops	29.29	35.38	18.19	82.80	51.43

Table 2.8: Individual optimizations on go

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	40.97	42.75	42.65	95.83	99.77
inline-func	40.92	42.78	42.58	95.64	99.54
sched-instr2	43.07	44.01	45.25	97.87	102.82
sched-instr	43.52	44.89	46.52	96.96	103.63
unroll-loops	39.38	41.95	39.30	93.88	93.69

The *-fschedule-instr* optimization does simple basic block list-scheduling and *-fschedule-instr2* does aggressive global scheduling. Both options are expected to increase the IPC and hence the power. It can be seen that IPC goes up in most benchmarks, in some benchmarks up to 4.6% (in *su2cor*). Correspondingly, the power dissipation also increases considerably, in some benchmarks by as much as 10%. The aggressive scheduler (in option *-fschedule-instr2*) increases register pressure. Increased register pressure introduces a significant number of spill instructions, thereby increasing the total number of instructions executed and correspondingly the total energy consumed. The increase in number of instructions and energy consumption are up to 3.52% and 2.14% respectively. Thus, aggressive instruction scheduling not only increases processor power but also has a significant impact on total energy consumption. This optimization requires significant improvements before it can be used in systems where power dissipation and energy consumption are a concern.

Loop unrolling is a good optimization for reducing energy consumption because the number of instructions reduce significantly. However, reducing the energy does not necessarily reduce instantaneous power. For instance, in *li*, the power goes up by 10%. Unrolling increases the size of the basic block, allowing the compiler to increase the overlap of instructions. This leads to an increase in the number of simultaneous operations being executed; note that the IPC in *li* increases dramatically by 25%. However, this observation is not consistent among all the benchmarks, in many benchmarks, there is no increase in IPC. This is because the target architecture has a good branch predictor and automatically performs unrolling in hardware for all programs, hence diluting the impact of specific software unrolling optimizations. Function inlining reduces the number of instructions and hence the total energy consumption. However, in the benchmarks studied, only *go* shows a very marginal decrease in the number of instructions and energy consumption.

This simple study on the state-of-the-art performance optimizations makes two key observations. The first is that energy consumption reduces when the optimizations reduce the number of instructions executed by the program, *i.e.*, when the amount of work done is less. Optimizations such as common subexpression elimination, reduce the energy consumed while improving performance. Any new optimization technique can be effective in reducing energy only if unnecessary work can be eliminated. Additionally, another key observation made is that power dissipation is directly proportional to the average IPC or performance of the program. Optimizations such as instruction scheduling, which increase parallelism in the code, increase power dissipation. Hence, instruction scheduling algorithms need to be modified to be make them power-aware. In the following subsection, several compiler optimizations specifically designed for lowering power are examined.

2.2 Compiler optimizations for reducing power dissipation

There have been several instruction scheduling techniques suggested for reducing power dissipation in the processor core. Tiwari *et al.* [59] suggest enhancing the instruction selection module of code generators in the compiler. At compile-time, code generators accept an intermediate representation (in the form of a directed acyclic graph) of each basic block in the source code. Using pattern matching and dynamic programming, the compiler tries to find a cover for the directed acyclic graphs in terms of the specified instruction patterns in such a way that the overall cost is minimized. The cost function used in most compilers in the number of execution cycles, but Tiwari *et al.* modify it to include energy costs to obtain a code generator that targets energy consumption. Su *et al.* [56] proposed *cold scheduling*, wherein, instructions are assigned priority based on some pre-determined power cost and use a generic list scheduler to schedule the instructions. The power cost of scheduling an instruction depends on the instruction it is being scheduled after. This corresponds to the switching activity on the control path. Toburen *et al.* [60] propose another power-aware scheduler which schedules as many instructions as possible in a given cycle until the energy threshold of that cycle is reached. Once that precomputed threshold is reached, scheduling proceeds to the next time-step or cycle. The main disadvantage of these instruction scheduling techniques is that they are designed primarily for VLIW or in-order issue processors. These techniques are not applicable in modern out-of-order issue processors since instructions can be issued in any order. The ordering performed by the compiler is likely to be changed by the run-time hardware.

Significant work has been done in reducing energy consumption in the memory. Most techniques achieve a reduction in energy through innovative architectural techniques and compiler involvement [28, 55]. and [28]. In [28], the authors suggest the use of an L-cache. An L-cache is a small cache which is placed between the I-cache and CPU. The L-cache is very small (holds a few basic blocks), hence consumes less energy. The compiler selects appropriate basic blocks to place in the L-cache. These techniques reduce thus power dissipation in programs while delivering high levels of performance by matching hardware configurations and program regions. Another approach to reduce memory energy is Gray code addressing [55]. This form of addressing reduces the bit switching activity in the instruction address path. Bunda *et al.* [12] and Asanovic [4] investigated the effect of energy-aware instruction sets. These techniques would involve the compiler even earlier in the code generation process. The work by Bunda *et al* [12] concentrates on reducing memory energy, and Asanovic [4] investigates new instructions to reduce energy in the memory, register files and pipeline stages. These new instruction sets expose the compilers directly to the energy details of the processor and can help generate energy-efficient code at the outset of the compiling process. However, due to legacy code issues, introducing new instructions is not a widely accepted solution in the desktop and server processor markets.

2.3 Summary

The key observations from the study on the state-of-the-art performance optimizations are that while these optimizations are capable of delivering tremendous improvements in program performance, many of these optimizations are not always appropriate in systems where the design goals include reducing

power dissipation and energy consumption. Specifically, optimizations that typically enhance parallelism in programs (such as instruction scheduling and loop unrolling), notably increase power dissipation even though the total execution time of programs is reduced. Further, although there have been power-aware instruction schedulers suggested for VLIW and in-order issue processors, there have no techniques specifically suggested for out-of-order issue processors. Another observation made by this study is that optimizations such as common subexpression elimination, an optimization used to eliminate redundant computations in the program and reduce the energy consumed while concurrently improving performance.

The Hybrid-Scheduling approach uses these lessons to effectively reduce power in the out-of-order issue units of the processor. The scheme reuses the compiler-generated schedules in applicable regions and eliminates an inherent redundancy in the system and significantly reduces energy consumption in the processor. Further, by using low power issue hardware, the Hybrid-Scheduling technique reduces power without sacrificing performance. This technique is particularly applicable in out-of-order issue processors where many of the previously proposed power-aware compiler optimizations have been ineffective.

Chapter 3

The Hybrid-Scheduling Approach

This chapter provides a detailed description of the Hybrid-Scheduling scheme proposed in this dissertation. Hybrid-Scheduling synergetically combines the strengths of both compile-time instruction scheduling and dynamic scheduling to alleviate processor power and hardware complexity without significantly sacrificing performance. In the Hybrid-Scheduling paradigm, regions of code that can be scheduled well statically, issue and execute in the order prescribed by the compiler with minimal hardware support. In this scheme, programs are thus divided into low power "static regions", called *S-Regions* and high power "dynamic regions". This chapter first describes the basic qualities of an S-Region and develops the compiler analysis to quantitatively identify S-Regions. S-Regions require inherently less complex resources. The hardware requirements of S-Regions are discussed next. Finally, several unique advantages of the scheme are summarized.

3.1 S-Regions

An S-Region could be a basic block or a larger code region such as a loop, hyperblock [39], superblock [40] or subroutine.

There are several important criteria that must be satisfied before the

compiler can select a given program region as a candidate S-Region [63, 65].

- **The region should exhibit ILP (instruction level parallelism) that is visible and exploitable at compile-time**

  S-Regions bypass the dynamic scheduling logic, hence it is critical that the compiler generates schedules comparable to the schedules generated by the out-of-order issue logic for these parts of the code. The compiler typically makes conservative optimization decisions when there it has insufficient information regarding program control and data-flow. For example, hard-to-predict branches limit optimizations such as hyperblock, superblock formation, creating shorter basic blocks and thus leaving fewer opportunities to exploit parallelism. Code sequences without hard-to-predict branches thus make suitable S-Region candidates. Compilers are restricted in the amount of inter-procedural analysis and optimizations that can be performed. Regions without function/system calls are also examples of good S-Region candidates.

- **Regions with few false register dependences are desirable.**

  The number of registers exposed to the compile-time register allocator is limited by the number of bits available in the instruction format for operand encoding. Most architectures use 32 general-purpose registers. Due to the limited number of available registers, the register allocator often has to assign the same register to independent instructions. This reuse of registers could result in anti- and output- dependences, which in turn limit the instruction scheduler's reordering opportunities.

  Dynamically scheduled processors do not suffer from this limitation since they are able to employ dynamic register renaming to remap architec-

tural registers to hardware physical registers. With register renaming, each instruction entering the pipeline is assigned a new physical register. The number of hardware physical registers in a processor is considerably higher than the number of architectural registers exposed to the static scheduler (more than twice). Elimination of false-dependences allows the out-of-order issue scheduler greater flexibility in selecting instructions for issue each cycle. Regions that have no or few false dependences are good S-Region candidates, since the quality schedule of the compiler schedule for such regions is comparable to that of the schedule created by the run-time hardware.

- **Regions with few memory aliases are desirable.**

  The compiler must make conservative decisions when there are unresolved memory addresses in the code. When the compiler can not resolve the addresses of two memory operations, it adds a dependence edge between the operations and forces them to execute sequentially. Since out-of-order issue processors typically employ run-time memory disambiguation, a given static schedule may perform poorly compared to its equivalent dynamic schedule if the compiler respects dependences that actually do not occur during execution. Memory aliases limit the available ILP at compile-time and thus regions with a large number of aliases are not suitable S-Region candidates.

- **Regions should have few memory misses**.

  Due to the absence of dynamic scheduling in static mode, it is difficult to hide cache miss latencies by instruction overlap. Regions with no memory misses or regions with regular access patterns that are amenable

to techniques such as prefetching are thus suitable candidates for static mode issue.

- **Regions should contribute significantly to the execution time**.

  The Hybrid-Scheduling scheme introduces a few additional hardware structures to the conventional superscalar processor to support low-power issue of S-Regions. This overhead of additional hardware, the potential performance loss due to use of simpler hardware, and the compilation time overhead are not justified if the program spends an insignificant amount of the program execution time in S-Regions. In such cases, the energy reduction will not be sizable and further, the overheads of the scheme might in fact degrade performance and/or power dissipation significantly.

3.2 Identifying S-Regions through quantitative analysis of region schedule quality

The role of the compiler in the Hybrid-Scheduling scheme is to identify basic-blocks with inherently low dynamic reorder requirements. As noted in the previous section, for a given basic-block (*i.e,* a sequence of data-dependent instructions with no control-flow dependence between them), there are three primary impediments to achieving good schedules at compile-time: (a) false-dependences (b) unresolved memory aliases and (c) non-uniform load latencies. Instructions within such blocks will require dynamic reordering to hide pipeline stalls. This section describes in detail how the compiler, with the help of profile-time statistics, evaluates the impact of the above constraints on the schedule quality of a block.

3.2.1 Impact of anti- and output- dependences on static schedules

A critical path in a basic-block is defined as the path wherein instructions exhibit zero scheduling freedom or *slack*. Critical paths in the block are computed by considering only true data dependences. The longest critical path in a basic-block represents the minimum schedule that can be achieved by the compiler. The degradation of the schedule quality due to anti- and output-dependences can be estimated by comparing the length of this path to the actual schedule achieved by the static instruction scheduler. If the anti- and output- dependence edges impact the schedule, the schedule length of the block ($Sched\_len$) will be larger than the maximum critical path (CP_{max}). Such a block will clearly require dynamic reordering support. The degradation in schedule quality caused by anti- and output- dependences (SD_{fd}) can be estimated as:

$$SD_{fd} = \frac{Sched\_len - CP_{max}}{CP_{max}} * 100 \tag{3.1}$$

Note, Equation 3.1 conservatively assumes that with a sufficiently large physical register file, the dynamic issue logic can remove all the false dependences.

3.2.2 Impact of unresolved alias edges

The impact of aliases on the static schedule is estimated by profiling all load and store dependences in a basic-block. All the memory dependences within a block that occur during program execution are profiled and for every pair

of memory operations that are dependent each time they occur during execution (*always_occurs*), the memory dependence edge is converted to a true data-dependence edge during the computation of the critical path. The critical path then reflects this true memory dependence. Consequently, Equation 3.1 estimates the potential schedule degradation due to both false data dependences and false memory dependences.

3.2.3 Impact of cache misses

Load misses usually pose a serious performance bottleneck to in-order issue of instructions since it is not possible to hide the empty cycles with other independent instructions. Even if the load miss is known at compile-time, it is difficult for the static scheduler to find sufficient additional instructions in a single basic block to fill all issue slots, particularly since modern superscalar processors have very wide issue and large cache miss latencies. The impact of cache misses (SD_{cm}) on a given basic-block schedule is estimated using profile data on the average number of L1 misses ($L1_{misses}$) and L2 misses ($L2_{misses}$) per block. The estimated performance degradation is given by:

$$SD_{cm} = \frac{L1_{misses} * L1_{effective\_cost} + L2_{misses} * L2_{effective\_cost}}{CP_{max}} * 100 \qquad (3.2)$$

The *effective costs* in Equation 3.2 represent the fraction of the cache miss latency that the out-of-order issue logic can hide. The effective cost is different from the actual latency of a cache miss (usually lower). It is a complex function of miss latencies, issue queue size, issue width and available ILP in the program. Larger issue queues can potentially hide a larger fraction of the miss stalls. The effective costs are identified empirically in this work.

3.2.4 Selecting and annotating S-Regions

After the estimated schedule degradation due to false dependences (SD_{fd}) and cache misses (SD_{cm}) are computed, a block can be selected as a static block if both SD_{fd} and SD_{cm} are less than certain threshold values, FD_{th} and CM_{th} respectively. These threshold values are the tuning handles for the block selection heuristics and represent the acceptable performance degradation levels tolerated by the user.

There are several means to convey the block reordering requirement to the underlying microarchitecture. Most processors have a few unused instructions that can be used as marker instructions. For example, the Alpha processor has unused floating point load operations that can be used as marker instructions. Additionally, it is also possible to directly encode the annotations into the unused bits of each instruction, which is particularly feasible in processors that use 64-bit instructions. This will completely eliminate the need for special marker instructions. This work assumes that the compiler annotations are conveyed to the hardware using marker instructions. Figure 3.1 shows a summary of the block selection heuristics.

3.3 Hardware support for S-Regions

The previous section showed how the compiler can identify blocks that require less dynamic scheduling. This section presents a novel issue queue design that exploits the varying requirement of basic blocks to significantly alleviate the complexity and power of the issue logic.

Instructions belonging to static regions, *i.e.,* blocks for which the compiler can perform near-optimal scheduling can potentially be issued in their

```
Profile Collection Phase
for each basic block {
  record L1 miss rate, L2 miss rate;
  record pairs of dependent memory operations
}

Profile Analysis Phase
for each basic block {
    for each 'always_occurs' memory edge {
        convert to true dependence edge;
    }
    Compute CP_max;
    Compute SD_fd;
    Compute SD_cm;

    if (SD_fd < FD_th && SD_cm < CM_th)
        Select block as static block;
    else
        Select block as dynamic block;
    Annotate block with marker instruction;
}
```

Figure 3.1: Block selection algorithm

statically scheduled order. However, in today's wide-issue processors, it is not sufficient if the instructions within basic blocks are scheduled perfectly. Instructions from different blocks must be overlapped to fill the available issue slots. Thus, instructions in a program require two forms of reordering namely *intra-block reordering* and *inter-block overlap*. Static regions do not require intra-block instruction reordering but require inter-block overlap to limit performance loss. Blocks where the compiler is limited by artificial dependences and memory misses will require both intra-block and inter-block reordering. Based on the inherent reordering requirement, static regions or blocks can also be called *Low Reorder Required* (LRR) blocks. All blocks requiring both intra-

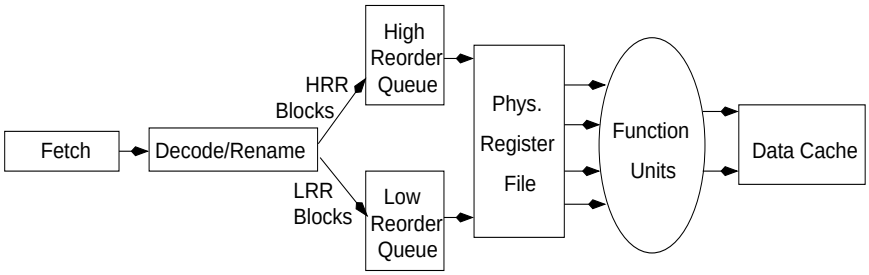


Figure 3.2: High-level view of the proposed Hybrid-Scheduling scheme

and inter-block reordering, *i.e.,* blocks corresponding to dynamic regions are called *High Reorder Required* (HRR) blocks.

In the Hybrid-Scheduling scheme, there are two separate issue queues for LRR and HRR blocks. Blocks are directed to different queues based on their reordering requirement, *i.e.,* the issue logic is now Reorder-Sensitive [65]. Since HRR blocks require considerable reordering, conventional dynamic issue logic which is typically fully associative is most appropriate for these blocks. LRR blocks however use a novel low complexity, low power FIFO based that provides only inter-block overlap. The different issue queues for LRR and HRR blocks are called the Low Reorder (LR) issue queue and the High Reorder (HR) issue queue respectively. In each cycle, instructions are selected from either the out-of-order issue queue or the heads of the FIFOs for execution. The total number of instructions issued in each cycle is limited to the issue width of the processor. Figure 3.2 presents a high-level view of this proposed scheme.

3.3.1 Low reorder issue queue

The LR queue as shown in Figure 3.3, consists of several FIFO buffers, with each buffer as wide as the average ILP available in a basic block. At run-time, each new LRR block is directed to one of these buffers. Instructions can be

written only into the tail pointer of each buffer and can only be read from the head of the queue. In each cycle, instructions from the heads of the FIFOs can be selected for execution. Since there are multiple queues, instructions from different LRR blocks can be overlapped (Figure 3.3). Instructions in each basic block are therefore issued in their statically-scheduled order but are overlapped with instructions from successive basic blocks. For each LRR block, a FIFO is selected in a round-robin fashion. Each new block is directed to a different FIFO. The number of FIFOs can be chosen at design time based on the extent of overlap seen during execution for the target workloads. For example, experiments on an 8-way conventional out-of-order issue processor with a 128-entry issue queue, revealed that in nearly 70% of execution cycles, instructions are issued from at most three consecutive basic blocks. Three FIFO buffers will be able to thus capture a significant portion of the required inter-block overlap between LR blocks.

The LR queue is conceptually similar to the region-slip-enabled issue buffer proposed by Spadini *et al.* [54]. Their proposed mechanism uses a FIFO-based issue buffer that allows a block's schedule to 'slip' into the schedule of a previous block. However, the disadvantage of the region-slip buffer is that it is a monolithic structure requiring a large number of entries and ports, making the power consumption of the buffer excessively high. Further, similar to other data-prescheduling based issue queues [1, 15, 20, 41, 42, 46, 48, 37], the region-slip buffer requires additional pipeline stages to identify a suitable buffer entry for each instruction. In contrast, the proposed LR queue is designed to be low-complexity and low-power.

Another approach to achieving inter-block overlap is to employ compiler optimizations such as trace scheduling [21], hyperblock scheduling [39] or superblock scheduling [40]. However, as shown by Spadini *et al.* [54], the

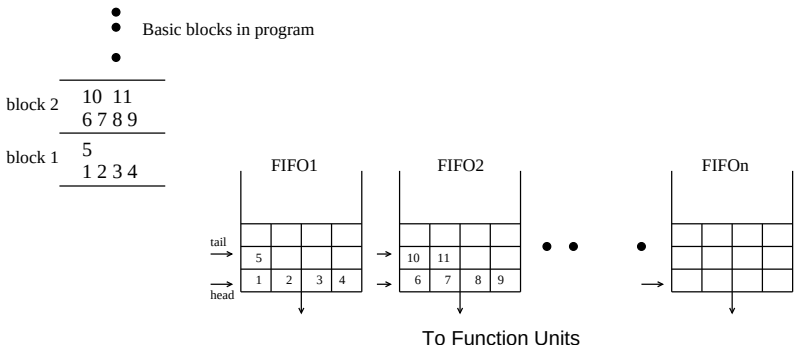


Figure 3.3: Low-Reorder (LR) issue queue

cycle-time degradation due to block boundaries remains unacceptably high despite aggressive optimizations. Hence, a hardware hardware approach is more suitable.

3.3.2 Power and complexity analysis of the low reorder issue queue

The power consumption of the LR buffers is significantly low. Each FIFO requires a small number of ports (typically 2) since instructions are written only into the head pointer location and read from the tail pointer. The complexity of the LR queue is also inherently low since it is FIFO based. There are no complex CAM structures or global signals for instruction wake-up. Instructions at the heads of the FIFOs check a small table to see if their operands are ready [46]. Alternatively, in compacting-style queues, only the head entry of the issue queue can have CAMs to compare operand tags [58, 68].

