8-31-2010

Architecture and Compiler Support for Leakage Reduction Using Power Gating in Microprocessors

Soumyaroop Roy
University of South Florida

Scholar Commons Citation

Roy, Soumyaroop, "Architecture and Compiler Support for Leakage Reduction Using Power Gating in Microprocessors" (2010). *Graduate Theses and Dissertations.*
https://scholarcommons.usf.edu/etd/3479

Architecture and Compiler Support for Leakage Reduction Using Power Gating in Microprocessors

by

Soumyaroop Roy

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Nagarajan Ranganathan, Ph.D.
Co-Major Professor: Srinivas Katkoori, Ph.D.
Hao Zheng, Ph.D.
Sanjukta Bhanja, Ph.D.
Natasha Jonoska, Ph.D.

Date of Approval:
June 8, 2010

Keywords: Compiler Directed Power Gating, Microarchitectural Techniques, Embedded Microprocessors, Multithreading, Multiprocessing, Multicore, Niagara, CGMT, FGMT, SMT, GCC, SUIF, MachineSUIF, M5

DEDICATION

To my family:

Parents, Alpana and Snehansu, and sister, Shreya

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my doctoral advisors Dr. Nagarajan Ranganathan and Dr. Srinivas Katkoori for their valuable guidance and support throughout my graduate school life. I would particularly like to thank them for their patience and belief in me and my work. Their knowledge, work ethics, and the professional standards they adhere to have been very intrumental in inspiring me to put my heart and mind into solving the problems that are presented in this disseration. I would also like to thank Dr. Hao Zheng, Dr. Sanjukta Bhanja, and Dr. Natasha Jonoska for taking the time to be in my doctoral committee and providing valuable suggestions to improve this manuscript. I am extemely grateful to the CSE department for providing me financial assistance throughout all my years. I would like to thank Dr. Rangachar Kasturi, Dr. Larry Hall, Dr. Ken Christensen, Dr. Sudeep Sarkar, Dr. Dmitry Goldgof, Dr. Rafael Perez, Yvette Blanchard, Alex Dashner, Theresa Collins, Catherine Burton, and others for all their help and support and making my stay at USF a remarkable and unforgettable experience. I would also like to thank my past and present colleagues at Computer Architecture and Nano VLSI Systems (CANS) Research Group, Himanshu, Koustav, Upavan, Ransford, Mahalingam, Ziad, Michael, Saurabh, Yue, Vidyasangkar, Elizabeth, Haiqiong, Sameer, Pradeep, Hari, Vyas, Jared, Narendra, and others for a fun and intellectually stimluating atmosphere in the labs. I would also like to thank other colleagues in the CSE department and friends, Sergiy, Matt, Kurt, Olya, Satrajit, Ravi, Anand, Vladimir, Pedro, Miguel, Daladier, Puneet, Noelia, Jessica, Jason, Erin, Tatiana, Vaso, Neil, Sara, Keri, and others. I am also very grateful to the efficient CSE tech support team, Daniel, Peter, Sridhar, and Brian and the engineering computing team for their help with IT related problems. I would also like to thank Korey Sewell of M5, Glenn Holloway of MachineSUIF, and members of the processor architecture laboratory, EPFL. Finally, I would like to thank my mother, Alpana, father, Snehansu, and sister, Shreya, for playing an extremely important part in making me the person that I am today. They provided me their unconditional love and support in my decision to travel several thousand miles westward from home for my doctoral studies and throughout my studies. I am also very thankful to my cousin, Saurabh, for being a beacon of brilliance since we were very young.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Architecture and Compiler Support for Leakage Reduction Using Power Gating in Microprocessors

Soumyaroop Roy

ABSTRACT

Power gating is a technique commonly used for runtime leakage reduction in digital CMOS circuits. In microprocessors, power gating can be implemented by using sleep transistors to selectively deactivate circuit modules when they are idle during program execution. In this dissertation, a framework for power gating arithmetic functional units in embedded microprocessors with architecture and compiler support is proposed. During compile time, program regions are identified where one or more functional units are idle and sleep instructions are inserted into the code so that those units can be put to sleep during program execution. Subsequently, when their need is detected during the instruction decode stage, they are woken up with the help of hardware control signals. For a set of benchmarks from the MiBench suite, leakage energy savings of 27% and 31% are achieved (based on a 70 nm PTM model) in the functional units of a processor, modeled on the ARM architecture, with and without floating point units, respectively. Further, the impact of traditional performance-enhancing compiler optimizations on the amount of leakage savings obtained with this framework is studied through analysis and simulations. Based on the observations, a leakage-aware compilation flow is derived that improves the effectiveness of this framework. It is observed that, through the use of various compiler optimizations, an additional savings of around 15% and even up to 9X leakage energy savings in individual functional units is possible. Finally, in the context of multi-core processors supporting multithreading, three different microarchitectural techniques, for different multithreading schemes, are investigated for state-retentive power gating of register files. In an in-order core, when a thread gets blocked due to a memory stall, the corresponding register file can be placed in a low leakage state. When the memory stall gets resolved, the register file is activated so that it may be accessed again. The overhead due to wake-up latency is completely hidden in two of the schemes, while it is hidden for the most part in the third. Experimental results on multiprogrammed workloads comprised of SPEC 2000 integer benchmarks show that, in an 8-core processor executing 64 threads, the average leakage savings in the register files, modeled in FreePDK 45 nm MTCMOS technology, are 42% in coarse-grained multithreading, while they are between 7% and 8% in fine-grained and simultaneous multithreading. The contributions of this dissertation represent a significant advancement in the quest for reducing leakage energy consumption in microprocessors with minimal degradation in performance.