A majority of the previously proposed techniques reduce complexity of the issue queue by limiting the number of candidate instructions to be consid-

ered for issue [1, 15, 20, 41, 42, 46, 48, 37]. These techniques typically consist of a *pre-scheduling* phase wherein the data-dependences of instructions are analyzed. Instructions are typically held in a separate buffer and are considered for issue in their approximate data-flow order [1, 15, 20, 41, 42, 48, 37]. In some cases, after the pre-scheduling phase, instructions are steered to different low-complexity FIFOs based on their dependences with older instructions in the queues [1, 46, 48]. While these techniques indeed alleviate the complexity of the issue logic, they often require extra hardware and/or the addition of a few pipeline stages. In the Hybrid-Scheduling scheme, since the necessary analysis is performed statically, there are no extraneous hardware structures or pipeline stages required for steering instructions to different queues.

3.3.3 High reorder issue queue

The HR queue in the reorder-sensitive issue queue caters to instructions that require considerable dynamic reordering and hence is fully associative similar to the conventional out-of-order issue queue. However, note that in the block selection heuristics, blocks with a large number of cache misses are automatically deemed as HRR blocks. Consequently, the available ILP in these blocks is limited. The HR buffer can thus be small and have a less aggressive issue width. Since the wakeup logic has a quadratic dependence on issue width, the effective complexity of the HR queue is greatly reduced.

Thus, by identifying inherent requirements of different basic-blocks in a program, the reorder-sensitive issue logic can use smaller and simpler hardware structures for energy-efficient processing of instructions. The following section summarizes several advantages of the Hybrid-Scheduling scheme.

3.4 Advantages of the Hybrid-Scheduling approach

The Hybrid-Scheduling scheme offers several key benefits.

- Removes inherent redundancy in the system

  In processors with dynamic scheduling logic, the hardware searches for parallel instructions, *irrespective* of whether the compiler-generated schedule is perfect or not. The Hybrid-Scheduling architecture uses aggressive and power-hungry scheduling hardware for program regions that warrant it, while facilitating low energy, low complexity execution for structured, regular regions that do not require such hardware.

- Allows us to apply hardware power saving techniques orthogonally

  Another favorable aspect of the proposed scheme is that during the extremely low ILP phases of the program, several previously proposed dynamic resizing schemes [5, 22, 47, 30, 16, 61, 31] can be applied orthogonally to the HR queue (associative queue) for larger overall energy savings in the processor. Further, Chapter 6 shows how the Hybrid-Scheduling scheme can be particularly effective in regions where the recently proposed dynamic microprocessor resource adaptation schemes, such as: [14][22][25][30][47], have been less effective.

- Does not require a significant change in the ISA

  The Hybrid-Scheduling scheme does not require a significant change in the instruction set architecture (ISA) of the underlying processor. At the most, one new instruction will need to be introduced (marker instruction) to convey the block classification to the hardware.

- Highly suitable for clustering

  Hybrid-Scheduling is naturally amenable to clustering. Since basic blocks are a logical sequence of dependent instructions, the maximum number of data dependences are expected to occur between instructions within a basic block. Thus, the LR queue and the HR queue can be separated into different clusters without significantly impacting performance.

- LR and HR queues can be optimized separately.

  An important advantage of the split-issue scheme proposed is that the two queues can be optimized separately. For example, each cluster can be clocked at different frequencies. The LR queue is extremely simple and can clock faster than a large monolithic associative queue. An additional benefit is that LRR blocks contain fewer cache misses. Hence, the relative increase in memory latency due to increased processor clock frequency will not adversely affect the performance.

3.5 Summary

This chapter presented the complete compiler-level and micro-architectural details of the Hybrid-Scheduling scheme. Subsequent chapters present a detailed evaluation of this technique and show that it can lead to substantial improvement in processor energy consumption with only a minimal loss in performance.

Chapter 4

SPHINX: A Combined Compiler/Simulator Framework

This dissertation develops a unique framework called **SPHINX** to evaluate the proposed Hybrid-Scheduling scheme. SPHINX integrates a detailed out-of-issue processor simulator that is largely based on Simplescalar's *sim-outorder* simulator with the Trimaran 2.0 [72] compiler framework. The SPHINX simulator incorporates power models derived from Wattch [11] to estimate power dissipation in the processor. The SPHINX framework thus provides a unified platform for exploring a range of software and hardware techniques for low power.

4.1 The SPHINX compiler

An overview of the SPHINX tool is shown in Figure 4.1. SPHINX has been developed within the Trimaran framework. Trimaran is a compiler infrastructure for supporting state-of-the-art research in compiling for Instruction Level Parallel (ILP) architectures. Trimaran is oriented towards EPIC (Explicitly Parallel Instruction Computing) architectures, and supports compiler research in what are typically considered to be "back end" techniques such as instruction scheduling, register allocation, and machine-dependent optimizations.

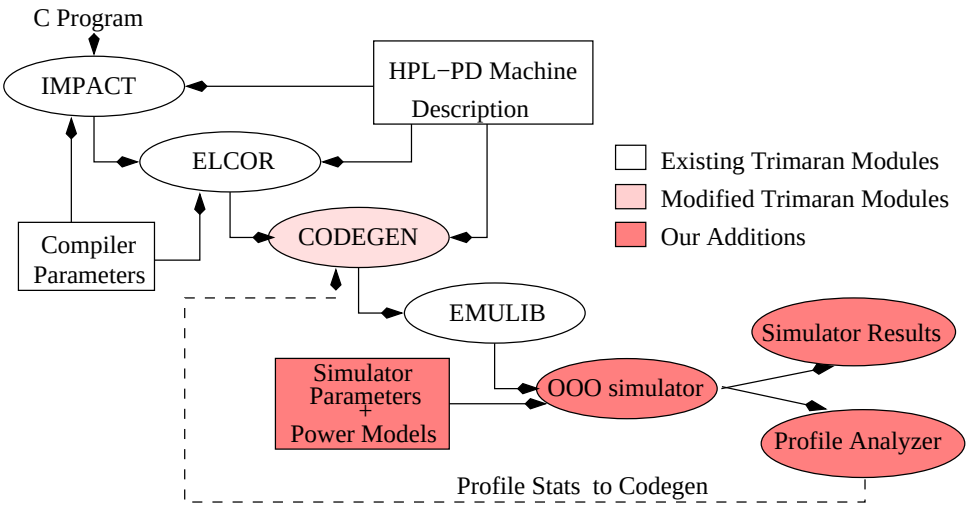


Figure 4.1: SPHINX compiler/simulator framework

The Trimaran compiler infrastructure comprises of the following components:

- A machine description facility, *mdes*, for describing ILP architectures.
- A parameterized ILP architecture called HPL-PD.
- A machine description language called HMDES that allows the user to develop a machine description for the HPL-PD processor in a high-level language.
- A compiler front-end (IMPACT) for C that performs parsing, type checking, and a large suite of high-level (i.e. machine independent) classical and instruction-level parallelism optimizations.
- A compiler back-end (ELCOR) that performs instruction scheduling, register allocation, and machine-dependent optimizations.

- An extensible intermediate program representation (IR) which has both an internal and textual representation, with conversion routines between the two. The textual language is called "Rebel". This IR supports modern compiler techniques by representing control flow, data and control dependence, and many other attributes.

- An integrated Graphical User Interface (GUI) for configuring and running the Trimaran system. The GUI includes tools for the graphical visualization of the program intermediate representation and of the performance results.

The Trimaran infrastructure is typically used for designing, implementing, and testing new compilation modules to be incorporated into the ELCOR back end. These new modules may augment or replace existing ELCOR modules. Although there are several compiler infrastructures available to the research community, Trimaran is especially useful for the following reasons:

- It is especially geared for ILP research.

- It provides a rich compilation framework. The parameterized ILP architecture (HPL-PD) space allows the user to experiment with machines that vary considerably in the types of functional units and register files modeled.

- The modular nature of the compiler back end and the single intermediate program representation used throughout the compiler back-end (ELCOR) makes the construction and insertion of new compilation modules into the compiler especially easy.

- The framework is already populated with a large number of existing compilation modules, providing leverage for new compiler research and

supporting meaningful experimentation. Some of the important optimization modules include loop unrolling, hyperblock formation [39], superblock formation [40] and predication [3].

4.1.1 Configuring Trimaran to support superscalar execution

The HPL-PD machine description mainly supports EPIC (Explicitly Parallel Instruction Computing) style instruction execution. An example of a processor family which employs this design philosophy is the Itanium family of processors [71]. In EPIC, the compiler decides and explicitly indicates to the hardware which instructions will be executed together. Individual instructions are arranged in 'bundles' or 'packets' by the compiler. All the instructions in a bundle are guaranteed to be independent and can be issued to the function units in parallel. A bundle is typically called a *MultiOP* instruction in the HPL-PD machine description.

HPL-PD supports two forms MultiOP instructions, namely: *MultiOP-P* and *MultiOP-S*. The *MultiOP-P* semantics requires that the correct execution is guaranteed only if all the operations in the instruction are issued simultaneously. The compiler can schedule code with the assurance that all operations in one instruction will be issued simultaneously. For instance, the compiler can schedule two mutually anti-dependent copy operations, which together implement an exchange copy, in the same instruction. Without this assurance, the exchange copy would have had to be implemented as three copy operations that require two cycles. Thus, the MultiOP-P semantics permits admissible dependences between operations (*i.e.,* anti- and output- dependences) to be bi-directional across the instruction.

MultiOP-P semantics pose a problem with respect executing instructions sequentially and also inhibit compatibility across a family of machines with differing amount of functional units. When code that was generated for a machine with a certain width (*i.e.,* number of functional units) has to be executed by a narrower machine, the narrow processor must necessarily issue the MultiOP instructions semi-sequentially one portion at a time. Unless care is taken, this could potentially violate MultiOP-P semantics and lead to incorrect results. For example, if the aforementioned copy operations are issued at different times, the intended exchange copy is not performed.

The other variation, called *MultiOP-S* semantics, simplifies sequential execution by excluding bi-directional dependences across a MultiOP instruction. MultiOP-S instructions can still be issued in parallel, but they can also be issued sequentially from left to right. The compiler ensures that admissible dependences between operations in a MultiOP-S instructions occur only from left to right. The MultiOP-S instruction style which will effectively generate a sequential stream of instructions is used for supporting a superscalar machine in the SPHINX framework.

Further, the HPL-PD machine description language (HMDES) can be used to define a contemporary superscalar processor. In Trimaran, a multitude of machine description aspects can varied including the register file type and size, number and type of function units, operation latencies etc. For example, Table 4.1 shows how some of the important high-level HMDES parameters can be used to define a typical 8-issue superscalar processor.

As seen in Table 4.1, the HPL-PD machine supports different register file types including conventional types such as the general-purpose and floating point. Additionally, HPL-PD allows four other types of register files:

- **Predicate Registers:** Predicated or guarded execution refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. Predicated execution is an efficient method typically used in VLIW and EPIC architectures to handle conditional branches present in a program. Many superscalar architectures also allow some form conditional execution. For example, the conditional move instructions facilitate conditional execution in the Alpha family of processors. HPL-PD provides support for predicated execution with a predicate register file (single-bit entries), several forms of predicate defining instructions and predicated versions of all instructions. The SPHINX simulator provides the micro-architectural support required for predicated execution.

- **Rotating Registers:** The HPL-PD architecture supports both static and rotating registers. Most register files are partitioned into static and rotating portions with differing numbers of registers. The static registers are conventional registers; the rotating registers logically shift in register address space every time the rotating register base (RRB) is decremented by certain loop-closing branches. Rotating registers are typically used for efficient implementation of software pipelining in the processor. The SPHINX simulator currently does not support rotating registers.

- **Branch Target Registers:** In the HPL-PD architecture, branches are performed in multiple steps. Special *prepare-to-branch* (PBR) instructions that hold the target address of a branch in one of the branch target registers usually precede a branch instruction. These instructions can potentially be used to provide hints regarding branch prediction to the target architecture. BTR registers are accessed only by PBR

instructions. The BTR file contains only replicates of values that are already held in the general-purpose register (GPR) file; it does not extend the GPR file in any way. Further, branch hints are not supported in SPHINX.

- **Control Registers:** Control registers provide a uniform scheme to access internal state within the processor. Some of the control registers are the PC (instruction counter), PSW (Processor Status Word) and RRB (Register Relocate Base for rotating registers). A complete list can be found in the HPL_PD technical report [32].

Table 4.1 shows how the Trimaran HMDES parameters can be used to define a superscalar processor. The Trimaran modules use these parameters for implementing machine-specific optimizations.

There are certain HPL-PD features that are not currently supported in the SPHINX simulator. They include branch hints, rotating registers, memory transfer instructions and data speculative instructions (LDV and SDV instructions).

4.2 The SPHINX simulator

The code generator module CodeGen in SPHINX (Figure 4.1) converts the program from its intermediate program representation into instructions that can be executed on a virtual HPL-PD machine. The emulation library Emulib of Trimaran contains an interpreter and a set of emulation routines for the HPL-PD virtual machine. The interpreter generates a sequential trace of instructions that feed into a detailed out-of-order issue simulator. The out-of-order simulator (OOO Simulator) in SPHINX is a heavily modified version

Table 4.1: High-Level HMDES parameters

Feature	Attributes
Integer GPRs	32
Floating Point GPRs	32
Predicate Registers	32
Branch Target Registers	16
Rotating Registers	None
Control Registers	None
Issue Width	8
Function Units	6 IALU, 1-cycle latency
	2 FPALU, 3-cycle latency
	4 IMULT, 2-cycle latency
	1 FPMULT, 10-cycle latency
	2 LD/ST, 2-cycle latency

of Simplescalar's *sim-outorder* simulator. The important differences between *sim-outorder* and the SPHINX implementation are as follows:

1. Sim-outorder uses a combined issue queue and reorder buffer organization called Register Update Unit (RUU) for tracking instructions through the pipeline. The RUU is replaced with a separate issue queue and physical register file as found in contemporary architectures such as the Pentium 4 processor [29].

2. BTR, predicate and control register files are included in the architecture.

3. Support for squashing predicated instructions in the issue stage if their predicate operand is false, has been added. Additionally, for predicated code executing on a dynamically scheduled machine, it is possible to have multiple instructions guarded by different predicate registers writing into one architectural register [67]. In the simulator, it is assumed that such conflicts are perfectly resolved.

The SPHINX simulator further incorporates the branch predictor and cache modules used in *sim-outorder*. These modules can be used to define a variety of complex, multi-level branch predictors and cache hierarchies.

4.2.1 Power estimation in SPHINX

The power/energy estimates in the simulator are based on the suite of parameterizable power models in Wattch 1.0 [11]. In CMOS microprocessors, dynamic power consumption (P_d) is the main source of power consumption, and it is defined as: $P_d = CV_{dd}{}^2af$, where, C is the load capacitance, V_{dd} is the supply voltage and f is the clock frequency. The activity factor a, is a fraction between 0 and 1 indicating how often clocks lead to switching activity on average. Wattch power models are estimates of the capacitance values of different micro-architectural units based on the circuit style and the transistor sizings. V_{dd} and f depend on the assumed process technology. The power models are provided for the .35um process technology. Models for different process technologies can be obtained by using appropriate scaling factors provided in Wattch. Power models for different microarchitectural units are constructed using one or more of the following parameterizable models: RAM-based array structures, CAM-based array structures, complex logic blocks and clock. For example, the issue queue structure is constructed using RAM models for holding the data and CAM models for tags. More details on these models can be found in the Wattch technical paper [11]. The activity factor is set based on the circuit-style. For circuits that pre-charge and discharge on every cycle (i.e., double-ended array bitlines) an a of 1 is used. The activity factors for certain critical subcircuits (i.e., single-ended array bitlines) are measured from the benchmark programs using the architectural simulator. For complex circuits,

where it is difficult to measure activity factors with a high-level simulator, a base activity factor of 0.5 is assumed.

Wattch also supports three different options for clock gating to disable unused resources in the processor. Ideal clock gating assumes that the maximum power will be dissipated if any access occur in a given cycle, and zero otherwise. The second possibility assumes that if only a portion of a unit's port are accessed, the power is scaled linearly according to the number of ports being used. In the third clock gating scheme, power is scaled linearly with port or unit usage, but unused units dissipate 10% of their maximum power. In this dissertation, power and energy results corresponding to the third scheme are always chosen since it is the most realistic of all schemes [11]. The Wattch power models interface with the cycle-accurate simulator through resource usage counts. The simulator keeps track of the units accessed in each cycle to estimate the total energy consumed.

The SPHINX simulator retains all the statistics generated by sim-outorder and Wattch and further adds several statistics relevant to the Hybrid-Scheduling scheme. SPHINX also consists of a profile analysis phase (Profile Analyzer) which examines the profile statistics collected and analyzes the data. For example, in the Hybrid-Scheduling scheme, the profile analyzer estimates the degree of reordering required by each basic-block in the program. This information is provided to the CodeGen module, which in turn annotates blocks with appropriate marker instructions.

4.3 Summary

This chapter described the SPHINX evaluation tool in detail. SPHINX integrates a compiler, micro-architectural simulator and power models into a single

platform enabling future research in a variety of areas including software, hardware and cooperative hardware/software techniques for both performance and power.

Chapter 5

Experimental Evaluation of the Hybrid-Scheduling Scheme

This chapter presents an evaluation of the Hybrid-Scheduling scheme using the SPHINX framework. The chapter first describes the benchmarks and baseline processor configurations. The Hybrid-Scheduling scheme can potentially lead to sizable energy savings in the processor core without significantly sacrificing performance. Power dissipation and performance results of the scheme along with a detailed evaluation of the compiler heuristics for selecting program regions with dynamic scheduling requirements are also provided.

5.1 Benchmarks

General-purpose integer programs are less amenable to compile-time optimizations due to the presence of hard-to-predict branches, pointer-intensive memory accesses and extensive use of function and library calls. The Hybrid-Scheduling scheme is thus evaluated for several integer benchmarks from the SPEC CPU benchmark suite [73]. Table 5.1 shows the benchmarks and the input sets used. All SPEC integer benchmarks that compiled and ran successfully on Trimaran were chosen. The MinneSPEC reduced input set [35] was used where applicable. The MinneSPEC workload is derived from the standard SPEC CPU 2000 workload and allows computer architects to ob-

Table 5.1: Benchmarks and inputs.

Benchmark	Profile Input Set	Inputs for Perf. Evaluation
compress	10000 q 2131	220000 q 2131
gzip	mgred.random (train)	lgred.random (ref)
li	SPEC test	SPEC train
mcf	mgred.in (train)	lgred.in (ref)
parser	2.1.dict mgred.in (train)	2.1.dict lgred.in (ref)
vortex	SPEC test	SPEC ref
vpr	place mgred.net (train)	place lgred.net (ref)

tain simulation results in a reasonable time using existing simulators. For the benchmarks studied, the MinneSPEC profiles closely match the SPEC reference dataset program behavior. MinneSPEC inputs are not available for the spec95 benchmarks and for *vortex*, the little-endian version of the input set was not provided. The input data used by the compile-time analyzer to classify blocks is different from the input set used to evaluate the proposed technique, *i.e., true* profiling is used. For all benchmarks, a maximum of one billion instructions are simulated.

5.2 Processor configurations

The configuration of the baseline processor is given in Table 5.2. The out-of-order issue processor considered consists of an 8-wide, 128 entry issue queue. Power models corresponding to the 0.18μ process at a 2V supply voltage and 1GHz operating frequency are employed in the simulator. Unused units dissipate 10% of their maximum power [9].

Power distribution in different hardware structures in the baseline processor are shown in Tables 5.2 and 5.3. The power breakdowns in Table 5.2

Table 5.2: Baseline out-of-order issue processor configuration

Feature	Attributes
Issue Units	IQ - 128 entries LSQ - 64 entries Physical registers - 128 Fetch/Decode/Issue/Commit width - 8
Cache Hierarchy	32KB 4-way L1 Dcache (2-cycle hit) 32KB DM L1 Icache (1-cycle hit) 512KB 4-way L2 (20-cycle hit)
Memory	150 cycles memory latency
Branch Pred.	4K Gshare 15 cycles misprediction latency
Function units	6 integer ALUs, 2 FP ALUs , 4 integer multiply units, 1 FP multiply units, 2 load/store units

represent the maximum power per unit. The maximum processor power for the baseline configuration is 83W. The issue queue dissipates a peak power of 18W, approximately 23% of the total power (this includes the power dissipated in the clock nodes of the queue). Table 5.3 shows the average power distribution among different structures based on the activity factors for the benchmark *gzip*. Activity-based power dissipation also shows that the issue queue expends a significantly large part of the total processor energy budget. The issue queue accounts for nearly a quarter (24.7%) of the total energy consumed on the processor [11] [22] [26].

The baseline Hybrid-Scheduling configuration has three 4-wide, 8-entry FIFO-based LR buffers and an HR queue (see Table 5.5). The width of each FIFO is restricted to four entries since in most blocks, the average ILP was not higher than 4. The number of FIFOs was determined by measuring the degree of overlap in the SPEC integer benchmarks. Table 5.6 shows the maximum distance between instructions issued in one cycle in terms of the number of basic blocks. The results reveal that in nearly 70% of execution cycles,

Table 5.3: Power distribution for different hardware structures in the baseline processor. The power breakdowns represent the maximum power per unit.

Unit	Power	Unit	Power
BPred	3.40%	LSQ	2.64%
IQ	16.5%	Rename	0.80%
Reg. File	11.1%	Res. Bus	6.2%
Func. Units	10.8%	ICache	3.03%
DCache	6.5%	Clock	35.1%
L2 Cache	3.6%	Total	100%

Table 5.4: Activity-based power distribution for *gzip*.

Unit	Power	Unit	Power
BPred	4.62%	LSQ	2.0%
IQ	18.2%	Rename	0.9%
Reg. File	13.5%	Res. Bus	6.3%
Func. Units	5.9%	ICache	4.8%
DCache	7.5%	Clock	35.5%
L2 Cache	0.1%	Total	100%

instructions are issued from at most three consecutive basic blocks. Hence, three FIFOs should be sufficient to provide significant the overlap between blocks. The baseline reorder-sensitive issue queue has 3 FIFOs. In subsequent sections, power-performance trade-offs for several other configurations of the reorder-sensitive issue queue are also evaluated.

The HR queue in the reorder-sensitive issue logic is an out-of-order issue queue to allow aggressive reordering of instructions in HRR blocks. Due to high cache miss rates of the blocks issued to the HR issue queue, these blocks have inherently low ILP and hence do not require the full capability of an 8-way out-of-order issue queue. A 4-wide, 32-entry associative HR queue is used in these experiments. In contrast to the baseline processor, 75% of the entries in the reorder-sensitive issue logic are thus in FIFOs and the remaining

Table 5.5: Baseline reorder-sensitive issue queue parameters

Feature	Attributes	Values
LR Queue	Number of FIFOs	3
	IW of FIFOs	4
	Rows per FIFOs	8
	Number of Read Ports/FIFO	1
	Number of Write Ports/FIFO	2
	Peak Power per FIFO	0.7W
HR Queue	Issue Width	4
	Number of Entries	32
	Peak Power of HR Queue	3.6W

Table 5.6: Instruction overlap from different blocks

# of blocks	≤ 1	≤ 2	≤ 3	≤ 4	≤ 5	≤ 6	≤ 8
% cycles	52.6	61.4	70.1	74.7	79.7	82.7	100

25% of the entries are in a 4-wide, 32-entry associative issue queue. Note that all the queues/buffers in the reorder-sensitive issue queue are significantly less aggressive and smaller when compared to the baseline out-of-order issue queue.

5.3 Distribution of LRR and HRR blocks

Table 5.3 shows a detailed characterization of the benchmarks in terms of LRR and HRR blocks. The block selection thresholds were set to 5%(FD_{th}) and 0.1%(CM_{th}). Section 5.10 presents an evaluation of the technique for several other threshold values. Also note, the effective L1 and L2 miss costs are fixed at 30 cycles (Section 5.10 also presents an evaluation of the technique for several other effective costs).

The first three rows in Table 5.3 show the number of blocks statically

Table 5.7: Static distribution of LRR and HRR blocks for threshold values $FD_{th} = 5\%$ and $CM_{th} = 0.1\%$.

Benchmark	compress	gzip	li	mcf	parser	vortex	vpr
# of blocks where $SD_{cm} > SD_{th}$	44	307	255	155	1156	1165	163
# of blocks where $SD_{fd} > FD_{th}$	51	290	539	93	1599	3287	736
# of blocks failed due to both	20	50	103	51	419	693	74
# of HRR blocks	99	581	967	229	2666	5469	1122
# of LRR blocks	93	307	277	160	1647	3113	728

classified as HRR blocks due to either a large number of cache misses, due to a large number of false dependences or due to both impediments, respectively. The following two rows provide the total number of HRR and LRR blocks (static blocks) and show that even with low block selection thresholds, a significant number of blocks qualify as LRR blocks.

Table 5.8 shows the distribution of dynamic instructions in terms of HRR and LRR instructions. The results reveal that on average, 32% of all dynamic instructions are directed to the LR queue (last row).

Table 5.8: Dynamic distribution of LRR and HRR blocks for threshold values $FD_{th} = 5\%$ and $CM_{th} = 0.1\%$.

Program Time	compress	gzip	li	mcf	parser	vortex	vpr
In blocks where $SD_{cm} > SD_{th}$	48.47%	45.66%	49.85%	58.89%	62.06%	57.87%	4.41%
In blocks where $SD_{fd} > FD_{th}$	40.75%	25.45%	42.49%	9.84%	46.91%	62.25%	48.14%
In HRR blocks failed due to both	21.79%	9.96%	23.92%	7.45%	31.05%	42.74%	2.30%
In HRR blocks	67.66%	62.55%	69.97%	61.28%	78.00%	78.15%	54.69%
In LRR blocks	32.34%	37.45%	30.97%	38.72%	22.00%	21.85%	45.31%

5.4 Energy consumption results

Figures 5.1 and 5.2 show normalized issue queue energy consumption and total energy consumption of the Hybrid-Scheduling scheme with respect to the baseline out-of-order issue queue. The Hybrid-Scheduling issue scheme dramatically reduces the issue queue energy consumption on an average by nearly 70%. Several factors contribute to the overall energy reduction.

First, as seen in Table 5.8, nearly 32% of the instructions are issued from the LR FIFOs. The FIFOs consume low power since they require very few ports. Since the fetch width of the processor is twice that of the FIFO width, two write ports are required to dispatch instructions to each FIFO. Further, since instructions are issued only from the head of the FIFO, it needs only one read port. The peak power of each FIFO buffer is 0.7W while the peak power of the 8-wide issue queue is 18W. The 32-entry HR queue in the reorder-sensitive architecture is also significantly smaller than the baseline 128-entry issue queue. More importantly, the issue width of the HR queue is half that of the baseline issue queue. The HR queue thus has significantly fewer number of ports than the baseline 8-wide issue queue. The number of dispatch (*i.e.*, writing to the issue queue) ports on the HR queue is the same as in the baseline queue since the fetch width of both configurations is the same. However, the issue and wakeup (result-broadcast) ports are only four each. The total power for this queue is only 2.7W.

The total power savings seen by the Hybrid-Scheduling scheme is approximately 18% (Figure 5.2). These results demonstrate that the Hybrid-Scheduling scheme is highly effective in reducing the energy consumption in superscalar processors.

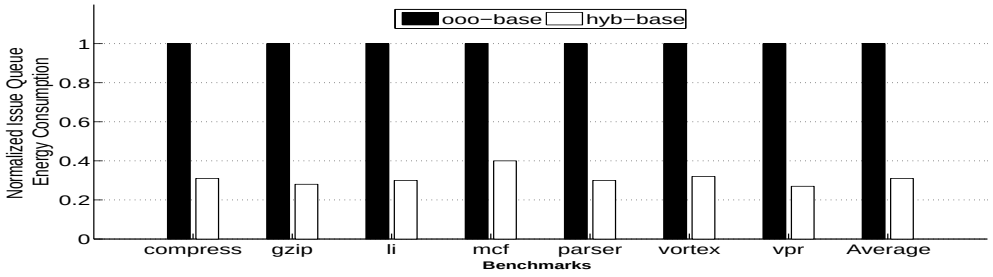


Figure 5.1: Normalized issue queue energy consumption

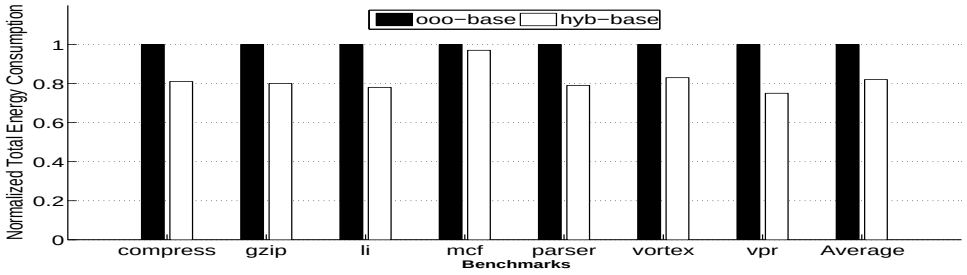


Figure 5.2: Normalized total energy consumption

5.5 Performance results

Figure 5.3 shows the performance results of the scheme. The performance degradation seen by the base Hybrid-Scheduling scheme (hyb-base in the figures) scheme is 11%. The performance degradation is significant because the heuristics used to select blocks for issue in the LR queue are guided by localized estimates made at the basic-block level. Dependences between basic-blocks, especially, between LRR blocks and HRR blocks are critical but are not captured by the localized heuristics. When a cache miss is serviced, the instructions waiting on the data can issue immediately if they are either in the out-of-order issue queue or at the heads of FIFOs. However, if there several dependences between LRR blocks and HRR blocks with cache misses, it is

unlikely that all the dependent instructions will be available at the heads of the FIFOs.

5.6 Improving the performance of the reorder-sensitive issue queue

Based on the observation that dependences between LRR blocks and HRR blocks are important, an improvement to the reorder-sensitive issue logic is provided wherein for each HRR block with a large number of cache misses, a few subsequent consecutive blocks are directed to the out-of-order issue queue irrespective of whether they are LRR blocks or HRR blocks. This helps capture the immediate dependences between LRR blocks and HRR blocks with a large number of cache misses. Note all HRR blocks need not have high miss rates; some blocks are classified as HRR blocks due to false dependences.

The number of blocks redirected to the HR queue is controlled by a simple saturating counter whose maximum value can be determined and set for each benchmark. Table 5.9 shows the average distance between dependent

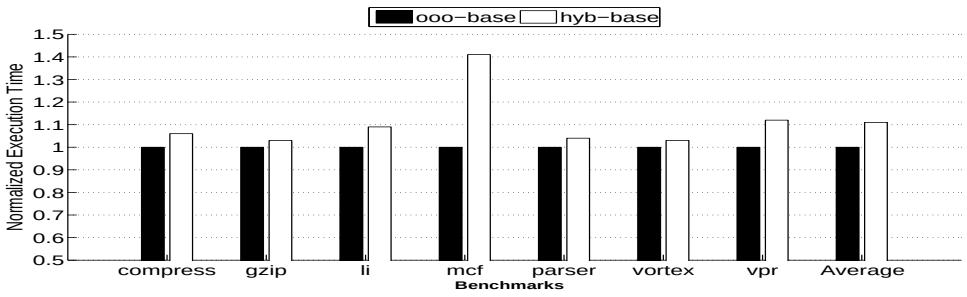


Figure 5.3: Normalized execution time

Table 5.9: Dependence distance in terms of number of basic blocks. The first column (= 0) accounts for all dependences that occur within the same block.

# of blocks	= 0	≤ 1	≤ 2	≤ 3	≤ 4	≤ 7	≤ 10
% dependences	83.6	92.6	96.0	97.5	98.4	98.7	99.9

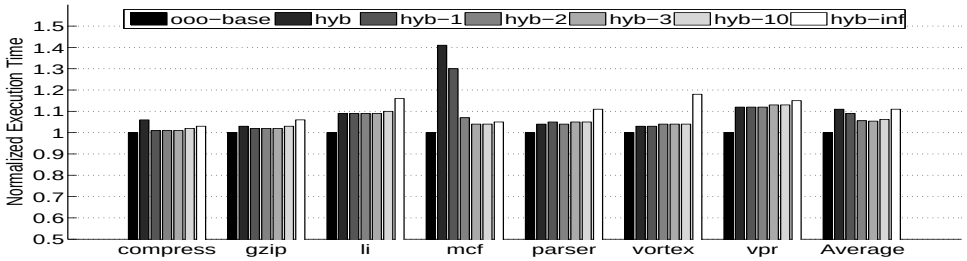


Figure 5.4: Normalized execution time for different saturating counters

instructions in number of basic blocks. The table shows that a significant portion (92%) of all the data dependences between instructions extend to just one basic-block and further, that nearly all data dependences (98%) between instructions extend to a maximum of 4 basic blocks. Hence, the maximum value of the saturating counter need not exceed 4. This value will ensure that almost all dependences between HRR and LRR blocks are captured in the HR queue.

Figure 5.4, 5.5, and 5.6 show the performance, issue queue energy savings and total energy savings of the Hybrid-Scheduling scheme for different values of the saturating counter. Figure 5.4 shows that allowing one additional block improves performance in many benchmarks. Increasing the counter value to 2 captures almost all dependences and improves performance further. The benchmark *mcf* particularly shows a good improvement for higher counter values. In benchmarks where the cache miss rates and consequently the number

of misses per block are high, it is important to have a higher counter value since many of the dependences will be critical dependences. For example, Table 5.10, shows that in *mcf*, as many as 11% of all instructions depend on loads that miss in the cache, making a significant fraction of instruction dependences critical. A bigger counter value captures more dependences and hence performs better in *mcf*. However, increasing the counter value beyond 3 begins to provide diminishing returns. In fact, increasing the counter value to a very large value begins to negatively impact performance and issue queue energy since now too many blocks are diverted to the small HR queue leading to increased dispatch stalls in the pipeline. Figure 5.4 shows that the overall performance degradation reduces from 11% to 5.5% for 2 blocks and to 5% for 3 additional blocks directed to the HR queue. The percentage of instructions issued from the LR queue reduces slightly from 30% to 26% on an average (for 2 blocks) and to 25% for 3 blocks. Note that even for the best saturating counter value, there is performance loss in the Hybrid-Scheduling scheme because the LR queue has a limited number of FIFOs, it does not provide the ideal amount of inter-block overlap. Further, not all of the HRR blocks are low ILP blocks. Some blocks that do not suffer from cache misses are also directed to the HRR queue. These blocks are unnecessarily penalized by the low issue HR queue.

The corresponding issue queue and total energy reduction are shown in Figure 5.5 and Figure 5.6 respectively. These results show that the Hybrid-Scheduling scheme can effectively reduce the power and complexity of the issue logic hardware without significantly sacrificing performance.

Table 5.10: Cache hit rates and load dependence statistics

Statistic	compress	gzip	li	mcf	parser	vortex	vpr
DL1 miss rate	8.2%	3.4%	3.9%	20.7%	2.2%	1.3%	1.2%
DL2 miss rate	20.6%	0.3%	5.7%	24.7%	18.2%	4.3%	0.6%
L1 misses/blk	0.05	0.11	0.09	0.46	0.10	0.05	0.00
L2 misses/blk	0.01	0.00	0.00	0.21	0.05	0.02	0.00
% instrs dep. on LD misses	3.3%	2.12%	3.0%	10.7%	1.91%	1.31%	1.23%

5.7 Design space exploration of the reorder-sensitive issue queue

The baseline reorder-sensitive issue queue evaluated in previous sections consisted of an LR queue with three 4-wide FIFO buffers. This section presents an evaluation of several different configurations of the reorder-sensitive issue queue. Particularly, the number, size and widths of the FIFOs are varied. The experimental results for this design space exploration are presented in Figure 5.7 and Figure 5.8.

Figure 5.7 shows the percentage performance degradation, issue energy saved and total energy saved for different number and widths of the FIFO

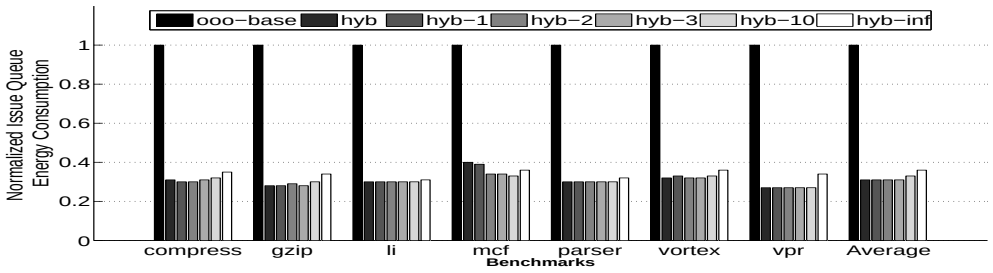


Figure 5.5: Normalized issue queue energy consumption for different saturating counters

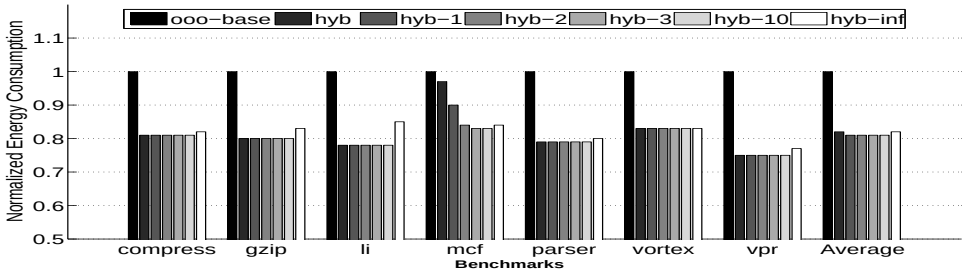


Figure 5.6: Normalized total energy consumption for different saturating counter values.

buffers. The widths (IW) examined are 2, 3 and 4. The numbers of FIFOs is varied from 2 to 8. For this figure, the number of rows per FIFO was set to 8. The first row of graphs shows the performance degradation of the reorder-sensitive configuration with respect to the baseline out-of-order issue configuration. The figures show that for a given width of the FIFO, the performance of the Hybrid-Scheduling scheme increases dramatically for increasing number of FIFOs. The primary benefit of increased number of FIFOs is that more inter-block overlap is provided. However, the disadvantage of the increasing number of FIFOs is that the issue queue energy and the total energy saved reduces, although the reduction in energy savings is not very severe. This is because FIFO structures are inherently low-power and hence adding FIFOs still provides a reasonable power-performance trade-off. Another disadvantage of increased FIFOs is that the complexity of the issue logic increases. Particularly, the complexity of "waking up" instructions and selecting instructions for issue increases since more instructions at the heads of the FIFOs need to be examined.

For increasing issue widths of the FIFOs, the performance is significantly improved. For example, a 4-wide 2 FIFO configuration results in 7%

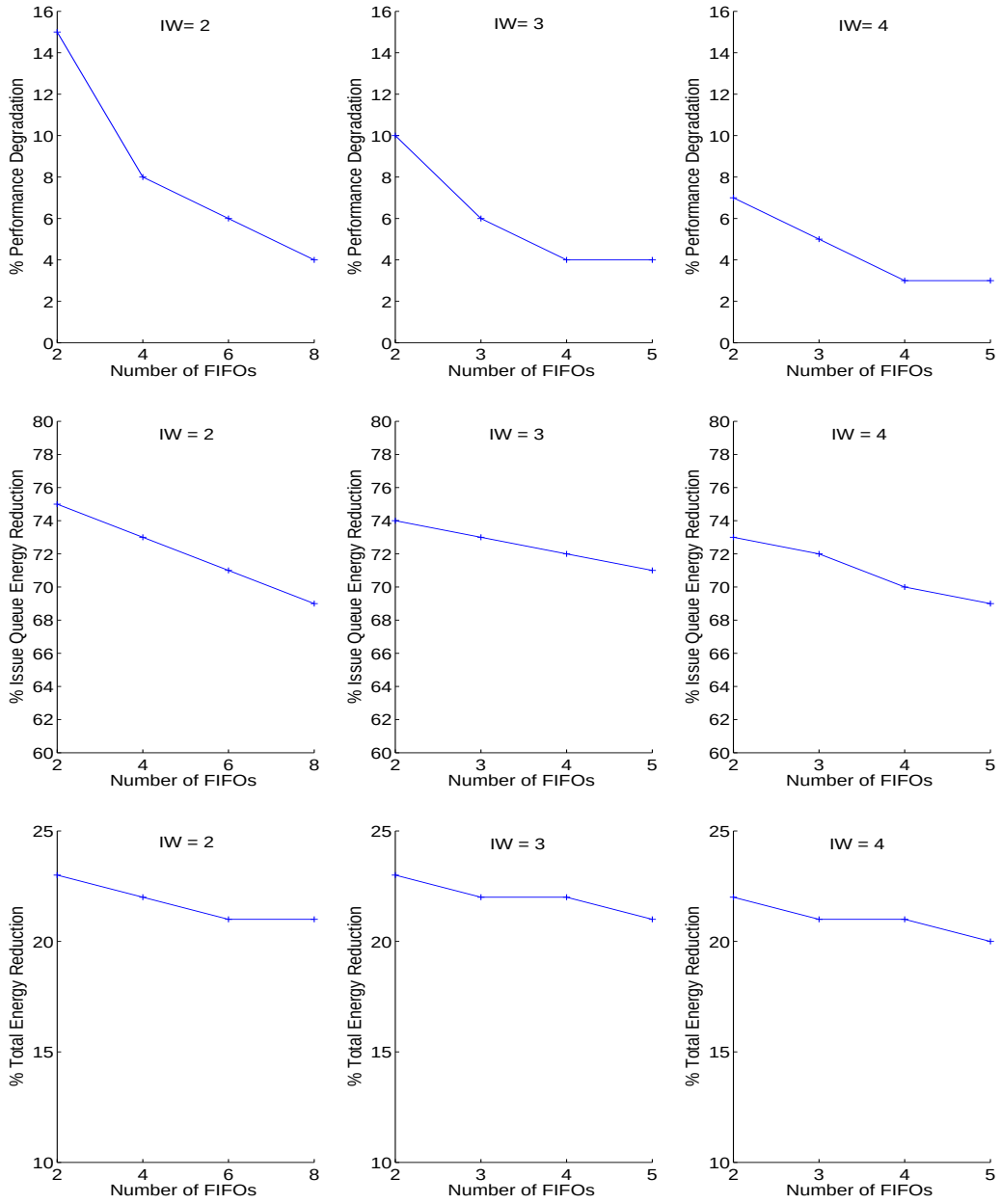


Figure 5.7: Percentage performance degradation, issue queue energy reduction and total processor energy reduction for varying FIFO configurations, averaged over all benchmarks. The issue widths examined are 2,3 and 4. The number of FIFOs vary from 2 to 8. The number of rows per FIFO is set to 8.