CHAPTER 1

INTRODUCTION

Advances in integrated circuit (IC) technology have helped the semiconductor industry to keep pace with Moore's law for over five decades. Due to technology scaling, the minimum feature size has continued to shrink while the chip density as well as the transistor performance have continued to improve. This scaling trend has multiplied the complexity of VLSI circuits which, in turn, has increased the importance of power considerations in chip design. In high-performance microprocessors, power density limits have restricted the upward scaling of clock frequencies to achieve greater performance. In embedded microprocessors, high power consumption impacts the engineering feasibility of battery-powered portable devices as well as their reliability. Designers have to make a tradeoff between the size of the battery packs and the operating life of the devices. These issues have forced the designers to pursue low-power design methodologies in an aggressive manner.

1.1 Power and Energy Concern in Computing Systems

The market for embedded computing systems is proliferating at a tremendous rate. It is reported that more than one billion cell phones are sold each year and this market is ever expanding [1]. It is estimated that media devices such as cell phones, video cameras, and digital televisions perform more computations than desktops, laptops, and data centers and at power consumption rates that are orders of magnitude lower than those of the latter computation platforms. Efficient embedded processors and DSPs consume about 250pJ per operation [2], while laptop processors consume about 20 nJ per operation [3] indicating that embedded processors are about 40X more energy efficient than laptop processors. This is of paramount importance because embedded processors feature in portable and handheld devices and the packaging technology restricts the maximum power dissipation of these devices to about 1W [1]. Moreover, the battery life of a handheld device is one of its most prominent features that impacts its success as a product in the market. Therefore, low power and energy efficient design techniques are of paramount importance in the design of embedded processors.

In the domain of high-performance microprocessors, until recently, technology scaling was making it possible to build increasingly complex processor architectures with larger on-chip caches operating at higher clock frequencies. In recent years, however, increased concerns about power density and thermal effects have emerged as fundamental barriers that have severely restricted the upward scaling of clock frequencies for further performance improvement [4]. Apart from the benefits offered by technology scaling, advances in architectural design techniques have further improved the performance of microprocessors. Superscalar CPU architectures with multiple functional units were developed so that several instructions could be executed simultaneously within a single clock cycle. Deeper pipelines and dynamic scheduling to allow out-of-order execution of instruction streams within a single thread are employed to exploit instruction-level parallelism in the program. However, several complex hardware units such as branch predictors, issue logic, reorder buffers, etc., are needed to implement out-of-order execution, which in turn requires higher power and die area budgets. It has been reported that, with the same process technology, a new microprocessor design with performance improvement of 1.5x to 1.7x results in 2x to 3x increase in the die area [5] and 2x to 2.5x increase in the power consumption [6]. Thus, power efficiency has become the epicenter of all design efforts from an architectural standpoint as well.

While the CPU performance has been measured in terms of the execution throughput of a single thread, lately, an alternate metric, referred to as *throughput performance*, has been gaining more prominence. Throughput performance is defined as the number of threads that can complete execution per unit time by utilizing multiple CPU cores to perform more computations in parallel. A survey of commercially available multi-core processors can be found in [7]. As power dissipation continues to be an increasingly difficult challenge, there has been a shift in the paradigm in terms of CPU design. Instead of building a large and complex out-of-order processor, the designers are building multiple simple in-order processors within the same chip area. Each of those simple cores could further support the simultaneous execution of multiple threads resident within the core. Such multi-core systems are commercially available applied in high-end servers, gaming platforms, and embedded processors. Niagara [8] and Niagara2 [9] are multi-core general purpose microprocessors from Sun Microsystems used in high end servers that feature up to eight in-order cores. While each core in Niagara is capable of executing four threads, each core in Niagara 2 is capable of executing eight threads simultaneously. Intel's Larrabee architecture [10] for visual computing uses in-order CPU cores that support an extended version of the x86 instruction set. Each core supports execution of two hardware threads. The number of CPU cores is implementation-dependent. MIPS 1004K coherent processing system [11] is comprised of 1-4 multi-threaded cores, where each core is capable of executing two hardware threads simultaneously.

1.2 Power Consumption in Digital CMOS Circuits

Power consumption in digital CMOS circuits can be classfied into two major categories - *dynamic* power and *static* power [12] (Figure 1.1). While dynamic power is due to the activity in the circuit block and the switching frequency, static power is due to the fact that transistors are imperfect switches. The major component of dynamic power is contributed by the charging and discharging of the gate capacitances in the circuit during signal switching. The other component of dynamic power, short-circuit power, is a result of conducting paths between the voltage supply and ground for a brief period during which a logic gate makes a transition. During that period both the pull-down and the pull-up networks are ON. The primary component of static power is due to the subthreshold drain-source leakage, I_{subth}, which is dependent exponentially on the threshold voltage, V_T, As V_T is reduced linearly, I_{subth} increases exponentially. The other two components of static power are due to gate leakage currents and junction leakage currents. In contemporary CMOS technologies, subthreshold leakage is of paramount concern to circuit designers.