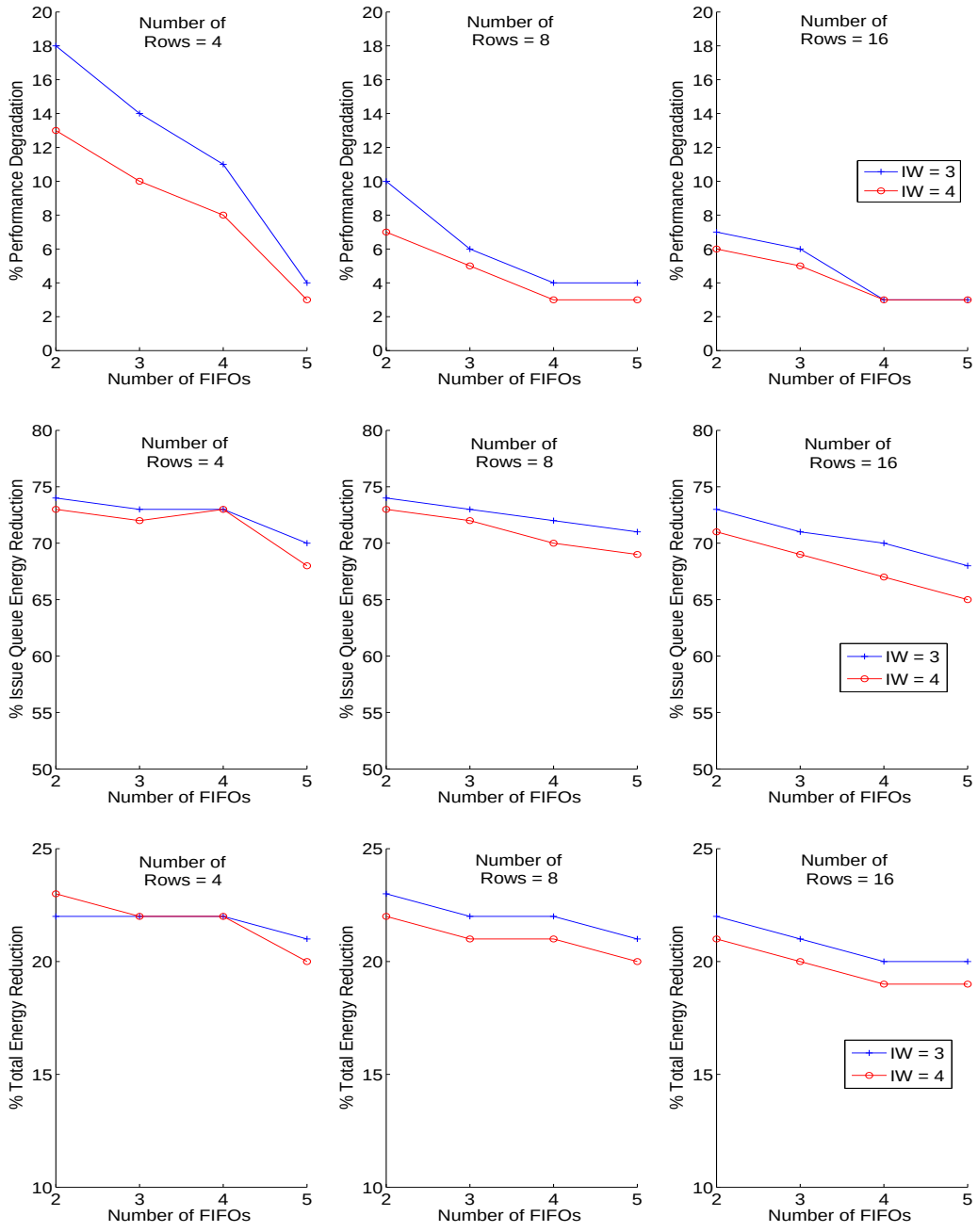


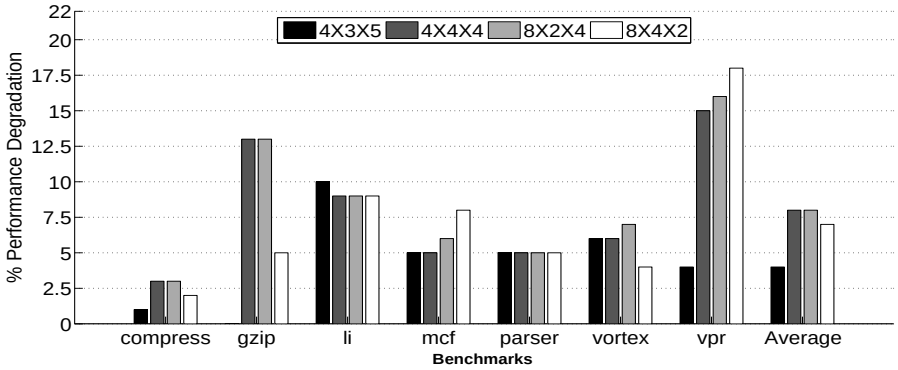
Figure 5.8: Percentage performance degradation, issue queue energy reduction and total processor energy reduction for varying FIFO configurations. The number of FIFOs vary from 2 to 8. The Issue widths of FIFOs are 3 and 4. The number of rows per FIFO is 4, 8 or 16.

performance degradation while a 2-wide 2 FIFO configuration causes the performance to drop by as much as 15%.

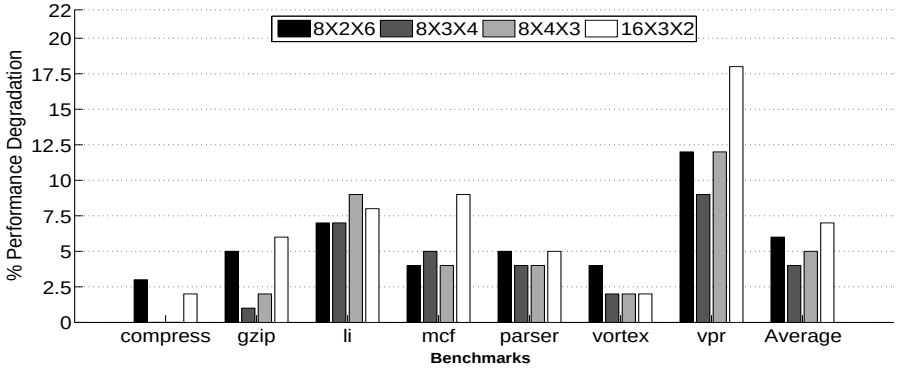
Figure 5.8 plots the performance degradation, issue queue energy savings and total energy savings for varying number of rows per FIFO. The number of rows examined are 4, 8 and 16. The figure shows the results for the issue widths 3 and 4, which correspond to the better performing configurations. The figure shows that increasing the number of rows has a significant impact on the performance degradation. For example, increasing the number of rows per FIFO, decreases the performance degradation of the Hybrid-Scheduling scheme from 18% for a 3-wide 2 FIFO configuration with 4 rows per FIFO to 7% for 16 entries per FIFO. Note that in all cases, the issue queue and the total energy consumption of the processor do not increase significantly, indicating that FIFOs are indeed low overhead structures.

An interesting observation from Figure 5.7 and also in Figure 5.8 is that the increasing issue width has lower impact when the number of FIFOs is larger (Figure 5.7) or when the number of rows in each FIFO is higher (Figure 5.8). This trend suggests that the total number of entries rather than any individual parameter has the most impact on the overall performance of the scheme.

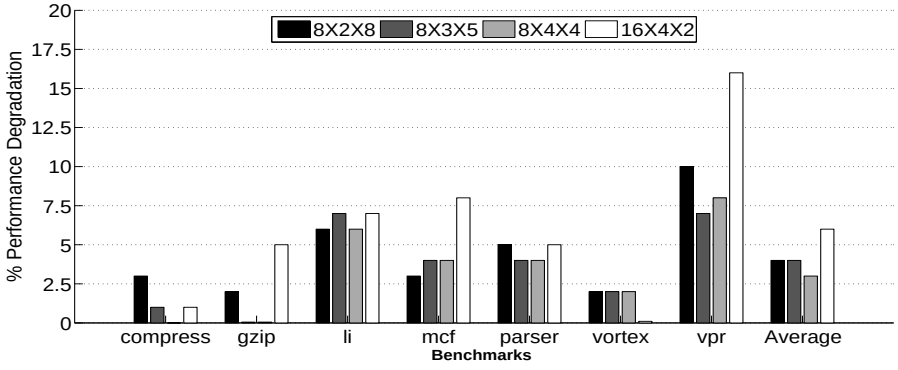
Figure 5.9 shows several different configurations of the Hybrid-Scheduling scheme but with each configuration having the same number of total FIFO entries. The key observation from the figure is that in general, configurations with larger number of FIFOs performs better than a configuration with fewer FIFOs. For example, the figure shows that a 3-wide 5 FIFO with 4 rows per FIFO configuration (Figure 5.9(a)) performs better than a 4-wide 2 FIFO with 16 rows per FIFO configuration (Figure 5.9(c), even though the total number of entries in the former is half that of the latter. The configurations 16X3X2



(a) 64 Entries



(b) 96 Entries



(c) 128 Entries

Figure 5.9: Percentage performance degradation of the Hybrid-Scheduling scheme for several different FIFO configurations. Each configuration has the same number of total FIFO entries. Configurations are represented as a combination LxMxN, where L is the number of rows per FIFO, M is the width of the FIFO and N is the number of FIFOs.

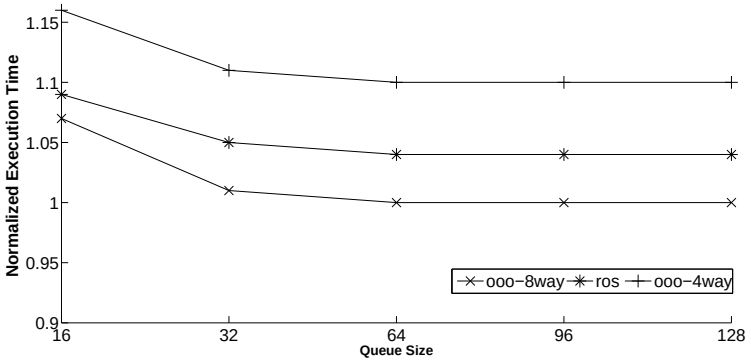


Figure 5.10: Performance results of different reorder-sensitive and conventional issue queue configurations for varying queue sizes. For the ROS schemes, queue size corresponds to the HR queue size. Results shown are averaged over all benchmarks.

(which indicates a 3-wide, 2 FIFO with 16 rows per FIFO scenario), and 16X4X2, which have the lowest number of FIFOs, are the worst performing schemes. This is intuitive since larger number of FIFOs lend more inter-block overlap. Further, configurations where the number of rows per FIFO or the width of the FIFO is too low also cause significant performance degradation (4X4X4, 8X2X4, 8X2X6).

5.8 Comparing out-of-order issue queue and reorder-sensitive issue queue configurations

Figures 5.10 and 5.11 compare the Hybrid-Scheduling issue logic with 4 and 8-wide conventional queues for varying issue queue sizes. The results demonstrate that reorder-sensitive issue queue configurations can consistently achieve high performance (within 5% of an 8-wide out-of-order issue queue), while consuming significantly lower energy (close to a 4-wide out-of-order issue queue).

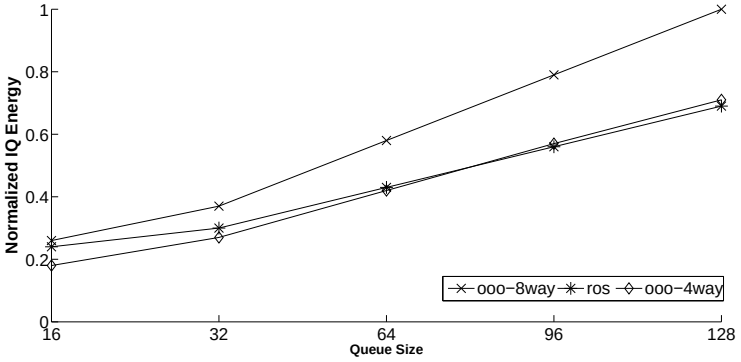


Figure 5.11: Issue queue energy results of different ROS and conventional issue queue configurations for varying queue sizes. For the ROS schemes, queue size corresponds to the HR queue size. Results shown are averaged over all benchmarks.

5.9 Comparing a dynamic dependence-based FIFO scheme with the Hybrid-Scheduling approach

There have been several run-time schemes proposed for alleviating the complexity of the issue logic. These techniques attempt to reduce the complexity of critical issue logic tasks such as instruction wakeup and select. A majority of the previously proposed techniques reduce complexity of the issue queue by limiting the number of candidate instructions to be considered for issue [1, 15, 20, 41, 42, 46, 48, 37]. These techniques typically consist of a *pre-scheduling* phase wherein the data-dependences of instructions are analyzed. After the pre-scheduling phase, instructions are typically steered to various low-complexity FIFOs based on their dependences with older instructions in the queues [1, 46, 48].

While these techniques indeed alleviate the complexity of the issue logic,

they often require extra hardware and/or the addition of a few pipeline stages. In the Hybrid-Scheduling work, since the necessary analysis is performed statically, the burden is shifted to the compiler, thereby eliminating the need for any auxiliary hardware resources or pipeline stages.

This section compares the Hybrid-Scheduling scheme with a dynamic scheme to alleviate complexity and power. Abella *et al.* [1] present a comprehensive analysis of several complexity-effective issue queues and show that the FIFO-based scheme suggested by Palacharla *et al.* [46] is one of the best performing schemes for integer programs. A brief description of the scheme is provided first.

The scheme proposed by Palacharla *et al.* uses several single-issue FIFOs. Only the instructions at the head of each FIFO are considered for issue. The scheme uses hardware logic to analyze dependences between instructions to steer instructions to different FIFOs. Instructions are dispatched with the following heuristics:

Let `I` be the instruction under consideration. Depending upon the availability of `I`'s operands, the following cases are possible:

- All the operands of `I` have already been computed and are residing in the register file. In this case, `I` is steered to a new (empty) FIFO acquired from a pool of free FIFOs.

- `I` requires a single outstanding operand to be produced by instruction Isource residing in FIFO `Fa`. In this case, if there is no instruction behind `Isource` in `Fa` then `I` is steered to `Fa`, else `I` is steered to a new FIFO.

- `I` requires two outstanding operands to be produced by instructions `Ileft` and `Iright` residing in FIFOs `Fa` and `Fb` respectively. In this

case, apply the heuristic in the previous bullet to the left operand. If the resulting FIFO is not suitable (it is either full or there is an instruction behind the source instruction), then apply the same heuristic to the right operand.

If all the FIFOs are full or if no empty FIFO is available then the decoder/steering logic stalls. A FIFO is returned to the free pool when the last instruction in the FIFO is issued. Initially, all the FIFOs are in the free pool.

Dependence information between instructions is maintained in a table called the SRC_FIFO table. This table can be indexed using logical register designators or physical register designators. If the table is indexed with logic registers, then instruction steering could potentially be performed in parallel with register renaming. However, since completing instructions have tags which correspond to physical registers, updating this table each cycle is simpler if it is indexed with the physical register designators. This avoids potential translations through multiple tables.

The dynamic dependence-based scheme thus require several additional hardware structures such as the SRC_FIFO table, additional dependence logic similar to the renaming logic for identifying the appropriate FIFO for each instruction and other auxiliary structures to maintain occupancy status for the FIFOs. Additionally, the steering logic, an inherently serial operation [41], potentially requires one or more additional pipeline stages, especially in smaller process technologies.

A detailed analysis by Abella *et al.* [1] shows that for integer programs, having higher number of queues is better than more entries per queue. The dynamic scheme used for comparison consists of 16 FIFOs with eight entries

each. Thus, the total number of entries in the dynamic scheme, the Hybrid-Scheduling scheme and the out-of-order issue baseline are all 128 entries. Each FIFO in the dynamic scheme consists of 8 write ports and 1 read port. Further, power consumption of several axillary structures have been taken into account. The structures include the SRC_FIFO table, FIFO_tail table (to maintain information about tail instructions), the counters to maintain occupancy status of the FIFOs, and the dependence logic required to compute the correct FIFO for each instruction.

Figures 5.12 and 5.13 show the performance and issue queue energy consumption of the out-of-order issue baseline, the Hybrid-Scheduling scheme and the dynamic dependence-based FIFO scheme. It can be seen that the Hybrid-Scheduling scheme achieves higher reduction in issue queue energy when compared to the dynamic scheme. Further, although in the ideal case, *i.e.,* when the dynamic scheme requires no additional pipeline stages, it performs marginally better than the Hybrid-Scheduling scheme. However, in realistic scenarios, the dynamic scheme requires additional pipeline stages for steering instructions and thus exhibit lower performance when compared to the Hybrid-Scheduling approach.

5.10 Evaluation of the block selection heuristics

To compute the estimated schedule degradation due to cache misses (SD_{cm}), the compiler requires good estimates (*effective costs*) for the miss hiding capability of the out-of-order issue logic. The effective cost seen by an instruction executing in its statically-scheduled order is usually smaller that the actual

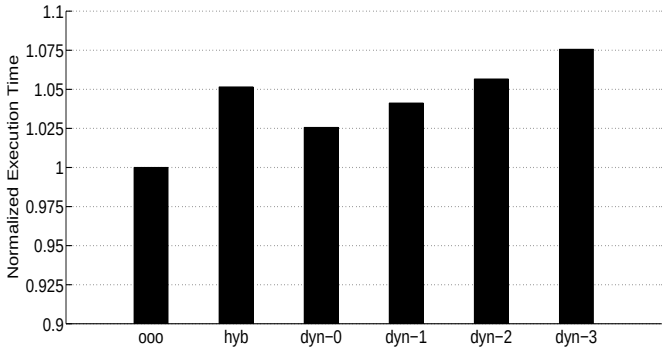


Figure 5.12: Performance results of the baseline out-of-order issue processor, Hybrid-Scheduling scheme and the dynamic dependence-based FIFO scheme. For the dynamic schemes, n indicates the additional pipeline stages required by the steering logic.

latency of a cache miss, since the hardware can not hide the complete cost of the miss. The effective cost is a complex function of the cache miss latencies, size of the issue queue, issue width of the processor and available ILP in the program.

The effective L1 and L2 miss costs are empirically identified as shown in Figure 5.14. At smaller estimated costs, a larger percentage of blocks qualify as LRR blocks and hence a larger number of instructions are issued from the LR queue. However, the performance loss is also prohibitively high, indicating that the miss hiding capability of the out-of-order issue logic is being underestimated. As the assumed costs are progressively increased, the number of instructions issued from the LR queue and the performance degradation decrease. The figure shows that assuming an L1 miss cost in the range of 20-30 cycles is a reasonable estimate for the L1 cost, since increasing the estimate beyond this does not dramatically decrease the performance loss. Assuming that the average IPC of integer programs on an 8-way machine is 3, a 30-cycle

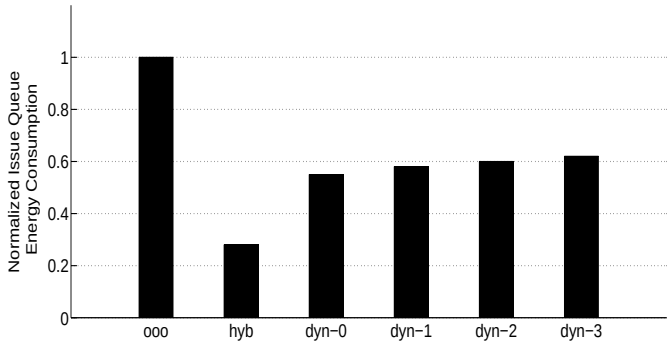


Figure 5.13: Performance results of the baseline out-of-order issue processor, Hybrid-Scheduling scheme and the dynamic dependence-based FIFO scheme. For the dynamic schemes, n indicates the additional pipeline stages required by the steering logic.

cost can be roughly interpreted as the out-of-order issue queue being able to hide about half the L1 miss latency (10 cycles) with 3 instructions issued each cycle. Note that when executing on a fully in-order machine, where it is not possible to hide *any* stalls due to cache misses at run-time, the cost would be considerably higher. However, since the LR queue allows some degree of overlap of instructions, the effective cycle-time cost is lower. It can also be observed that since the number of L2 misses are considerably lower than the number of L1 misses, the block selection heuristic is largely insensitive to the L2 miss cost.