Traditionally, dynamic power has vastly dominated the overall power consumption of CMOS circuits. However, in sub-90 nm technologies, leakage power has emerged as a significant component of the total power consumed. This is because, in order to keep pace with the scaling trends in technology, the suppy voltage, V_{DD}, was lowered down to avoid excessive power density of the chip.

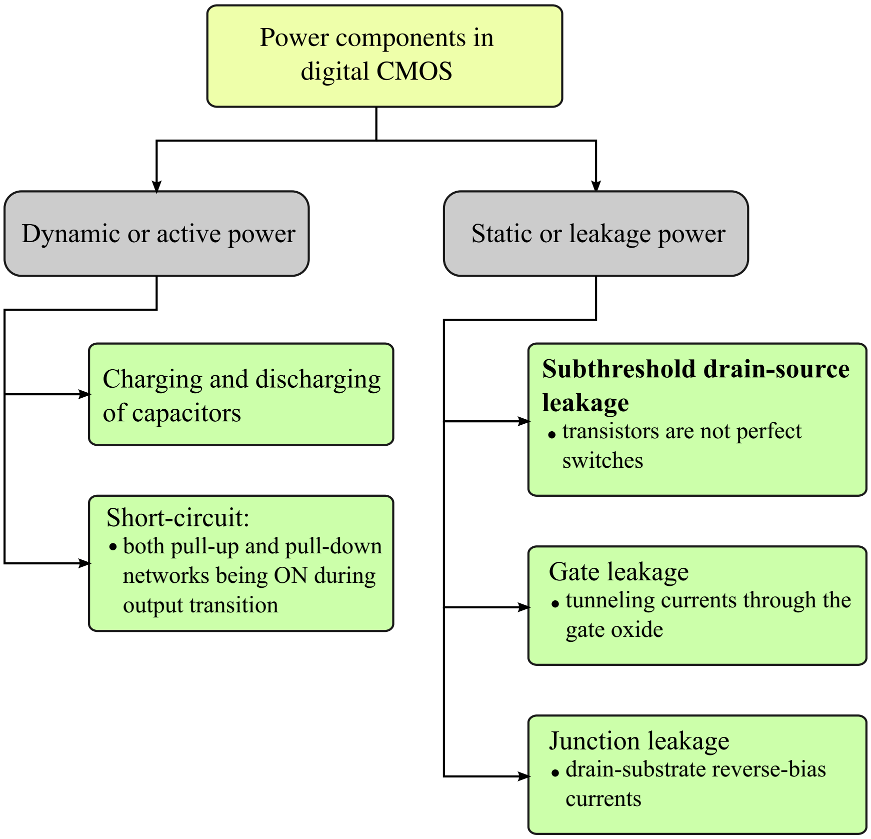

Power components in digital CMOS
Dynamic or active power
Static or leakage power
Charging and discharging of capacitors
Short-circuit:
• both pull-up and pull-down networks being ON during output transition
Subthreshold drain-source leakage
• transistors are not perfect switches
Gate leakage
• tunneling currents through the gate oxide
Junction leakage
• drain-substrate reverse-bias currents

Figure 1.1 Components of power consumption in digital CMOS circuits. Power consumed due to subthreshold leakage dominates the static power dissipation component in the contemporary technologies and is targeted in this dissertation.

This resulted in lowering the threshold voltage, V_T, in order to maintain the circuit performance. This caused the subthreshold leakage current, I_{subth}, to increase exponentially, thereby increasing the static power drastically. It has been pointed out in [13] that there has been a 3 - 5× increase in subthreshold and gate leakage currents per generation due to threshold voltage and gate-oxide scaling. Therefore, leakage power has become an important design aspect in low-power CMOS circuits. The main focus of this dissertation is the reduction of the subthreshold leakage component of static or leakage power in microprocessors.

1.3 Motivation for this Dissertation

The issues discussed above form the main motivation for the works reported in this dissertation. It has been reported that the leakage component of Intel's Xeon Tulsa processor [14] in 65 nm technology is about 30% of the total chip power. The core leakage is about 37%. The leakage components of Niagara [8] and Niagara2 [9] microprocessors by Sun Microsystems, in 90nm and 65 nm technologies, respectively, are between 25% and 30% of the total chip power. In sub-45 nm technologies, the leakage power component in microprocessors are projected to be at least 50% of their dynamic power component [15].

In view of these facts, it is important to investigate techiques to reduce leakage in microprocessors. In this dissertation, architectural techniques to reduce leakage energy in arithmetic functional units and register files, both of which are core components of a microprocessor, are proposed. While a compiler-directed framework with architectural support is presented for reducing leakage energy in functional units in embedded processors, purely microarchitectural techniques are proposed for reducing leakage in register files in general-purpose multithreaded processors.

1.4 Contributions of this Dissertation

The main contribution of this dissertation is a set of methodologies proposed at the architecture level to effectively use power gating, a circuit-level leakage reduction technique, for reducing leakage in embedded and general purpose microprocessors. The theme and the contributions of this dissertation are shown in Figure 1.2.