The compiler selects LRR blocks based on statically computed estimates of the schedule quality degradation due to false dependences (SD_{fd}) and memory misses (SD_{cm}). The compiler classifies a block as an LRR block if its estimated schedule degradation due to false dependences and memory misses (SD_{fd} and SD_{cm}) are less than their respective threshold values (FD_{th} and CM_{th}). Figure 5.15 presents an evaluation of the sensitivity of the block

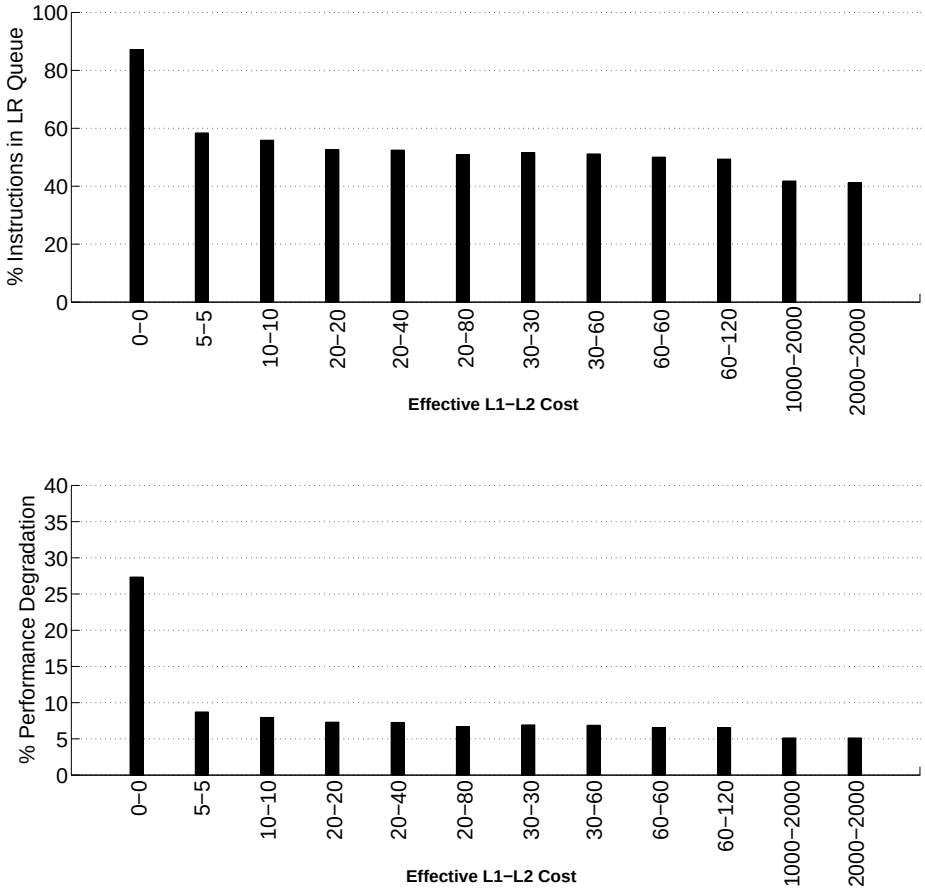


Figure 5.14: Percentage of instructions in the LR queue and performance degradation for different L1-L2 miss costs (averaged over all benchmarks). Costs expressed as `X1-Y1`, where `X1` is the effective L1 miss cost and `Y1` is the L2 miss cost in cycles. CM_{th} is fixed at 5%.

selection policy to varying threshold values. The figure shows the percentage of dynamic instructions that are issued from the LR queue and the corresponding performance degradation for different FD_{th} and CM_{th} values for the benchmark *compress* (the sensitivity analysis is shown for only one benchmark but all other benchmarks show largely similar trends.)

Larger threshold values indicate willingness to tolerate a larger performance loss. With large threshold values the compiler selects more blocks for

issue in the LR queue. Since the LR queue is extremely simple and consumes lower power when compared to the HR queue, the processor can operate in a power-savings mode by setting large threshold values. Alternatively, setting lower thresholds allows the processor to operate in a more performance-sensitive mode. The actual performance degradation observed is lower than the estimated performance degradation (indicated by the threshold values). The reason for this is that the estimate is computed assuming the blocks will be issued in a strictly in-order fashion. However, since instructions from the heads of the FIFOs can issue in any order, the LR queue provides some degree of overlap between blocks. Thus, a considerable portion of the cache miss stall cycles are hidden in the reorder-sensitive issue queue.

An interesting observation from Figure 5.15 is that even when both thresholds are set to 0, there are a significant number of instructions (32%) that are selected for issue in the LR queue. For all benchmarks, on average, approximately 30% of instructions qualify as LRR instructions. This indicates that a large number of blocks do not contain any false dependences nor experience any memory misses. For these blocks, expending energy to reorder instructions in hardware is wasteful since the compiler can generate efficient schedules. The reorder-sensitive scheme removes this inherent redundancy by providing the ability to harness compiler-generated schedules for these blocks.

Another important observation is that when the number of blocks issued to the LR queue is very high, the performance degradation can be significant. Table 5.11 shows the performance degradation experienced when all instructions are directed to FIFOs. For this experiment, four 4-wide FIFO buffers with eight entries in each, are used. These results underscore the importance of dynamic scheduling and show that an "all FIFO" configuration is not a viable alternative to fully associative queues. The Hybrid-Scheduling

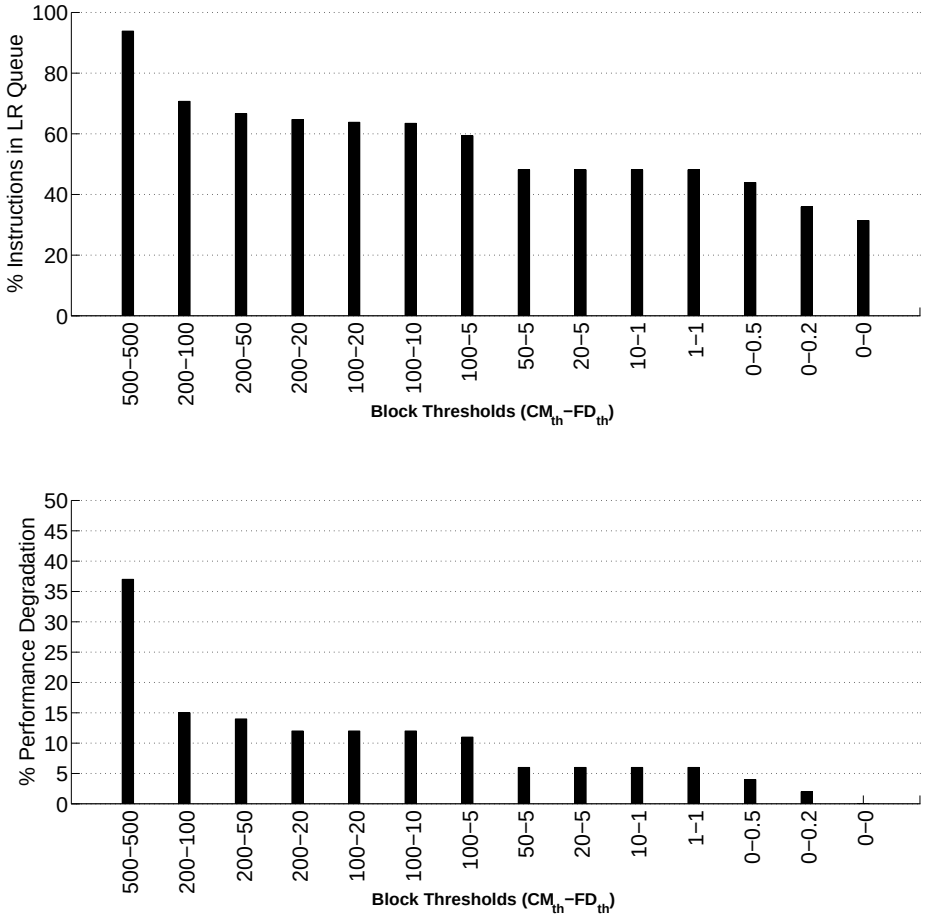


Figure 5.15: Percentage of instructions in the LR queue and performance degradation for different block selection thresholds (for benchmark *compress*). Thresholds represented as FD_{th}-CM_{th}.

scheme combines the benefits of both FIFOs and fully associative queues in the reorder-sensitive issue queue and by judiciously selecting appropriate blocks to different scheduling queues limits performance loss while maximizing energy reduction.

Table 5.11: Percentage performance degradation when all instructions in the program are dispatched to LR queue. The LR queue used consists of four 4-wide FIFOs with 8 entries each.

compress	gzip	li	mcf	parser	vortex	vpr	Average
29.5%	34.2%	31.3%	97.9%	30.5%	27.35%	36.8%	43%

5.11 Discussion

As with any profile-based approach, the performance of the Hybrid-Scheduling scheme depends on how closely a program's actual run-time tendencies match those seen during specialization. The experiments presented in this chapter demonstrate that the degradation is not significant even when the input data is different from the profile input. However, it is desirable to have a protection mechanism against the rare cases when the input data completely deviates from the profile data. Unforeseen cache misses are particularly hazardous since the LR queue is quite limited in its capacity to hide the miss penalty.

One attractive solution is to implement the reorder-sensitive scheme within a hardware or software dynamic optimization framework [50, 74, 75, 76]. The dynamic optimizer could be used to monitor the behavior exhibited by the running application and dynamically tune the heuristics to continually divert blocks with cache miss rates to the fully-associative queue which is better equipped to handle misses.

Another approach is to add some global loop-level or procedure-level control over the localized issue policy used in the Hybrid-Scheduling scheme. Prior work has shown that it is possible to predict the IPC of a program statically with reasonable accuracy using simple compile-time analysis [61]. During execution of the program, the hardware could dynamically monitor the IPC of important program hotspots. If the observed IPC is significantly

below the estimated IPC, the reorder-sensitive issue mode can be disabled and instructions can default to the conventional out-of-order issue queue.

5.12 Summary

This chapter presented a detailed evaluation of the Hybrid-Scheduling scheme. The experimental results demonstrate that the proposed scheme can reduce energy consumption in the issue queue dramatically by 70% on average. The issue queue expends a significant portion of the total energy in wide-issue processors. Consequently, the Hybrid-Scheduling scheme on average saves 18% of the total processor power without significant performance loss. Additionally, the proposed issue hardware is significantly less complex when compared to a conventional monolithic out-of-order issue queue. The low complexity and improved power margins can be leveraged to operate the processor at a considerably higher frequencies.

Chapter 6

Hybrid-Scheduling for Media and Scientific Programs

Media and scientific applications are important applications in the desktop and server computing markets respectively. The execution time of these applications is dominated by regular loops. For well-understood structures such as loops, the compiler can exploit and enhance parallelism with the help of several aggressive optimizations such as loop unrolling, software pipelining and trace scheduling. These programs thus exhibit tremendous potential for applying the Hybrid-Scheduling technique.

This chapter presents Hybrid-Scheduling in the context of such regular applications. Considering S-Regions at larger, loop-level granularities offers opportunities to simplify the hardware requirements dramatically. The profile-driven compiler analysis for identifying and selecting loop-level S-Regions is described first, followed by a description of the Hybrid-Scheduling micro-architecture for these types of applications. A detailed evaluation of the proposed scheme is also presented.

6.1 Overview

Program execution time in media and scientific applications is typically dominated by regular loops. Loops are highly amenable to several compile-time ILP

(instruction-level parallelism) optimizations. The loops in these applications also exhibit regular, predictable memory access patterns [49]. Regions with regular access patterns are amenable to techniques such as prefetching and are thus suitable candidates for static issue. Further, these loops are typically long-running, thus creating long contiguous S-Regions.

In the context of Hybrid-Scheduling, the primary advantage of loop-level S-Regions is that the compiler can create large basic-blocks using optimizations such as loop unrolling and can enhance the available ILP using aggressive instruction scheduling optimizations such as software pipelining and trace scheduling. With large basic blocks and ample available ILP, the compiler can find sufficient number of instructions to fill a significant number of the issue slots in the pipeline. Thus, loop-level S-Regions do not require "inter-block overlap"(Chapter 3) and can issue in their statically-scheduled order with minimal hardware support for long durations [63].

In this derivative of the Hybrid-Scheduling scheme, rather than supporting two differing types of issue queues, there are two distinct *modes* of execution. In this **Dual-Mode (DM) Hybrid-Scheduling** microarchitecture, the processor runs in the superscalar mode with dynamic scheduling until a special instruction that indicates the beginning of an S-Region is detected. At this point, the out-of-order issue engine is shut down, and the processor switches to a VLIW-like *static mode* in which instruction packets scheduled by the compiler are issued sequentially in consecutive cycles with minimal hardware support. Energy is conserved primarily by reducing the work done in the out-of-order issue logic.

This chapter describes the compiler and micro-architectural details of the Dual-Mode Hybrid-Scheduling scheme. The heuristics for identifying loop-level S-Regions are described first.

```
Profile Collection Phase

for each loop region {
    record L1 miss rate, L2 miss rate;
    record Iteration counts of innermost loop nests I1 and I2;
    record Duration D;
}

Profile Analysis Phase

for each loop region {

    Compute PD_sw; // performance degradation due to switching overhead
    Compute PD_ss; // performance degradation due to imperfect static scheduling
    Compute PD_cm; // performance degradation due to cache misses

    if (PD_sw + PD_sc + PD_cm < PD_th){
        Select loop as S−region;
        Annotate S−Regions with marker instructions;
    }
}
```

Figure 6.1: Algorithm for selecting loop-level S-Regions

6.2 S-Regions in media and scientific programs

Chapter 3 discussed the compiler heuristics for identifying S-Regions at the basic-block level. The heuristics required for selecting S-Regions at loop-level granularity are similar. Figure 6.1 summarizes the algorithm for selecting S-Regions are that suitable for the Dual-Mode Hybrid-Scheduling architecture. The primary goal is to choose loops that provide energy reduction with minimum performance degradation when issued in the statically-scheduled order.

Applications are profiled for several characteristics to estimate the performance degradation caused due to the following: (a) switching overhead (PD_{sw}),

(b) imperfect schedules (PD_{ss}) and (c) cache misses (PD_{cm}). Specifically, for each S-Region i, the following statistics are collected during profiling:

- Duration of each S-Region (D_i)
- Average number of L1 misses for each S-Region ($L1_i$)
- Average number of L2 misses for each S-Region ($L2_i$)
- Iteration count of loop (for each nest-level: $I1_i$, $I2_i$, $I3_i$,..)

Step 1: Estimating cost of switching overhead PD_{sw}

S-Regions should have long durations since there is an overhead for switching between the two independent execution modes. The overhead incurred is mainly the time required to drain (and prime) the superscalar pipeline when entering (and exiting) the static mode. With long running S-Regions, this cost becomes negligible. For example, for the processor configuration described in Section 6.6, the penalty for switching between the static mode and dynamic modes of execution is observed to be approximately 20-30 cycles. By setting the smallest duration of S-Regions to 500 cycles, the performance drop due to the switching overhead (PD_{sw}) can be limited to to approximately 5%. PD_{sw} can be computed as:

$$PD_{sw} = \frac{Switching_{penalty}}{D_i} \tag{6.1}$$

Step 2: Estimating performance degradation due to imperfect static schedules PD_{sc}

Unresolved memory addresses or memory aliases and insufficient registers often restrict the static scheduler from deriving optimal schedules. However, the media and scientific programs examined in this study exhibited virtually no memory aliases and false register dependences. The number of memory aliases were limited since the primary data structures used in these programs were arrays with regular data access patterns. For such arrays, the compiler can typically disambiguate the memory addresses statically. The number of false dependences within the loops were also few. There were two main reasons for this. First, the Alpha compiler used in the experiments employed is a sophisticated register allocator to efficiently use the 32 architectural registers in the Alpha ISA. Further, there are few live-in and live-out values in the loops, implying that almost all 32 registers are typically available for use within the S-Regions.

Although false data and memory dependences are minimal, other impediments such as integral schedule length restrictions still limit the compile-time instruction scheduler. Indeed, the primary cause for performance degradation was the integral schedule length limitation faced by compile-time schedulers (*i.e.,* the schedule length of a basic block is restricted to be an integer value). For example, consider a simple singly-nested media loop that contains 42 instructions. Since the loop is highly parallel, the ideal schedule if the target architecture can issue four instructions each cycle is $42/4 = 10.5$ cycles (assuming there are sufficient functional units). However, the final schedule achieved by the static schedule is $\lceil 10.5 \rceil = 11$ cycles, since the compiler cannot assume non-integral schedule lengths. Two issue slots of every 40 are left

unused in this schedule, leading to 5% performance degradation. On the other hand, the dynamic issue logic in the superscalar processor has no wasted slots since instructions from the next iteration can be overlapped with the current iteration at run-time.

Hence, to estimate the schedule efficiency of the compiler-generated schedule, the penalty on the schedule due to this limitation is estimated. To estimate the cost, iteration counts of the S-Regions are profiled. In most loops, it is sufficient to profile the innermost loop count alone since the performance is determined primarily by that loop. However, in media programs, several important loops have very short innermost loops (such as *dct or quantization*). Hence, the iteration counts of the innermost two loops ($I1_i$, $I2_i$) are profiled and the total time taken for the compiler-generated loop (SC_i) to execute is computed:

$$SC_i = (\ (\ C1_i * I1_i \ ) + C2_i \ ) * I2_i \tag{6.2}$$

where, $C1_i$ and $C2_i$ are the schedule lengths of the two loop nests.

To estimate the cost of running the loop in the dynamic mode with out-of-order issue, the total number of instructions N_i executed by the loop is first computed. If the processor has a sufficiently large issue queue and reorder buffer, for ILP-rich regions such as media and scientific loops, the dynamic issue logic can find sufficient parallelism to keep the pipeline and functional units busy in almost every cycle. Hence, the total time taken for the S-Region to execute in the dynamic mode (SD_i) is estimated to be simply the total number of instructions divided by the issue width of the processor.

$$N_i = (\ (\ N1_i * I_{i1}\ ) + N2_i\ ) * I_{i2} \tag{6.3}$$

where, $N1_i$ and $N2_i$ are the number of instructions in each nest level of the loop.

$$SD_i = \frac{N_i}{IW} \tag{6.4}$$

The estimated performance difference between compiler-generated schedules and those achieved by the dynamic issue logic (PD_{ss}) can thus be computed as:

$$PD_{ss} = \frac{(SC_i - SD_i)}{SD_i} \tag{6.5}$$

Step 3: Estimating cost of cache misses PD_{cm}

Without dynamic scheduling it is difficult to hide the latency of an L1 cache miss and even harder to hide the larger latency miss in the L2 cache. Fritts *et al.* [24] and Banerjee *et al.* [7] note that long latency cache misses are some of the biggest impediments to achieving high performance in statically scheduled processors. Thus, the cost of these misses is estimated to ensure that the performance impact in selected S-Regions is limited. To identify the cost, the average number of L1 misses ($L1_i$) and L2 misses ($L2_i$) are profiled. The execution time (E_i) of the S-Region increases by:

$$E_i = SC_i + L1_i * L2\ Latency + L2_i * Memory\ Latency \tag{6.6}$$

The heuristics conservatively estimate that the dynamic scheduler can hide all the L1 and L2 misses with a sufficiently large window. Hence, the performance degradation is given by:

$$PD_{cm} = \frac{(E_i - SD_i)}{SD_i} \tag{6.7}$$

Step 4: Final selection of S-Regions

Final S-Regions are selected such that the sum of the performance degradation due to switching overhead, imperfect schedules and memory misses, is less than a given performance threshold of PD_{th} as shown in Figure 6.1. The compiler identifies regions in programs that satisfy the given constrains and schedules them with different ILP optimizations to create efficient schedules. The compiler schedules instructions into 'packets' of independent instructions. All the instructions in a packet are guaranteed to be independent and can be issued to the function units in parallel. A packet style similar to the MultiOp-S packet semantics described in the HPL-PD instruction set architecture [66] is used. MultiOp-S instructions can be issued in parallel but can also be issued sequentially from left to right. This permits the S-Region code to run correctly in both static and dynamic modes. Additionally, a program compiled for a specific Hybrid-Scheduling machine can also run correctly on general-purpose processors with different resource configurations. The compiler finally annotates S-Regions with special "start-static" and "end-static" instructions to

indicate the beginning and the end of S-Regions. S-Regions can have multiple basic-blocks and the first instruction in each basic-block starts a new instruction packet. The compiler also annotates each basic-block with a "start-block" instruction.

6.3 The Dual-Mode Hybrid-Scheduling microarchitecture

A high-level view of the Dual-Mode Hybrid-Scheduling microarchitecture is shown in Figure 6.2 [63]. In this architecture, S-Regions bypass the dynamic issue logic and execute in a low power static mode. S-Region instruction packets are issued to the functional units in their statically-scheduled with minimal hardware support.

The Dual-Mode Hybrid-Scheduling microarchitecture extends the features of a generic superscalar processor with various low energy structures to support static mode issue. These include a small buffer to hold S-Regions and a low-energy mechanism to support precise exceptions for instruction execution in the static mode. The remainder of this section describes in detail the various features of the Dual-Mode Hybrid-Scheduling architecture.

Initial Program Execution: In this scheme, the program begins execution in the normal superscalar fashion, *i.e.*, decoded instructions are dispatched to the instruction window where they wait for their operands, ready instructions are issued to the functional units and finally instructions retire in-order from the reorder buffer. Execution continues in the superscalar mode until "start-static", an instruction indicating the beginning of an S-Region, is detected.

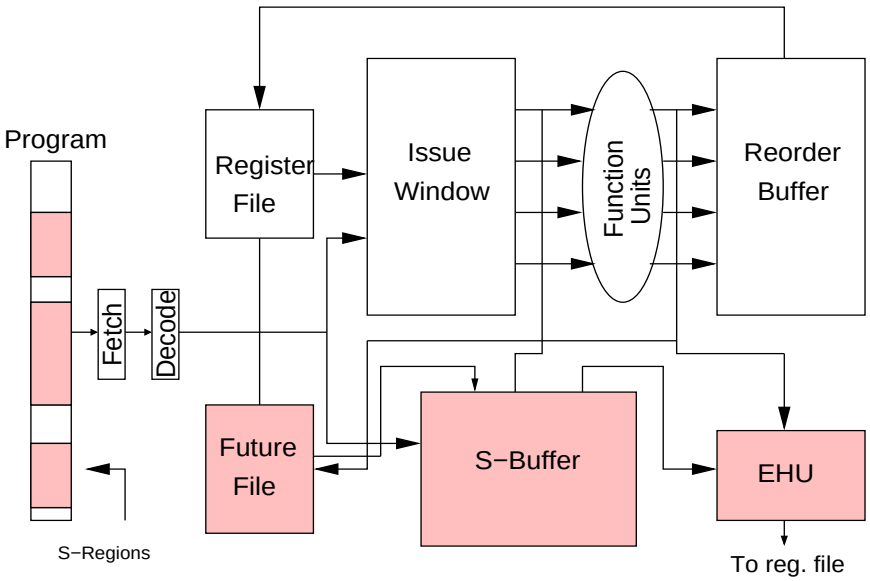


Figure 6.2: The Dual-Mode Hybrid-Scheduling microarchitecture

The S-Buffer: All instructions following the "start-static" instruction, *i.e.*, S-Region instructions, are stored in the S-Buffer. Instructions are placed in the S-Buffer until the "end-static" instruction is detected indicating that all the instructions belonging to the S-Region have been captured in the buffer. A detailed structure of the S-Buffer is shown in Figure 6.3. The S-Buffer is a circular buffer and each line holds one instruction packet. The packet size is fixed and is determined at design time. The maximum packet size is the number of function units available in the processor. Each S-Buffer line contains a PC field where the program counter value of the first instruction in the group is stored. Each line also has two special bits, namely, S-bit and B-bit, which are set if the line holds the first instruction of an *S-Region* or if it holds a branch instruction respectively. The buffer also maintains a pointer (*fill_ptr*) to the next available entry for filling. Consecutive instruction packets are placed in consecutive lines of the S-Buffer.

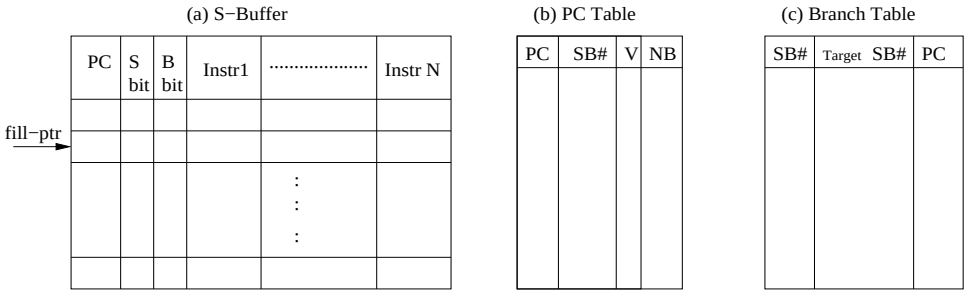


Figure 6.3: Structures used in the static mode

Switching to Static Mode: The compiler schedules S-Region instructions assuming that all the function units are available for use in the static mode and that all the register values are available in the register file. This implies that the S-Region instructions can begin execution only after the last instruction in the superscalar mode has completed. Therefore, static mode issue can begin immediately provided the superscalar pipeline has drained while the S-Region was being captured. If not, the processor stalls for a few additional cycles until the superscalar pipeline drains completely. Thus, there is a cycle-time overhead incurred in switching between the two modes. However, by choosing S-Regions that execute for long durations in the static mode, the switching cost can be amortized. After the last instruction in the superscalar mode has completed, the dynamic issue logic is turned off and instructions begin issuing from the S-Buffer. Thus, just prior to static mode execution, the latest values of registers are available in the register file. Note that since S-Region instructions are issued in their statically-scheduled order, they need not be renamed and are placed into the buffer before register renaming is performed.

Instruction Execution in Static Mode: Instructions from the S-Buffer

are issued to the functional units without any further dependence analysis. One complete S-Buffer line is issued every cycle. Memory misses are handled by hardware interlocks and cause the issue of the subsequent packet to stall. All instructions issued in the static mode completely bypass the issue logic, leading to enormous energy savings in the processor. Instructions issue in the static mode until the last instruction of the S-Region is executed (detected by the "end-static" instruction). The processor then exits the static mode, starts fetching from the instruction cache and returns to the superscalar mode of execution.

Tracking S-Regions: The scheme uses a small content addressable table, called PC-Table (Program-Counter Table) as shown in Figure 6.3, to track all S-Region blocks in the S-Buffer. Each entry in the table contains a PC field and the corresponding S-Buffer line number holding the first instruction of the region. Each time a new S-Region is filled into the S-Buffer, an entry is created in the PC table. When a "start-static" instruction is decoded, the PC table is probed to check if the corresponding S-Region is already present in the S-Buffer.

The "start-block" instruction also causes an entry to be created in the PC table. To track S-Regions with multiple basic blocks, the entries of the PC table are augmented with a field to hold the number of blocks (NB field). This is set by the "start-static" instruction. Hence, only the first PC table entry of a region has this field set.

The PC table entry is invalidated when the S-Buffer is full and a new S-Region has to overwrite an existing region. Invalidation of S-Regions is handled by the S-bit. The S-Buffer line holding the first instruction in the region has its S-Bit set. The value of the S-bit is always checked before filling

a line in the S-Buffer. If the bit is set, the entry of the previous S-Region (including all the basic blocks in the region) is invalidated in the PC Table before proceeding any further. Note that this simple scheme allows us to handle multiple basic block S-Regions with single-entry points. For regions with multiple entry points, all entries in the PC Table are invalidated before filling the S-Buffer with the new S-Region.

Branching in the Static Mode: Branches in the static mode by default are predicted as 'always taken'. The Branch Table shown in Figure 6.3 is used for storing the target address of the branch instruction. Entries in the branch table are created when the branch is decoded. Each entry holds the PC of the branch and the S-Buffer line number of the target instruction. The branch table is content addressable with an S-Buffer line number. When issuing an S-Buffer line containing a branch instruction (indicated by the B-bit), the branch table is accessed and the target S-Buffer line number is obtained. Subsequent issue of instructions is performed from this S-Buffer line. If the branch is not-taken, as soon as the branch has been resolved in the execute state, the instruction packet in the decode stage is squashed. The PC Table is searched with the computed target address and if there is no valid entry corresponding to the address, the processor exits the static mode and returns to the superscalar mode of execution. This branching scheme, similar to the IBM NorthStar branch prediction scheme [10], provides zero-cycle branch instructions in the taken case, and a unit cycle latency for the not-taken case (when the target instruction is in the S-Buffer). Hence, this scheme is particularly suited for branches that are highly biased in the 'taken' direction. The compiler employs *if-conversion* [3] to eliminate unbiased branches wherever possible.

Exiting the Static Mode: When execution switches from static mode to the dynamic mode on a branch, the instruction cache is accessed with the target of the branch. The processor waits until the last instruction in the static mode has completed (after which the correct values of registers are available in the architectural register file), before execution resumes in the dynamic mode.

In-order Retirement and Exceptions: In the static mode, instructions are issued in packetized groups and also commit as a group. This feature offers the opportunity to design a low-power reorder buffer with few ports and low associativity. Therefore, rather than using the reorder buffer present in the superscalar mode, the scheme provides a separate buffer for the static mode called the *Exception Handling Unit* (EHU) to support in-order commit of instructions and for handling precise exceptions in the static mode. The EHU is similar to the reorder buffer with future file structure proposed by Ozer *et al.* [45] for VLIW processors.

The structure of the exception handling unit is shown in Figure 6.4. Each line in the EHU contains only as many instructions as the issue width of the processor. The EHU is a circular buffer consisting of a head and a tail pointer, which point to the first and next available entries in the buffer, respectively. The maximum buffer length is the longest latency operation plus one [45].

When an instruction packet is issued, the buffer line number is placed in the line number field of the EHU entry pointed to by the tail pointer and the destination register numbers are written into the Dest Reg field. In the static-mode, the operations write results into the future file and also read operands from it. The future file is a replica of the architectural register file. This implies that when the execution modes switch from the superscalar to

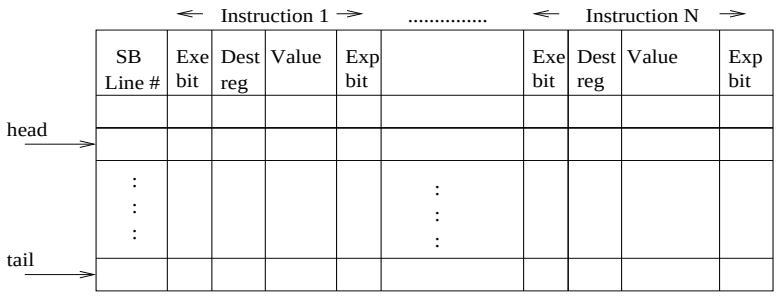


Figure 6.4: Exception handling unit for static mode. N is the number of instructions issued every cycle.

static mode, the register contents of the architectural register file need to be copied to the future file.

When an operation executes without an exception, the result is written into the future file, its Exe bit in the EHU is set and the result value is written in the Value field. If any instruction causes an exception, the Exp bit is set. At each cycle, the entry pointed to by the head pointer is examined. If all the Exe bits are set and all the Exp bits are clear, the results are written into the architectural register file. If an exception occurs in one of the instructions, instruction issue is stalled, the architectural register file contents are overwritten onto the future file and the EHU contents are discarded. The S-Buffer line number which caused the exception is extracted and the instruction causing the interrupt is reported to the operating system. Re-execution of instructions resumes from the first instruction in the packet.

The structures introduced for the static mode execution are inherently low energy structures due to small sizes, low associativity and fewer port requirements. These simple structures replace the complex, power-hungry hardware units such as the issue window and reorder buffer. A detailed evaluation of the scheme shows that the Hybrid-Scheduling scheme can provide tremen-

dous energy savings in generic superscalar processors with minimal performance loss.

6.4 Experimental Setup

This section describes the experimental setup used for evaluating the Dual-mode Hybrid-Scheduling scheme.

6.4.1 Benchmarks

The evaluation suite includes several media and scientific benchmarks. Five audio and video compression/encoding applications in the Mediabench [38] suite (*adpcm, epic, g.721, jpeg, mpeg2*) and several media kernels (*iir, add, scale, autocorr, fir, dct*) are examined in this study. Further, two scientific applications from the SPECFP suite of benchmarks (*swim, tomcatv*) also included in the benchmark suite. All benchmarks except *tomcatv* are run to completion. For *tomcatv*, the first 100 million instructions are skipped and run for a billion instructions.

6.4.2 Evaluation framework

The framework used to evaluate the Dual-Mode Hybrid-Scheduling scheme is shown in Figure 6.5. Benchmarks are compiled on a DEC Alpha machine with the *cc* and *g77* compilers. The benchmarks are compiled with the highest optimizations; optimizations such as loop unrolling, if-conversion, software pipelining, prefetching were applied semi-automatically to create compact schedules. By default all loops are unrolled four times. In some cases, higher unroll factors provided better performance benefits. Software pipelining is applied in

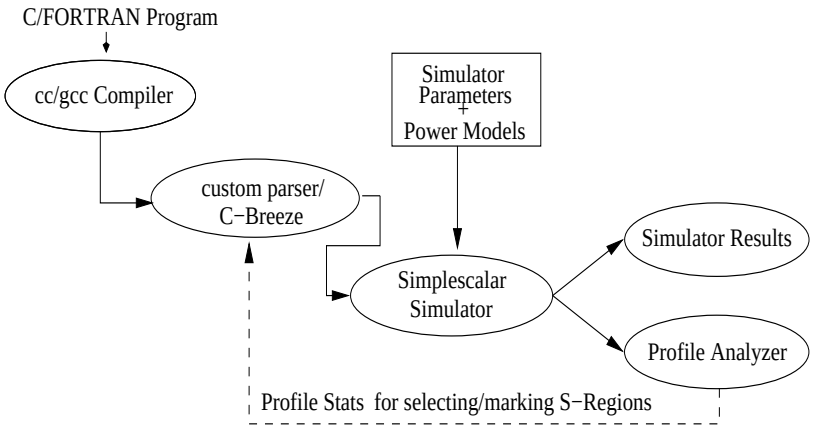


Figure 6.5: Framework for evaluating the Dual-Mode Hybrid-Scheduling scheme.

many of the S-Regions in media programs. The scientific benchmarks were not unrolled or software pipelined since they were very large and exhibited sufficient ILP even without unrolling. If-conversion is applied by the compiler using conditional move instructions. Only two major S-Regions (main loop in adpcm and sum-of-absolute-differences (SAD) loop in mpeg2) required predication. The compiler adds a prefetch instruction for all load instructions within loops. All load and store instructions are scheduled assuming they hit in the cache. Misses, if any, are handled by hardware interlocks.

All loops without function calls were considered to be potential S-Regions. Loops in the programs were identified using custom parsers (for FORTRAN programs) and C-Breeze [27], an academic source-to-source compiler (for C programs). Selected S-Regions are annotated with special "start-static", "end-static" and "start-block" instructions.

The Dual-Mode Hybrid-Scheduling architecture is implemented within the Wattch 1.0 simulator [11] framework. Wattch is an architectural-level simulator for estimating CPU energy consumption. Wattch is based on Sim-

Table 6.1: Processor configuration

Feature	Attributes	Feature	Attributes
IW/LSQ	64/32 entries	S-Buffer	128 rows, 194 bits/row
ROB	64 entries	EHU	16 rows, 284 bits/row
Width	4-way	PC/Br Table	16/10 entries
Branch-Predictor	Combination predictor 4K Gshare + 2K bimod	BTB	512 entries,4-way
L1 Dcache	64K 4-way 1-cycle	IALU (4 units)	1-cycle latency
		FPALU (4 units)	2-cycle latency
L1 Icache	32K DM 1-cycle	IMult (2 units)	3-cycle mult lat. 10-cycle div lat.
L2 Cache	512KB, 4W 10-cycle	FPMult (2 units)	3-cycle mult lat. 15-cycle div lat.
Memory	100-cycle lat	LD/ST (2 units)	1-cycle

plescalar's cycle-accurate out-of-order issue simulator *sim-outorder* [13]. Details about the Wattch simulator are provided in Chapter 4 and are also available in the Wattch technical paper [11]. Complete configuration details of the simulated processor are given in Table 6.1. The default simulator has only five pipeline stages. An additional 30 cycles penalty is added to the switching overhead to account for the impact of deeper pipelines. The base processor has an issue width of four. Power distributions for different hardware structures in the base processor are shown in Tables 6.2 and 6.3. The power models corresponding to the 0.18μ process at 2V supply voltage and 1GHz operating frequency are used. Unused units dissipate 10% of their maximum power [9]. The power breakdowns in Table 6.2 represent the maximum power per unit. The total processor power for the baseline configuration was 77W. In the baseline configuration, benchmarks dissipated anywhere from 29W to 41W average power per cycle. Table 6.3 shows the average power distribution among different structures based on the activity factors of the benchmark *jpeg*.

Table 6.2: Power distribution for different hardware structures in the baseline processor.

Unit	Power	Unit	Power
BPred	4%	Rename	1%
IW/LSQ	10%	ROB	13%
Reg. File	3%	Res. Bus	3%
Func. Units	15%	ICache	3%
DCache	7%	Clock	37%
L2 Cache	4%	Total	100%

Table 6.3: Activity-based power distribution for *jpeg*.

Unit	Power	Unit	Power
BPred	2%	Rename	1%
IW/LSQ	18%	ROB	18%
Reg. File	1%	Res. Bus	5%
Func. Units	8%	ICache	6%
DCache	6%	Clock	34%
L2 Cache	1%	Total	100%

Table 6.4: Static mode structures

Unit	Entries	bits/row	Ports	Max access	Assoc. access	Power (Watts)	Power(% of total)
S-Buffer	128	194	1R/1W	1/cyc	No	0.6	0.8%
EHU	16	284	1R/5W	6/cyc	Partial	0.86	1.1%
Future File	32	64	8R/4W	12/cyc	No	2.24	3%

The static mode structures are modeled using the RAM and CAM models provided by Wattch 1.0. The static mode also supports execution of only four instructions per cycle. The structures introduced for this mode are inherently low energy structures due to small sizes, low associativity and fewer port requirements. More details of the structures introduced are given in Table 6.4. The size of the S-Buffer needs to be large enough to hold the largest

S-Region in the programs. For the benchmarks studied, the lengths of different S-Regions varied from 10 lines to 34 lines. Hence, the size of the S-Buffer is set to 128 lines to easily accommodate the largest S-Region. Each S-Buffer line contains 4 instruction entries. Each line in the buffer also holds a PC and two state bits (a total of 194 bits). The S-Buffer is accessed only once per cycle, either during fill or during issue.

The size of the exception handling unit is also small, since it needs to be only one entry longer than the longest latency operation [45]. Instructions access the EHU (exception handling unit) associatively during instruction writeback. However, unlike the reorder buffer, during issue and commit only a single entry is accessed [45]. Each row in the EHU contains four instruction entries. Each entry holds one result value, a destination register number and two state bits. Hence, each entry of the EHU contains 284 bits. The maximum power consumed by the EHU is 1% and the future file, which is similar to the register file, is 3% of the overall processor power. Note that the since the register file is updated from the ROB in the dynamic mode and from the EHU in the static mode; the total number ports on the register file remains unchanged. The PC and branch tables are small structures since only a few S-Regions are placed in the S-Buffer. They account for only 0.3% of the total processor power.

6.5 Workload characterization

Table 6.5 shows the characteristics of the programs in terms of S-Regions. Columns 2 and 3 show the total number of loops and the number of loops without function calls, or potential S-Regions in the benchmarks. Columns 4 and 5 show the number of S-Regions that were dynamically invoked and the

Table 6.5: S-Region characteristics in media and scientific applications

Bench -mark	Static Info.		Dynamic Info.		Duration			Regular Memory	
	Num of Loops	# w/out Function Calls	Num of Loops	% Time in S-Regs	Avg. Dura-tion	Num of S-Regs	%Time in S-Regs	Num of S-Regs	% Time in S-Regs
ADPCM	7	2	1	99%	17607	1	99%	1	99%
EPIC	75	49	18	87.8%	1.2e6	11	87.1%	11	87.1%
G.721	9	6	5	49.6%	84	0	0%	0	0%
MPEG2	109	77	77	88%	6107	32	75%	32	75%
JPEG	317	162	20	50%	6403	4	31%	4	31%
SWIM	16	14	14	93%	3.3e8	14	93%	2	67%
TOMC	9	6	6	93%	2.1e7	6	93%	4	71%

total time spent in the regions. The table shows that in these applications, a significant number of loops qualified as potential S-Regions and a considerable amount of program execution time was spent in potential S-Regions.

Column 6 in Table 6.5 shows the average duration of the potential S-Regions. The duration shown is the weighted average, where the weights are determined based on the percentage of program time attributed to a region. Figure 6.5 shows the distribution of S-Regions in the program in decreasing order of the program time spent in the S-Region. Typically in each program there are three or four dominant S-Regions. The average duration of the S-Regions in each program is typically determined by these dominant regions. On closer examination the results revealed that except in G.721, most of the dominant S-Regions were over 3000 cycles in duration (corresponding to approximately 1% performance loss when the switching overhead is 20-30 cycles). Columns 7 and 8 show the number of S-Regions eliminated due to short durations and the time spent in the remaining S-Regions respectively.