A brief description of the contributions are:

- *A Framework for Power Gating Functional Units in Embedded Microprocessors with Architecture and Compiler Support*: In this work, a framework is developed to power gate arithmetic functional units in embedded microprocessors with architecture and compiler support. The proposed framework includes an efficient algorithm for idle time estimation, appropriate insertion of sleep instructions within the code, and a method for reactivating the sleeping units that eliminates the need for having explicit wakeup instructions.

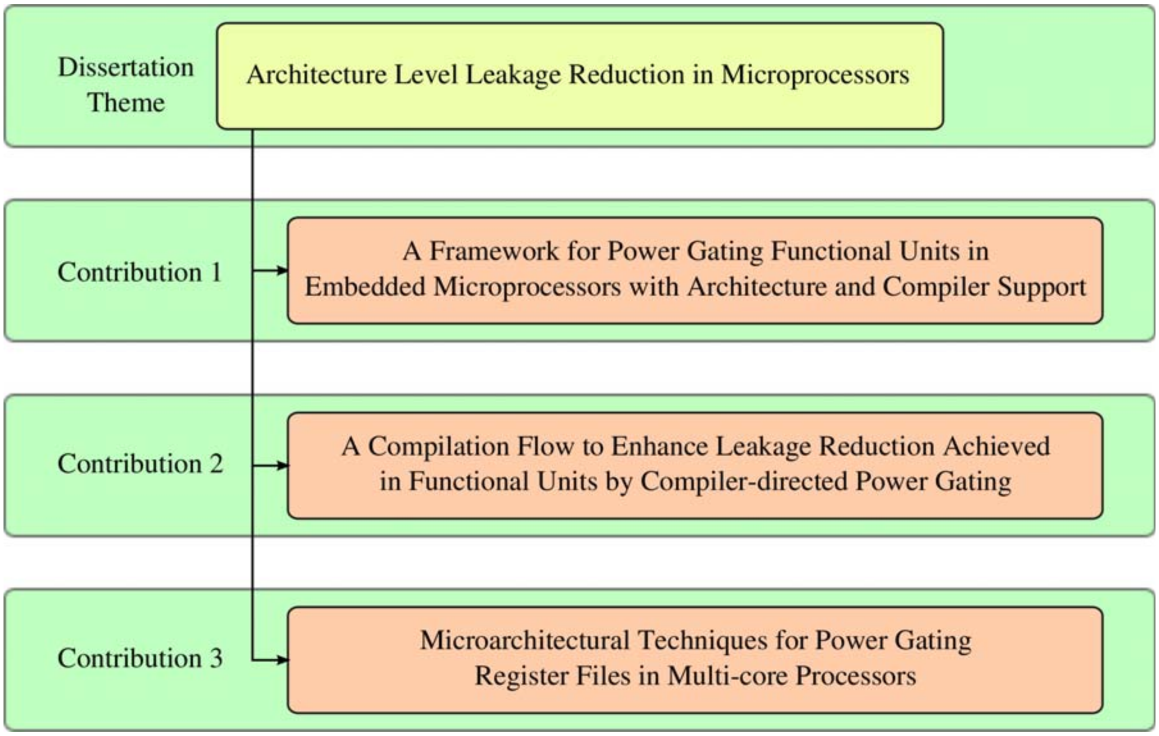

Dissertation Theme
Architecture Level Leakage Reduction in Microprocessors
Contribution 1
A Framework for Power Gating Functional Units in Embedded Microprocessors with Architecture and Compiler Support
Contribution 2
A Compilation Flow to Enhance Leakage Reduction Achieved in Functional Units by Compiler-directed Power Gating
Contribution 3
Microarchitectural Techniques for Power Gating Register Files in Multi-core Processors

Figure 1.2 Contributions of this dissertation. The theme of the disseration is architectural level leakage reduction in microprocessors. Contributions 1 and 2 are for embedded microprocessors, while contribution 3 is in the context of multi-core processors for general purpose applications.

- *A Compilation Flow to Enhance Leakage Reduction Achieved in Functional Units by Compiler-directed Power Gating*: In the context of the compiler-directed power gating framework discussed above, the impact of traditional performance-enhancing compiler optimizations on the amount of leakage savings obtained is studied through analysis and simulations. Based on the observations made, a leakage-aware compilation flow is derived that improves the effectiveness of the framework.

- *Microarchitectural Techniques for Power Gating Register Files in Multi-core Processors*: A class of microarchitectural techniques for fine-grained state-retentive power gating in integer register files to save leakage energy is proposed for multi-core processors featuring in-order cores that support hardware multithreading. In state-retentive power gating, the registers retain their states through the power gating period. In an in-order core, when a thread gets blocked due to a memory stall, the corresponding register file can be placed in a low leakage state through power gating for leakage reduction. When the memory stall gets resolved, the register file is activated for subsequent accesses. While state-retentive power gating in single cores has been studied in the literature, it is being investigated for multi-core architectures for the first time in this work.