In this study, $PD_{threshold}$ is set to 10%. However, the largest penalty observed was only 7% (ADPCM). Examining the schedules further showed that the limitation that schedules generated by the compiler are restricted

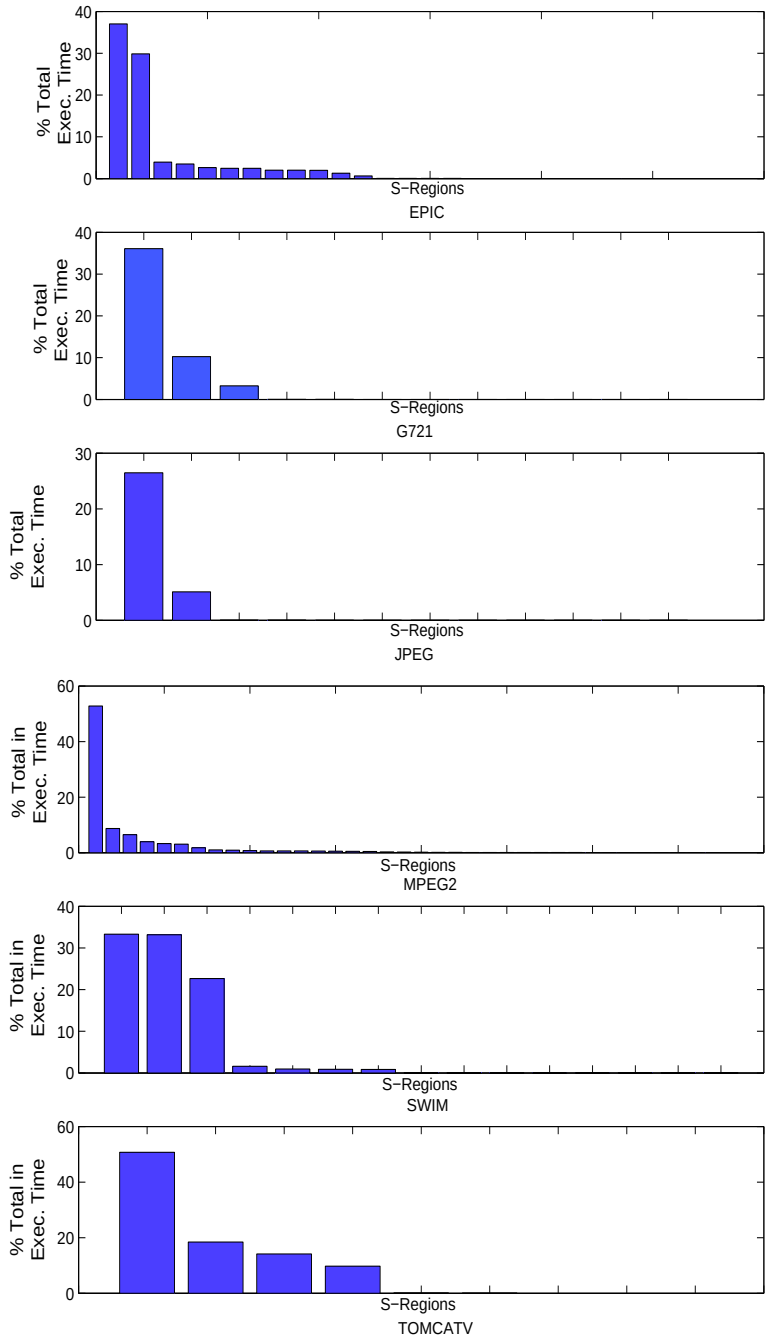


Figure 6.6: Distribution of S-Regions in decreasing order of the program time spent in the region.

to integral values caused significant performance penalty. The performance degradation caused due to rounding the lengths of schedules to integer values can be significant but is considerably reduced by unrolling (and/or software pipelining) the loop [36]. There were very few false dependences introduced due to register reuse. There were sufficient registers to assign each instance of a variable in the unrolled version (or software-pipelined version) to different destination registers to eliminate false dependences. Further, there were sufficient registers to do this without introducing any spill code. The Alpha ISA has 32 general-purpose integer registers, which are sufficient for the compiler to harness the ILP statically. In architectures with a fewer number of architectural registers (e.g: x86), the impact of false dependences will be higher than what is observed in this work and potentially more S-Regions will be eliminated due to this. One possible solution for such architectures is to apply the Hybrid-scheduling scheme within a dynamic optimization framework [50], wherein the dynamic optimizer creates schedules using the physical register file.

Columns 9 and 10 show that most of the S-Regions had regular access patterns for which simple prefetching schemes were sufficient to hide the memory latencies. However, in the scientific programs, few large S-Regions were eliminated due to frequent memory misses. With more sophisticated prefetching schemes, these S-Regions could potentially be included for static mode issue as well. Final S-Regions are selected such that the sum of the performance degradation due to switching overhead, imperfect schedules and memory misses, is less than 10%. However, the maximum performance drop was limited to only 7.7% in ADPCM. Column 9 thus indicates the final number of S-Regions selected by the compiler and column 10 shows the percentage of time spent in these regions.

6.5.1 Variability with input

Media and scientific programs typically have very regular control flow and regular memory access patterns that shows little variability across different inputs. Input data however directly controls some of the iteration counts in the dominant loops and hence the duration of the S-Regions could potentially be sensitive to input data. A total of five input sets were used for profiling multimedia programs and three sets for the scientific programs. For media programs, particularly small images and audio inputs were included to determine the sensitivity of the results to input data. For the floating point benchmarks the smallest inputs were the *test* inputs in the SPEC suite. Note, Table 6.5 shows the data for the default input sets for media and *test* inputs for scientific benchmarks. The main observations are summarized as follows:

- In several media programs, the duration of selected S-Regions were completely independent of input data. These programs included APDCM, G721 and MPEG2.

- The duration of S-Regions in image processing benchmarks JPEG and EPIC showed some dependence on input data. However, even with the smallest image used (32x96 pixels), the durations of the dominant regions remained as high as 1,311 and 45,269 cycles respectively.

- In a few media benchmarks, the percentage of time in S-Regions reduced with very small input sets, the largest difference was in EPIC were the percentage of time spent in S-Regions decreased from 87% to 67%. In most benchmarks, the time spent in S-Regions remained within 5%-10% of the values shown in Table 6.5 for any input. In a few benchmarks, increasing the input data size, increased the percentage time spent in S-

Regions considerably (for example, in EPIC, the time spent in S-Regions increased from 87% to 94% for a large 1184x1760 pixel image).

- In scientific programs, the average durations were very high even with smallest *test* input sets and with train and reference inputs, the average durations increased further.

6.6 Experimental results

Figure 6.7 provides the energy and energy-delay results for all benchmarks. The corresponding performance degradation suffered by the programs is given in Figure 6.8. The figure shows that the Dual-Mode Hybrid-Scheduling scheme is able to achieve very large improvements in energy consumption without any significant increase in the execution time.

In kernels, the average energy improvement is seen to be 35%, with the improvement ranging from 34% (`iir`) to 37% (`dct`). On average, the energy-delay product improves by 33%. The energy improvement and performance results for the applications are also included in Figures 6.7 and 6.8. On average, in applications, we observe an energy improvement of 20% and a performance degradation of 4.3%. In the applications, the energy savings are directly proportional to the amount of time spent in S-Regions. The highest improvement in energy is seen in ADPCM (35%). JPEG shows the lowest savings (8%).

The energy improvements seen are primarily due to the savings in the issue window and reorder buffer power. Additional power savings are observed in the fetch and decode phases of the pipeline. Since static mode instructions are accessed from the S-Buffer which is significantly smaller than the instruc-

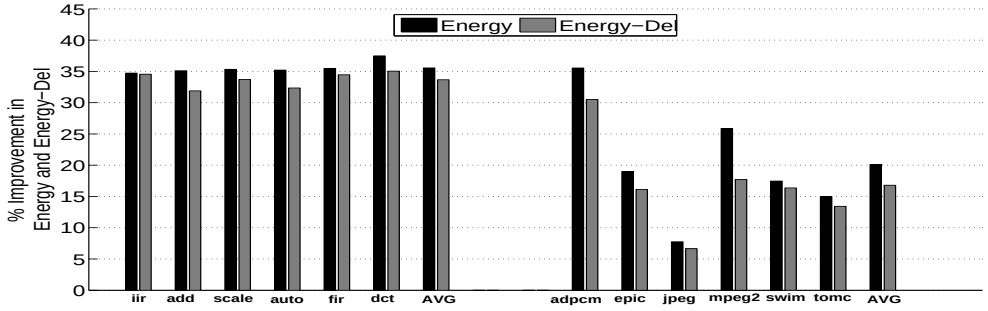


Figure 6.7: Energy and Energy-Delay improvements

tion cache, fetch power reduces considerably. Further, since instructions in static mode are not renamed, rename power is also saved in the static mode. Additionally, since the branch predictor is not accessed in the static mode, this leads to further energy savings. Figure 6.9 shows the energy savings in each hardware structure. Energy savings in the clock nodes of the structures is shown separately. Note that these are not absolute values but only portray the ratio of savings from each structure.

The performance degradation suffered by an application depends on the nature of the loop schedules, static mode cache misses and the switch-

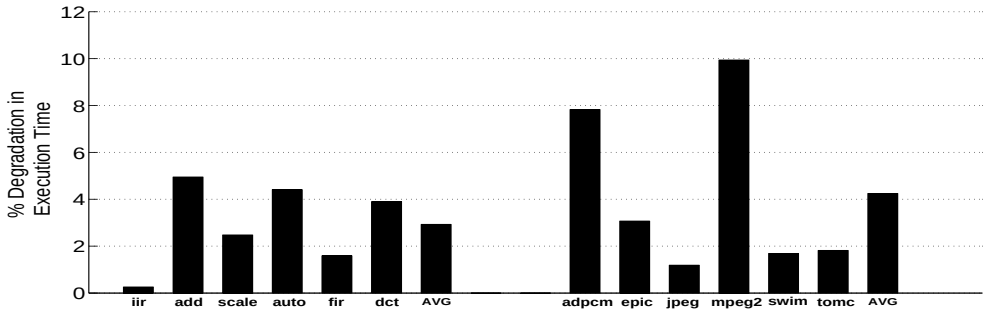


Figure 6.8: Performance degradation in the benchmarks

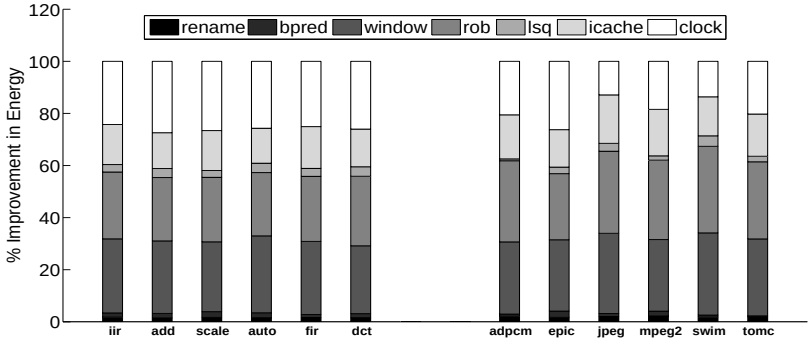


Figure 6.9: Energy improvements in different hardware structures

ing overhead incurred. The average performance degradation caused by the Hybrid-Scheduling approach is 3.6%. The highest performance drop in kernels (4.95%) is observed in `add` and the lowest is 0.26% seen in `iir`. The highest performance drop in applications was observed in MPEG2 (9.9%) and lowest performance penalty was seen in JPEG (1%). As described in Section 5.1, in most benchmarks the key constraining factor was the limitation that the schedules generated by the compiler are restricted to integral values. In most of the benchmarks, the performance penalty due to cache misses and switching overhead was limited to less than 1%. In MPEG2, the switching overhead accounted for approximately 3% performance loss. The largest S-Region in MPEG2 was only 800 cycles long.

6.6.1 Combining the generic and the Dual-Mode Hybrid-Scheduling microarchitectures

With nominal increase in design complexity and chip area, the generic and Dual-Mode Hybrid-Scheduling micro-architectures can also potentially be combined as shown in Figure 6.10. The resulting Hybrid-Scheduling architecture

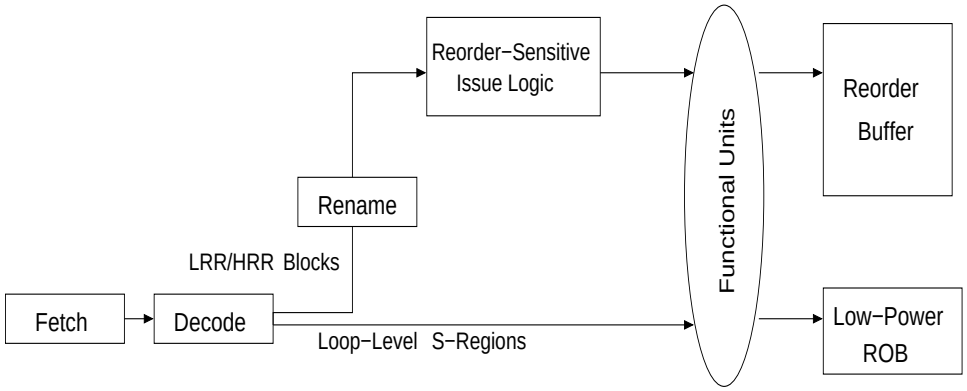


Figure 6.10: A Multi-Mode Hybrid-Scheduling scheme

replaces the out-of-order issue queue in the Dual-Mode Hybrid-Scheduling scheme with the reorder-sensitive issue queue to support three types of program regions: loop-level S-Regions, Low Reorder Required (LRR) blocks (basic-block level S-Regions) and High-Reorder Required (HRR) blocks (dynamic blocks) for larger overall energy savings. Such a trade-off can be particularly useful in general-purpose desktop systems which cater to diverse application domains such as integer, media and scientific. By routing instructions within a program region to scheduling engines tuned specifically to the region's inherent dynamic reordering requirements, the multi-mode Hybrid-Scheduling approach can thus provide substantial reduction in processor energy consumption while concurrently delivering high levels of performance.

6.6.2 Hybrid-Scheduling versus in-order issue

Table 6.6 compares and contrasts the Dual-Mode Hybrid-Scheduling architecture with a complete in-order issue processor. The results show that while the energy savings in an in-order processor of execution might be high, the performance degradation suffered is also significantly large. The main reason for the

Table 6.6: Hybrid-Scheduling approach versus in-order execution. All values are normalized with respect to out-of-order execution results.

Benchmark	Normalized Execution Time			Normalized Energy Consumption		
	OOO	Hyb-Sched	In-order	OOO	Hyb-Sched	In-Order
ADPCM	1.00	1.08	1.08	1.00	0.71	0.61
EPIC	1.00	1.04	1.35	1.00	0.70	0.59
JPEG	1.00	1.01	1.27	1.00	0.90	0.61
MPEG2	1.00	1.06	1.14	1.00	0.87	0.60
SWIM	1.00	1.03	1.70	1.00	0.74	0.67
TOMC	1.00	1.02	1.60	1.00	0.84	0.66

performance degradation in the media programs is short basic blocks created by frequent function calls in the non-S-Regions of the programs. The scientific programs typically have fewer function calls and larger basic blocks, however, the primary reason for the performance degradation is the large number of memory misses seen in the non-S-Regions. The results in Table 6.6 further indicate that partitioning a program into different execution regions as is done in the Hybrid-Scheduling architecture and mapping the regions to hardware units based on their requirements is critical for improving the power-performance balance modern superscalar processors.

6.6.3 Comparing Dual-Mode Hybrid-Scheduling with dynamic resource adaptation schemes

Recent work in the area of energy-effective microprocessors has shown that it is possible to reduce power in the out-of-order issue units and other units by dynamically resizing the structures [5][6][14][22][25][47][51]. The basic philosophy of these approaches is that the resource requirements of a program changes during its execution. Almost ubiquitously, changes in IPC (instructions per cycle) are used to estimate the resource requirements of the program

Table 6.7: IPC of different program regions

Benchmark	Average IPC	S-Region IPC	IPC of rest of the prog.
ADPCM	3.97	3.99	1.29
EPIC	2.44	2.61	2.05
G.721	2.48	0	2.48
MPEG2	3.38	3.55	2.82
JPEG	3.02	3.99	2.80
SWIM	3.53	3.99	2.98
TOMC	2.98	3.99	2.45

during run-time [5][14][22]. IPC is monitored and processor resources such as the issue window, reorder buffer, issue width are changed based on a predetermined FSM (Finite State Machine). Typically, when IPC is below a certain threshold, the number of resources (example: issue queue entries) are reduced. If the IPC increases, the number of entries are increased. The assumption is that the program is now in a region containing inherently high amount of ILP, and hence can benefit from more resources.

Table 6.7 shows the IPC of the programs studied, along with the average IPCs for S-Regions and the rest of program. The table shows that the S-Regions exhibit high IPCs compared to the rest of program. For these high ILP regions, the Dual-Mode Hybrid-Scheduling technique completely *eliminates* the use of the processor resources such as the instruction window and reorder buffer. The dynamic resource adaptation schemes on the other hand employ *larger* number of resources for the high ILP regions leading to less energy savings. Therefore, the proposed scheme does particularly well for regions where the IPC-based dynamic adaptation schemes fail to benefit from resource reduction. Furthermore, these dynamic resizing methods could potentially

Table 6.8: Resource occupancy of different regions in programs

Benchmark	ADPCM	EPIC	G.721	MPEG2	JPEG	SWIM	TOMC
Avg. ROB	63.46	23.38	19.71	28.91	33.53	58.69	59.85
S-Region ROB	63.8	23.48	0	29.6	63.79	68.3	63.68
Rest of Prog.	12.8	23.15	19.71	26.6	26.63	53.09	57.84
Avg. IQ	52.0	11.0	9.8	19.30	23.34	34.93	26.18
S-Region IQ	52.4	12.7	0	20.42	51.19	44.96	44.55
Rest of Prog.	4.7	8.6	9.8	15.9	16.99	22.69	16.53

be applied in the superscalar mode of execution in the Hybrid-Scheduling approach leading to larger overall savings in energy consumption.

A different approach to dynamically reconfigure processor resources is based on monitoring the occupancy of the different queues in the processor [47][18]. These techniques directly monitor the occupancy of structures such as the issue queue, load-store queue and reorder buffer to reconfigure their sizes. Table 6.8 shows the resource occupancy of the S-Regions and the rest of the program. The table shows that the issue queue and reorder buffer occupancy of the S-Regions is much higher than the rest of program. These occupancy-based reconfigurable techniques can also be applied in the superscalar mode of execution in the Hybrid-Scheduling approach for greater overall energy savings.

6.7 Summary

This chapter introduced a Hybrid-Scheduling technique specifically targeted to media and scientific applications. The compiler analysis and the microarchitecture for these applications is considerably simpler than the generic Hybrid-Scheduling scheme proposed for a diverse set of applications. In this scheme, regular regions such as loops, bypass the power-hungry units such as the issue

window and reorder buffer and execute in a low power static mode. Execution in the static mode also results in additional energy savings in the decode logic, instruction cache and branch predictor. The Dual-Mode Hybrid-Scheduling technique can reduce energy consumption by as much as 37% for kernels and up to 35% in full-length applications with minimal performance degradation.

Chapter 7

Related Research

This chapter briefly reviews previous contributions in areas related to the Hybrid-Scheduling scheme. These include compile-time and run-time techniques to reduce issue queue power, techniques that reuse dynamic schedules and complexity-effective issue queues.

7.1 Compile-time techniques to reduce issue queue power

A contemporaneous work similar to the Hybrid-Scheduling scheme was proposed by Jones *et al* [31]. Their software-directed issue scheme estimates the size of the issue queue required for each basic-block using compiler analysis. At run-time, the scheme uses the compiler-directed hints to resize the issue queue. One of the main differences between our approach and theirs is that rather than resizing the existing queue, we add a separate low power queue to the conventional out-of-order issue logic. An advantage of adding a separate queue is that several compiler-directed and dynamic resizing schemes [5, 22, 47, 30, 16, 61, 31], including the one suggested by Jones *et al.*, can be applied to the out-of-order issue queue for much larger overall savings in power. The *Cool-Fetch* scheme proposed by Unsal *et al.* [61] performs compiler-controlled adaptation of the fetch engine at loop boundaries. While this technique does not directly target the issue queue, the expected reduction in the number of

instructions fetched, indirectly reduces power consumption in the issue queue and other out-of-order issue units.

7.2 Run-time techniques to lower issue queue power

Run-time energy saving schemes typically lower the energy consumption in the issue queue by adapting the size of the queue based on the dynamic requirements of the program [5, 14, 16, 22, 30, 47]. These techniques usually sample measurable metrics such as IPC and issue queue occupancy to estimate the computational demand of programs and to guide adaptation. To ensure accuracy in detecting changes in computational demand, the sampling intervals are usually quite large [22, 47, 30]. In contrast, compile-time approaches have been shown to be better at adapting at smaller intervals [31, 61, 63]. For example, the techniques suggested by Unsal *et al.* [61] and the proposed dual-mode Hybrid-Scheduling scheme for media/scientific applications perform adaptation at loop boundaries. The Hybrid-Scheduling scheme and Jones *et al.* [31] resize the queue at an even finer granularity (at basic-block boundaries) using compiler hints.