1.5 Outline of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the background and a comprehensive literature survey related to the specific problems being addressed in this dissertation. More specifically, first a short tutorial on power gating, the circuit-level leakage reduction technique that forms the underlying technique in all the micro-architecture support modeled in all the works. Following that, a short tutorial on computer architecture is presented which describes the organization of a typical pipelined microprocessor. In Chapter 3, a framework for power gating arithmetic functional units in embedded microprocessors with architecture and compiler support is presented. This framework comprises of two components - a hardware component and a software component. The hardware component includes a library of arithmetic functional units redesigned with sleep transistors and a microarchitectural model of the embedded processor along with ISA support for controlling the sleep states of these units. The software component comprises of extensions made to a compiler infrastructure, which include an efficient algorithm for identifying program regions where functional units are idle so that sleep instructions may be inserted into the code to put those units to sleep during program execution. In Chapter 4, the impact of traditional performance-enhancing compiler optimizations on the amount of leakage savings obtained with the framework described in Chapter 3 is studied through analysis and simulations. Based on the observations, a leakage-aware compilation flow is derived that improves the effectiveness of this framework. In Chapter 5, purely microarchitectural techniques are investigated to perform state-retentive power gating of register files in multi-core processors supporting hardware multithreading. The techniques proposed are based on the fact that in an in-order core, when a thread gets blocked due to a pipeline stall, the corresponding register file can be placed in a low leakage state. When the stall gets resolved, the register file is activated so that it may be accessed again. Finally, some concluding remarks and the challenges going forward are discussed in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, some fundamental concepts that form the basis of the research work presented in this dissertation are described. More specifically, we discuss the basic design techniques for reducing subthreshold leakage in CMOS circuits and describe *power gating*, a circuit-level technique used for reducing subthreshold leakage, in detail. Followed by that, we present the basics of pipelined microprocessors and discuss the support for enabling hardware multithreading in such processors. A detailed description of the various related works for leakage power reduction in microprocessors applied at the architecture and microarchitecture level is also presented in this chapter.

2.1 Subthreshold Leakage Reduction Techniques

Subthreshold leakage in CMOS circuits is due to the inability of transistors to act as perfect on/off switches. Therefore, leakage reduction can be done by [12]:

- Increasing the resistance in the leakage path
- Reducing the voltage over the leakage path

All the fundamental leakage reduction techniques fall under one of the above two categories. This is shown in Figure 2.1. Most of the techniques fall in the former category because the latter is hard to achieve. The ability to change the supply voltage to a circuit either requires a variable voltage supply mechanism or voltage bias circuits that could enable multiple supply voltages. Since these are hard to achieve in complex designs because of a variety of design issues [12], most of the state-of-the-art circuit level leakage reduction techniques used by designers fall under the former category. These techniques [12, 16] are briefly described next:

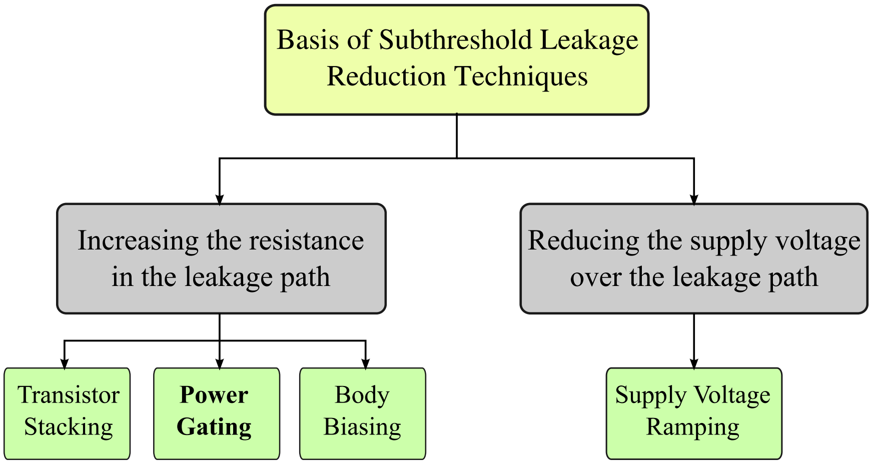

Basis of Subthreshold Leakage Reduction Techniques
Increasing the resistance in the leakage path
Reducing the supply voltage over the leakage path
Transistor Stacking
Power Gating
Body Biasing
Supply Voltage Ramping

Figure 2.1 Subthreshold leakage reduction approaches. In this dissertation, power gating is the underlying circuit level leakage reduction technique used in all the architecture level solutions proposed to reduce leakage in microprocessors.

- *Transistor Stacking*: Stacking of transistors has a super-linear impact on leakage reduction due to drain-induced barrier lowering (DIBL). DIBL is a deep-submicron effect and is related to a reduction of the threshold voltage as a function of the drain voltage. For a given circuit block, the ideal way to impose the maximum transistor stacking would be to control the inputs of each input gate independently. However, since this is not possible and the only inputs controllable are the primary inputs to the circuit block, the next best thing is to find the primary input pattern to this block that minimizes leakage during standby mode [17–19]. Although the leakage savings achieved by this technique is very limited, the advantage of this technique is that it has negligible impact on performance and its implementation is very simple.