Seng *et al.* [51] suggest a technique to reduce power by using an in-order queue and an out-of-order issue queue for critical and non-critical instructions respectively but do not reuse compiler schedules or support as in the Hybrid-Scheduling scheme. They use dynamic critical path information to steer instructions.

A few approaches use profile-time information to reconfigure dynamic issue hardware for energy savings. Iyer et al. explore a technique which profiles

different characteristics of a program such as ALU usage, register file usage, and instruction window usage [30]. Hotspots in the program are detected and processor units are scaled accordingly. Ghaisi *et al.* [25] propose a technique wherein the operating system dictates the expected IPC of a program (or even for different phases of the program) and allows the hardware to choose between different processor configurations such as pipeline-gating, in-order issue and out-of-order issue. Chi *et al* [16] propose a technique to combine hardware monitoring and software profiling to adapt microprocessor resources for power reduction.

7.3 Reusing dynamic schedules

Talpes *et al.* [57] suggest an approach that collects schedules created by the dynamic issue logic into a large trace cache and reuses them to save issue queue power. Franklin *et al.* [23] and Nair *et al.* [44] proposed similar schemes that were primarily aimed at improving the clock frequency of the processor. These techniques are insensitive to the available ILP in the various phases of programs, resulting in inefficient use of the caches holding the scheduled instructions. These techniques require large caches to store scheduled groups of instructions (nearly 100KB). In future processors, where leakage power is projected to equal active power, techniques such as the Hybrid-Scheduling scheme are more promising.

7.4 Complexity-effective issue queues

Another related direction of research focuses on designing complexity-effective issue queues. These techniques attempt to reduce the complexity of crit-

ical issue logic tasks such as instruction wakeup and select. A majority of the previously proposed techniques reduce complexity of the issue queue by limiting the number of candidate instructions to be considered for issue [1, 15, 20, 41, 42, 46, 48, 37]. These techniques typically consist of a *pre-scheduling* phase wherein the data-dependences of instructions are analyzed. Instructions are typically held in a separate buffer and are considered for issue in their approximate data-flow order [1, 15, 20, 41, 42, 48, 37]. In some cases, after the pre-scheduling phase, instructions are steered to various low-complexity FIFOs based on their dependences with older instructions in the queues [1, 46, 48]. While these techniques indeed alleviate the complexity of the issue logic, they often require extra hardware and/or the addition of a few pipeline stages. In the Hybrid-Scheduling work, since the necessary analysis is performed statically, we shift the burden to the compiler and thereby eliminate the need for any auxiliary hardware resources or pipeline stages.

Past research has suggested the use of a software layer to replace the out-of-order issue logic. The DAISY [19] and Crusoe [34] architectures from IBM and Transmeta, respectively, consist of a VLIW processor core logically surrounded by a software translation layer. The software converts binaries of any chosen architecture into VLIW instructions of the native processor. The main limitation of this scheme is that a considerable portion of the execution time is spent in running the translation software, making it nearly impossible to implement sophisticated compiler optimizations within the translator. Further, in cases where the ILP is not visible at compile-time, conservative decisions taken by the translation software could lead to significant performance degradation on a target machine that allows only the VLIW mode of execution.

Chapter 8

Future Work

This dissertation introduces a novel Hybrid-Scheduling paradigm that synergistically combines the benefits of both the compiler and the micro-architecture. The applications and opportunities for such a paradigm remain vast and largely unexplored. Listed below are several areas that look especially attractive for future exploration.

8.1 Using compiler assistance to lower power dissipation in various hardware units

The Hybrid-Scheduling scheme reduces energy consumption in the issue queue by reusing instruction scheduling work done at compile-time. There are a host of other hardware structures that can be made energy-efficient by harnessing compile-time intelligence. Value prediction hardware is one such example. Recent work has shown that while value prediction is an effective architectural technique for improving performance, the hardware tables used for recording the run-time history of data values and predictions consume a significant amount of power [8]. One of the primary reasons these structures consume energy is that they are highly multi-ported. In typical value predictors, although not many instructions show proclivity for value prediction, all instructions access the tables, thereby increasing the total switching activity. The compiler can help alleviate the power dissipation in these tables by providing

information regarding which instructions can truly benefit from value prediction. For example, address increment instructions in loops operating on array structures often increment the address register by a value of four. These instructions thus exhibit a high prediction accuracy. During the code generation phase, the compiler can mark instructions that perform address generation. By annotating these instructions at compile-time and allowing only a select few instructions to access the run-time hardware tables, energy consumption can be significantly lowered.

Another hardware structure for which compile-time assistance could potentially reduce energy consumption is the physical register file. The physical register file is typically much larger than the architectural register file (more than twice). However, as observed in Chapter 6, many regular program regions do not require hardware register renaming. In regions such as loops, the compiler can perform efficient software renaming with the architectural registers. The compiler can identify such program regions by evaluating the impact of false register dependences on the region schedule. For regions that are not expected to suffer significant performance degradation due to false dependences can potentially use only the architectural registers. A significant portion of the physical register file can then be turned off to save energy.

8.2 Hybrid-Scheduling in multi-core processors

A recent trend in the semiconductor industry is to offer multiple processing cores on a single chip. The focus on multiple cores arises from Moore's Law, which dictates that the number of transistors on a chip doubles every two

years. In the past, the extra transistors have been used to increase the size of the cache or to boost other ILP-enhancing features. However, employing the extra transistors to create additional cores to boost performance, without drastically increasing chips' power consumption.

Multiple cores open several new avenues for applying the Hybrid-Scheduling technique. In a multi-core scenario, it is possible to run different processors at different frequencies. A potentially energy-efficient processor design could have a combination of low-power, low-complexity issue queue based cores running at high frequencies, combined with associative queue based cores running at lower frequencies. Each core is designed to specifically cater to a different type of program region. Based on the particular dynamic phase, programs can be moved from one processing core to another more suitable core, for better power-performance trade-offs.

8.3 "Compile for power" switch

Current compilers already provide two axes in the optimizations used. Programs are typically "compiled for speed" (common in general-purpose processors) and/or "compiled for code size" (in embedded systems). With the increasing power dissipation and energy consumption concerns, it is worthwhile to offer users a third axis with optimizations where programs can be *compiled for power/energy*. The Hybrid-Scheduling technique is a key optimization that could be included in such an axis. Further, Chapter 2 in this dissertation evaluated several existing compiler optimizations for their impact on processor power and energy. The study noted that some optimizations such as common subexpression elimination are particularly suitable for lowering processor energy. A more comprehensive study can help identify other

such optimizations that can be included in an option where programs are compiled for power. The study will also help identify optimizations that require improvements when compiling for a system where power is among the primary concerns. Another interesting addition to this axis of compilation would be power-aware libraries. When compiled with this option, the user library calls can be directed to energy-efficient versions of the library routines to further reduce energy consumption in the processor.

Chapter 9

Conclusions

The Hybrid-Scheduling scheme developed in this dissertation is based on the key observation that all instructions and all basic-blocks in a program are not equal; some blocks are inherently easy to schedule at compile-time, while others are not. In conventional out-of-order issue superscalar processors, dynamic scheduling is performed for all program regions irrespective of the quality of the compiler-generated code. A detailed characterization of several general-purpose integer programs revealed that 20%-45% of program execution time can be attributed to basic-blocks classified as S-Regions, *i.e.*, regions for which the compiler can generate efficient schedules. Compile-time instruction-level parallelism (ILP) enhancing optimizations are particularly effective in media and scientific programs. In these programs nearly 30% to 99% program execution time (70% on average) can be ascribed to S-Regions. Thus, for a sizable portion of a program, the hardware issue logic typically expends energy performing superfluous dynamic reordering of instructions.

The Hybrid-Scheduling scheme exploits this inherent redundancy in a computer system to lower the power dissipation and complexity of the out-of-order issue units in the processor. In this scheme, energy is conserved by allowing S-Regions to bypass the dynamic issue hardware in the processor and execute on specially designed low-power, low-complexity hardware.

The proposed technique has both software and hardware components. At the software-level, the scheme includes a novel compile-time analyzer that

uses extensive profile-guided hints to evaluate the quality of schedules generated by the static scheduler. Basic blocks which contain a small number of anti- and output dependences, unresolved aliases and unknown load latencies are inherently amenable to compile-time scheduling and require less dynamic reordering. Such blocks are classified as S-Regions.

At the micro-architecture-level, S-Regions are directed to a novel issue queue that can harness compiler-generated schedules to save power. In the generic Hybrid-Scheduling architecture, basic-blocks that do not require dynamic reordering of instructions, *i.e.,* LRR (low reorder required) blocks, execute on simple low-power, low-complexity, First-In, First-Out (FIFO) queues. Non-S-Regions, *i.e.,* HRR (high reorder required) blocks are directed to a fully associative issue queue. The resulting issue logic is thus *Reorder-Sensitive*. A detailed analysis of the Hybrid-Scheduling architecture demonstrates that it is possible to save on average 70% of the issue queue energy with only 5% performance degradation. Further, the proposed issue hardware is less complex when compared to a conventional out-of-order issue queue, providing the potential for much higher clock speeds.

This dissertation also presents the Dual-Mode Hybrid-Scheduling architecture, which is a variation of the generic Hybrid-Scheduling architecture specifically developed for media and scientific applications. These programs contain long duration and contiguous S-Regions and hence offer the opportunity to massively lower hardware requirements. The Dual-Mode Hybrid-Scheduling architecture does not use two types of issue queues but instead supports two modes of execution. S-Regions bypass the out-of-order issue logic and execute in a VLIW mode wherein instructions are directly issued to the function units with minimal hardware support. In addition to saving a significant amount of energy in the issue queue, the Dual-Mode Hybrid-Scheduling

scheme exploits large granularity, loop-level S-regions to save energy in other structures such as the reorder buffer and the instruction cache. The scheme uses a low-power reorder buffer to support in-order retirement of S-Regions instructions and a low-power loop buffer to hold S-Region instructions. On average, the total energy saved in the Dual-Mode Hybrid-Scheduling architecture is a remarkable 35% with a negligible performance drop of 3.5%.

With nominal increase in design complexity and chip area, the generic and Dual-Mode Hybrid-Scheduling micro-architectures could also potentially be combined. The resulting multi-mode Hybrid-Scheduling architecture uses the reorder-sensitive issue queue in lieu of the out-of-order issue queue in the Dual-Mode Hybrid-Scheduling scheme to support three types of program regions: loop-level S-Regions, basic-block level S-Regions (LRR blocks), and dynamic blocks (HRR basic-blocks). The multi-mode Hybrid-Scheduling architecture could potentially considerably larger overall energy savings for a small increase in design complexity. Such a trade-off can be particularly useful in general-purpose desktop systems which cater to diverse application domains such as integer, media and scientific. By routing instructions within a program region to scheduling engines tuned specifically to the region's inherent dynamic reordering requirements, the multi-mode Hybrid-Scheduling approach can thus provide substantial reduction in processor energy consumption while concurrently delivering high levels of performance.

The Hybrid-Scheduling scheme is a cooperative hardware/software approach. The advantage of engaging the compiler is that the static scheduler has accurate information regarding the region to be executed in the near future. Hence, energy saving resource adaptations can be applied at finer granularities when compared to run-time hardware techniques [31][63][65]. Further, a compile-time approach offers opportunities to simplify the hardware beyond

what is achievable with typical hardware schemes. Additionally, a host of dynamic resource adaptation schemes such as : [14][22][25][30][47], can be applied orthogonally to the Hybrid-Scheduling technique for larger overall reduction in energy consumption.

Another key contribution of this dissertation is the SPHINX framework developed for evaluating the Hybrid-Scheduling technique. Good interaction between compilers and computer architectures is critical for designing highly efficient and effective computer systems. Effective compilers allow more efficient execution of application programs on a given computer architecture and well-conceived architectural features can support more effective compiler optimization techniques. The SPHINX tool provides a unified and user-friendly platform to enable future research in this important area of cooperative hardware/software schemes.

The applications and opportunities for a cooperative compile-time/micro-architectural approach are vast. This dissertation also briefly reviewed a few areas that look especially attractive for future exploration.

Bibliography

[1] J. Abella and A. Gonzlez. Low-complexity distributed issue queue. In *Proceedings of the 10th Annual International Symposium on High Performance Computer Architecture*, Feb 2004.

[2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, pages 248–259, Dec 1999.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Principles of Programming Languages*, Austin, Jan 1983.

[4] K. Asanovic. Energy-exposed instruction set architectures. In *Work In Progress Session, Sixth International Symposium on High Performance Computer Architecture*, Jan 2000.

[5] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 27th International Symposium on Computer Architecture*, Jul 2001.

[6] Y. Bai and R. I. Bahar. A dynamically reconfigurable mixed in-orderout-of-order issue queue for power-aware microprocessors. In *Proceedings of the International Symposium on VLSI*, Feb 2003.

[7] S. Banerjee, H. Sheikh, L. John, B. Evans, and A. Bovik. VLIW DSP vs. superscalar implementation of a baseline H11.263 video encoder. In

Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers, pages 1665 – 1669, Oct 2000.

[8] R. Bhargava and L. K. John. Latency and energy aware value prediction for high-frequency processors. In *Proceedings of the 16th International Conference on Supercomputing*, pages 45–56, 2002.

[9] S. Borkar. Design challenges of technology scaling. In *IEEE Micro*, pages 23–29, Jul/Aug 1999.

[10] J. Borkenhagen and S. Storino. 4th generation 64-bit PowerPC-compatible commercial processor design. IBM Server Group , White Papers, Jan 1999.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, Jun 2000.

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

[13] D. Burger and T. M. Austin. Evaluating future microprocessors: The simplescalar tool set. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1997.

[14] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Proceedings of the Workshop on Power-Aware Computers Systems*, Nov 2000.

[15] R. Canal and A. Gonzlez. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing*, pages 327–335, 2000.

[16] E. Chi, A. M. Salem, R. I. Bahar, and R. Weiss. Combining software and hardware monitoring for improved power and performance tuning. In *the Workshop on Interaction Between Compilers and Computer Architecture*, Feb 2003.

[17] L. DiCarlo. Putting New Life In Batteries. www.forbes.com/personaltech/2004/09/20/cx_ld_0920batteries.html, Sept 2004.

[18] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, Sep. 2002.

[19] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture*, Jun. 1997.

[20] D. Ernst, A. Hamel, and T. Austin. Cyclone: a broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 253–263, Jun 2003.

[21] J. A. Fisher. Trace scheduling: A technique for global microcode com-

paction. In *Proceedings of the IEEE Transactions on Computer, C-30m no.7*, pages 478–490, Jul 1981.

[22] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, Jun. 2001.

[23] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[24] J. Fritts and W. Wolf. Evaluation of static and dynamic scheduling for media processors. In *Proceedings of the 2nd Workshop on Media Processors and DSPs*, pages 34–43, Dec 2000.

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

[26] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the Design Automation Conference*, pages 726–731, 1998.

[27] S. Z. Guyer, D. A. Jimnez, and C. Lin. The C-Breeze compiler infrastructure. Technical report, Department of Computer Science, University of Texas, Austin, 2001.

[28] N. B. I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high

performance microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 70–75, July 1998.

[29] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. Technical report, Intel, Feb. 2001.

[30] A. Iyer and D. Marculescu. Run–time scaling of microarchitecture resources in a processor for energy savings. In *Proceedings of the Kool Chips Workshop*, 2000.

[31] T. Jones, M. O'Boyle, J. Abella, and A. Gonzalez. Software assisted issue queue power reduction. In *Proceedings of the 7th Annual International Symposium on High Performance Computer Architecture*, 2005.

[32] V. Kathail, M. Schlansker, and B. R. Rau. HPL Playdoh architecture specification: Version 1.0. Hewlett-Packard Computer Systems Laboratory, Oct. 1992.

[33] R. King. Optimizing Data Center Energy Efficiency Results In Cost Savings. http://thewhir.com/features/dce.cfm, Jun 2002.

[34] A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, Jan. 2001.

[35] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. In *ACM Computer architecture letters*, 2002.

[36] D. M. Lavery and W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 327–337, Dec 1995.

[37] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, Jun 2002.

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

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

[40] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and Superscalar compilation. In *The Journal of Supercomputing, Kluwer Academic Publishers*, pages 229–248, 1993.

[41] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Feb 2001.

[42] E. Morancho, J. M. Llaberia, and A. Olive;. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept 2001.

[43] T. Mudge. Power: A first-class architectural design constraint. In *IEEE Computer*, pages 52–58, April 2001.

[44] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processor by caching scheduled groups. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1997.

[45] E. Ozer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings, and T. M. Conte. A fast interrupt handling scheme for VLIW processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, Oct. 1998.

[46] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, Jun 1997.

[47] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 90–101, Dec 2001.

[48] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 318–329, Jun 2002.

[49] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[50] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 162, 2004.

[51] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[52] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. In *Western Research Lab (WRL) Research Report*, 2002.

[53] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, Jun 2003.

[54] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta. Improving quasi-dynamic schedules through region slip. In *Proceedings of the International Symposium on Code generation and Optimization*, pages 149–158, Mar 2003.

[55] C.-L. Su and A. M. Despain. Cache designs for energy efficiency. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 306–315, 1995.

[56] C. L. Su, C. Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Proceedings of the IEEE COMPCON*, Feb. 1994.

[57] E. Talpes and D. Marculescu. Power reduction through work reuse. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2001.

[58] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. In *IBM J. Research and Development, 46(1);5-27*, Austin, 2002.

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

[60] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Proceedings of the Power-Driven Microarchitecture*, Jun 1998.

[61] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Mortiz. Cool-fetch: A compiler-enabled IPC estimation based framework for energy reduction. In *ACM Computer architecture letters*, 2002.

[62] M. Valluri and L. John. Is compiling for performance == compiling for power? In *Chapter 6, in Interaction between Compilers and Computer Architectures, edited by Gyunggho Lee and Pen-Chung Yew, Kluwer Academic Publishers, 2001, ISBN 0-7923-7370-7*, Jan 2001.

[63] M. Valluri, L. John, and H. Hanson. Exploiting compiler-generated schedules for energy savings in high-performance processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, 2003.

[64] M. Valluri, L. John, and K. McKinley. Low-power, low-complexity instruction issue using compiler assistance. In *Technical Report: TR-040925-01, Laboratory for Computer Architecture, The University of Texas at Austin*, Nov 2004.

[65] M. Valluri, L. John, and K. McKinley. Low-power, low-complexity instruction issue using compiler assistance. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Jun 2005.

[66] V.Kathail, M.Schlansker, and B.R.Rau. HPL-PD architecture specification: Version 1.1. technical report HPL-93-80(R.1). Technical report, HewlettPackard Laboratories, Feb. 2000.

[67] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. Shen. Register renaming for dynamic execution of predicated code. In *Proceedings of the 7th Annual International Symposium on High Performance Computer Architecture*, Feb 2001.

[68] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look to the future. In *CoolChips Tutorial, An Industrial Perspective on Low Power Processor Design in conjunction with Micro-33*, Dec. 1999.

[69] V. V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures. In *Proceedings of the IEEE Transactions on Computers*, pages 268–285, Mar. 2001.

[70] P6 family of processors, hardware developer's manual. In *Intel Corporation*, sept 1998.

[71] Itanium 2 family of processors, hardware developer's manual. In *Intel Corporation*, Jul 2002.

[72] TRIMARAN: An Infrastructure for Research in Instruction-Level Parallelism http://www.trimaran.org/

[73] The Standard Performance Evaluation Corporation (SPEC) http://www.spec.org/

[74] Java technology http://java.sun.com

[75] Microsoft .NET technology http://www.microsoft.com/net/

[76] Jikes Research Virtual Machine (RVM) http://jikesrvm.sourceforge.net/

[77] Alpha Architecture Reference Manual. Digital Press, Boston, MA, 3rd edition, 1998.

Vita

Madhavi Gopal Valluri was born in Hyderabad, India on January 2nd, 1975 to Mr. Sivagopal Valluri and Mrs. Sarada Gopal Valluri. After living in Trivandrum for 6 years, she moved to Bangalore with her family. In the Fall of 1992, she began her undergraduate studies at Dayanada Sagara College of Engineering. After graduating with Distinction from D.S.C.E, she began her Graduate studies in the Supercomputer Education and Research Center at the Indian Institute of Science, Bangalore. She worked in the area of compilers for high-performance processors under the supervision of Prof. R. Govindarajan. She received a M.Sc (Engg) degree from I.I.Sc on completing her Master's thesis entitled "Evaluation of Register Allocation and Instruction Scheduling Methods in Multiple Issue Processors". In January 1999, Madhavi entered the Graduate School at The University of Texas at Austin to pursue her doctoral studies. During the summers of 2000 and 2003, she gained industry experience through internships at Hewlett Packard and Intel Corporation respectively. She married Vivekananda Vedula in December 2000. While pursuing her Ph.D, she also earned a Master of Science degree in Electrical and Computer Engineering in December 2004. Madhavi's graduate education was supported by University of Texas teaching and research assistantships. She is a student member of IEEE, IEEE Computer Society, ACM, and ACM SIGARCH.

Permanent Address: 531, 16th Main, 3rd Block, Koramangala,
Bangalore 560034

This dissertation was typeset with LaTeX 2_ε by the author.