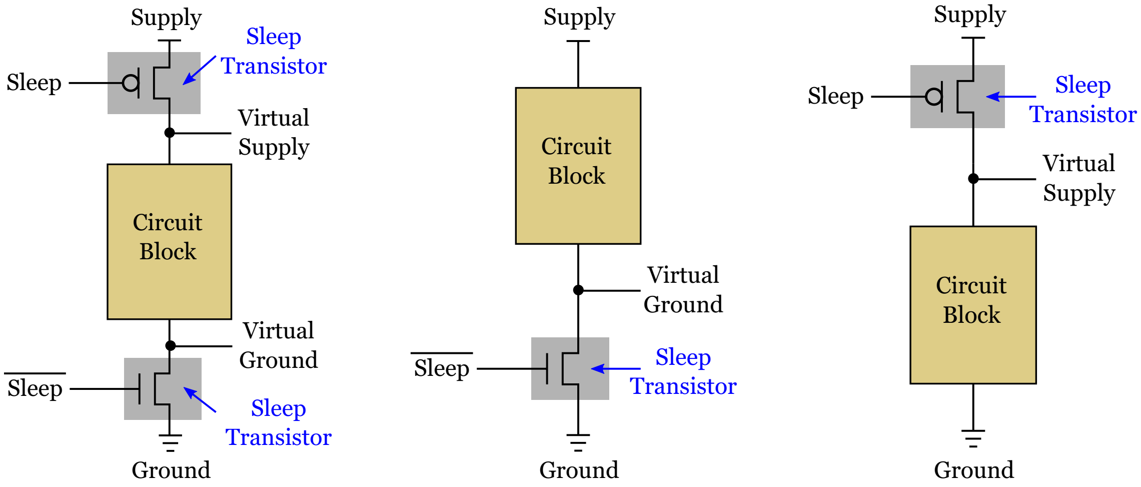
- *Power Gating*: The ideal way to eliminate subthreshold leakage is to disconnect the circuit block from the supply rails. But that would require the availability of perfect on-off switches. Since such switches do not exist in the contemporary CMOS technologies, switches are used as *large resistors* between the virtual supply rails of the circuit block and the global supply rails. Up to three orders of magnitude of leakage reduction can be achieved by this technique. Since power gating is the underlying circuit level leakage reduction technique in all the architectural leakage reduction solutions proposed in this dissertation, we discuss power gating in more details in Section 2.2.

- *Body Biasing*: An alternative approach to power gating is to decrease the leakage current by increasing the threshold voltage of the transistors in the circuit block. Each transistor has a fourth terminal, which can be used to increase the threshold voltage by reverse biasing. Since the subthreshold leakage current depends exponentially on the threshold voltage, this technique can be very effective in minimizing leakage in circuits. Moreover, this technique does not come with a performance penalty and it does not even change the circuit topology. Although this technique looks very attractive due to these attributes, few of its drawbacks are very critical in exploring this technique in real designs. First of all, the leakage reduction achieved by this technique is much lower (by up to two orders of magnitude) than that achieved by power gating. Secondly, it requires more sophisticated triple-well technology if both NMOS and PMOS transistors need to be controlled [12].

2.2 Power Gating

Figure 2.2 shows the schematic view of the power gating technique and the various options available to implement it. If the power gating device, also known as the *sleep transistor,* is inserted between the V_{DD} and the pull-up network of the circuit block, it is called a *header* device. On the other hand, if it is inserted between the ground rail and the pull-down network of the circuit, it is called a *footer* device. Leakage current in the circuit block reduces because of the following two reasons:

- *Increased resistance in the leakage path*: The header and the footer sleep devices act as large resistors to the leakage path during standby mode.
- *Source biasing introduced by stacking effect*: The additional devices in series with the circuit pull-up and pull-down networks introduces stacking effect, which increases the threshold of the transistors in stack.



(a) Footer + header (b) Footer only (c) Header only

Figure 2.2 Power gating options. Since NMOS transistors are more area-efficient than PMOS transistors, the footer only option (b) has the minimum area overhead amongst all the three options. The footer + header option (a) suffers from the maximum area overhead but is the most effective in achieving leakage reduction.

In Figure 2.2, three power gating options are shown:

- *Footer and header*: In this configuration (Figure 2.2(a)), both the header and the footer devices are used and are simultaneously turned off when the circuit block is idle. This configuration has a maximum area penalty because of the two sleeper devices but achieves the maximum leakage reduction by ensuring that the stacking effect is enforced independent of the input patterns to the circuit block.

- *Footer only*: In principle, it is sufficient to use only a single power gating device to achieve leakage reduction. In the footer-only configuration (Figure 2.2(b)), only a footer device (an NMOS transistor) is used. The area overhead is the minimum in this case.

- *Header only*: In this configuration, only a header device (a PMOS transistor) is used.

Most often, when a single power gating device is selected, a NMOS sleep transistor is preferred over the PMOS one because a NMOS transistor's ON-resistance is smaller than that of a PMOS transistor for the same transistor width. Since subthreshold current, I_{subth}, varies exponentially with respect to the threshold voltage, V_T, the savings achieved by the power gating technique are maximized when the technology supports both high V_T and low V_T transistors. While the latter can be used for logic to achieve lower delays, the former act as very effective power gating devices. When power gating is implemented using multiple threshold devices, it is often called Multiple Threshold CMOS (MTCMOS) technology [20–22].

2.2.1 Performance Aspects of Power Gating

When the circuit block is idle, the sleep transistor (in footer configuration) is placed in the cut-off mode, thereby introducing a large resistance in the standby leakage path between the supply and the ground. The circuit is referred to be in the *sleep* or *inactive* state. During this state, the virtual ground charges up to a steady state value that is determined by the resistive divider formed by the other transistors in the stack. To bring the circuit back to the *active* state, the virtual ground is restored to its nominal value by placing the sleep transistor in the saturation mode. Since this requires discharging the virtual ground node to actual ground, there is a *wake-up latency* associated with it. Moreover, since the deactivation and activation of the circuit block involves discharging and charging the output capacitances of the internal circuit nodes, it restricts how often the circuit block can be transitioned between the two states to achieve overall energy savings. The period of time that the circuit block should be kept in sleep state before bringing it back to the active state so that the leakage energy savings equals the dynamic power overhead incurred circuit activation is known as the *breakeven period* [23].

2.3 Pipelined Microprocessors

Pipelining is a very effective implementation technique for improving system throughput without requiring massive replication of hardware. It was first employed in the design of high-end mainframes in the 1960s - IBM 7030 [24] and CDC 6600 [25]. All modern microprocessors employ pipelining almost universally. In this section, we present a short tutorial on concepts related to pipelined microprocessors.

2.3.1 Pipelining Basics

In the context of instruction set processors, pipelining involves partitioning the processor design into multiple stages such that each stage performs only a part of the computation required by each instruction. A typical instruction cycle can be functionally partitioned into the following five generic computations [26]:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Operand(s) fetch (OF)
4. Instruction execution (EX)
5. Operand store (OS)

A typical instruction cycle starts with the fetching (IF) of a new instruction from the memory to be executed in the processor core. Following this, it is decoded (ID) so that *work* to be performed by the instruction may be determined. Depending on the type of the instruction, one or more operands may be fetched (OF) from the register file or the memory. Once the necessary operands are available, the instruction is executed (EX) in the appropriate functional units of the processor. Finally, the results generated by the computation in the EX stage are stored back (OS) to the register file or the memory. Since memory access latencies are multiple orders of magnitude slower than the latencies involved in the tasks done within the processor core, caches are universally employed between the processor datapath and the memory to speed up memory accesses. A cache that stores instructions is called an instruction cache (I-cache), while one that stores data is called a data cache (D-cache). Figure 2.3 shows the GENERIC (GNR) pipeline [26].

2.3.2 Classification of Instruction Types

Computations performed by instructions in a typical computer may be categorized into three generic tasks - (i) *arithmetic operation*, (ii) *data movement*, and (iii) *instruction sequencing*. Based on these tasks, in a typical modern processor architecture, instructions are classified into three types:

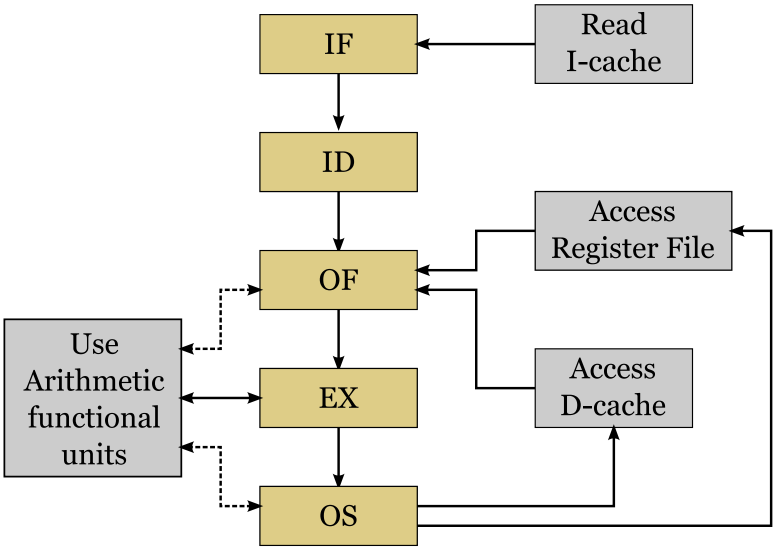


Figure 2.3 The GENERIC (GNR) pipeline. It has the five generic stages - instruction fetch (IF), instruction decode (ID), operand fetch (OF), instruction execute (EX), and operand store (OS). It also shows the the components that are typically accessed at each of these stages. This view of the pipeline is shown because, in this dissertation, architectural techniques are proposed to reduce leakage power in the arithmetic functional units and the register file when they are idle during program execution. It should be noted that this figure does not depict the physical organization of a pipelined processor core.

1. *ALU instructions*: These instructions perform arithmetic and logical operations.

2. *Memory or load/store instructions*: These instructions are used to move data between registers and the memory.

3. *Branch instructions*: These instructions control instruction sequencing based on the control flow of the program.

The semantics of each of the three instruction types can be specified based on the sequence of subcomputations performed by that instruction type. The semantics of all the instruction types enumerated above are defined in Tables 2.1, 2.2, 2.3.

Table 2.1 Specification of ALU instruction type

Generic Subcomputation	**Integer**	**Floating Point**
IF	Fetch instruction (access I-cache)	Fetch instruction (access I-cache)
ID	Decode instruction	Decode instruction
OF	Access integer register file	Access floating point register file
EX	Perform integer ALU operation	Perform float point operation
OS	Write back to integer register file	Write back to floating point register file

Table 2.2 Specification of memory or load/store instruction type

Generic Subcomputation	**Load instruction**	**Store Instruction**
IF	Fetch instruction (access I-cache)	Fetch instruction (access I-cache)
ID	Decode instruction	Decode instruction
OF	Access integer register file for base address Generate effective address (base + offset) Read from memory (access D-cache)	Access integer register file for the base address and the integer or floating point register file for the register operand
EX		
OS	Write back to register file	Generate effective address (base + offset) Write into memory (access D-cache)

2.3.3 In-Order Processors

Pipelined processor designs are classfied into two categories - *scalar* and *superscalar* - according to the number of instructions that can be processed in each of the pipeline stages. A scalar pipeline is one which at most one instruction may be processed at a time. A superscalar pipeline, on

Table 2.3 Specification of branch instruction type

Generic Subcomputation	Unconditional branch	Conditional branch
IF	Fetch instruction (access I-cache)	Fetch instruction (access I-cache)
ID	Decode instruction	Decode instruction
OF	Access integer register file for the base address Generate effective address (base + offset) Read from memory (access D-cache)	Access integer register file for the base address Generate effective address (base + offset)
EX		Evaluate branch condition
OS	Update program counter (PC) with target address	If the branch condition evaluates to true, update program counter (PC) with target address

the other hand, may process more than one instructions at the same time. Figure 2.4 shows examples of a scalar and a superscalar pipeline. Pipelined processor designs are classfied into two categories

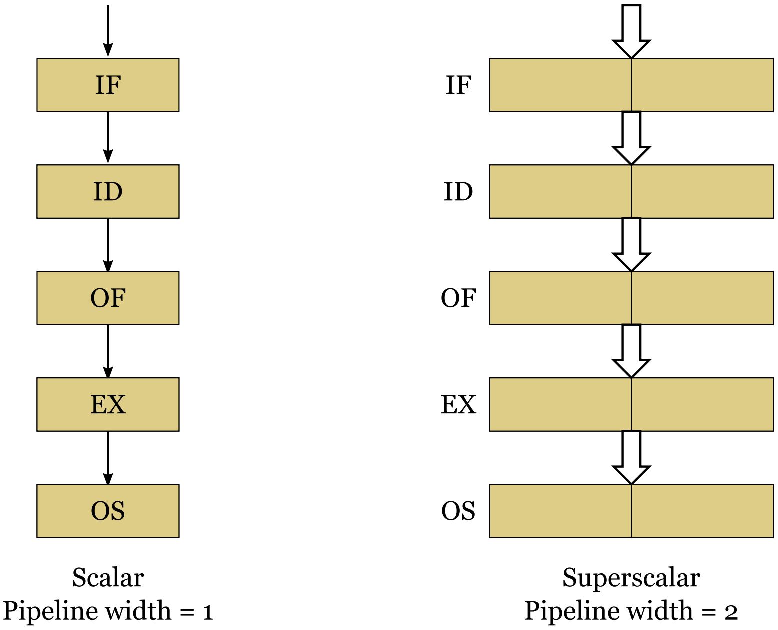


Figure 2.4 Scalar vs. superscalar pipeline.

- *in-order* and *out-order* - Processors are further classified according to how the instructions from a program are sequenced within its pipeline as *in-order* and *out-order* processors. In an in-order processor, instructions enter the pipeline, advance synchronously through all the pipeline stages, and finish in program order. Since, this imposes a lockstep fashion in the way the instructions advance through the pipeline, such pipelines are also called *lockstep* or *rigid* pipelines. The drawback of such pipelines is that when one instruction gets stalled, all the following instructions also get stalled which puts severe restrictions on the instruction throughput. In contrast to this, an out-order processor supports bypassing of a stalled leading instruction by trailing instructions, thereby allowing *out-of-order execution* of instructions. Due to this reason, the hardware complexity of out-order processors is significantly more than in-order processors. Most high-performance processors, where performance has traditionally been of foremost importance, are all out-order processors, while most embedded processors, where power efficiency is of paramount importance, are in-order processors. In this dissertation, all the processor models considered are in-order processors.

2.4 Hardware Multithreading

In this section, we present an overview of the various hardware multithreading approaches that have become increasingly prevalent in modern high performance microprocessors. In the previous section, the discussion of pipelined processors was restricted to single-threaded support. In Chapter 5, microarchitectural techniques to reduce leakage in register files are proposed in the context of hardware multithreaded multi-core processors.

Hardware multithreading is an approach which enables a processor to support the simultaneous execution of multiple threads. A processor that supports hardware multithreading is called a *multithreaded processor*. Multithreading approaches are categorized according to how they are implemented in hardware [26, 27] and are briefly described here:

- *Coarse-Grained Multithreading (CGMT)*: In this approach (Figure 2.5(a)), a thread uses all CPU resources until a long latency event, like a cache miss, a long latency operation, etc., occurs. Such an event causes a context switch and another ready thread is switched in, which runs till it encounters a long latency event. This implementation has a context switch latency

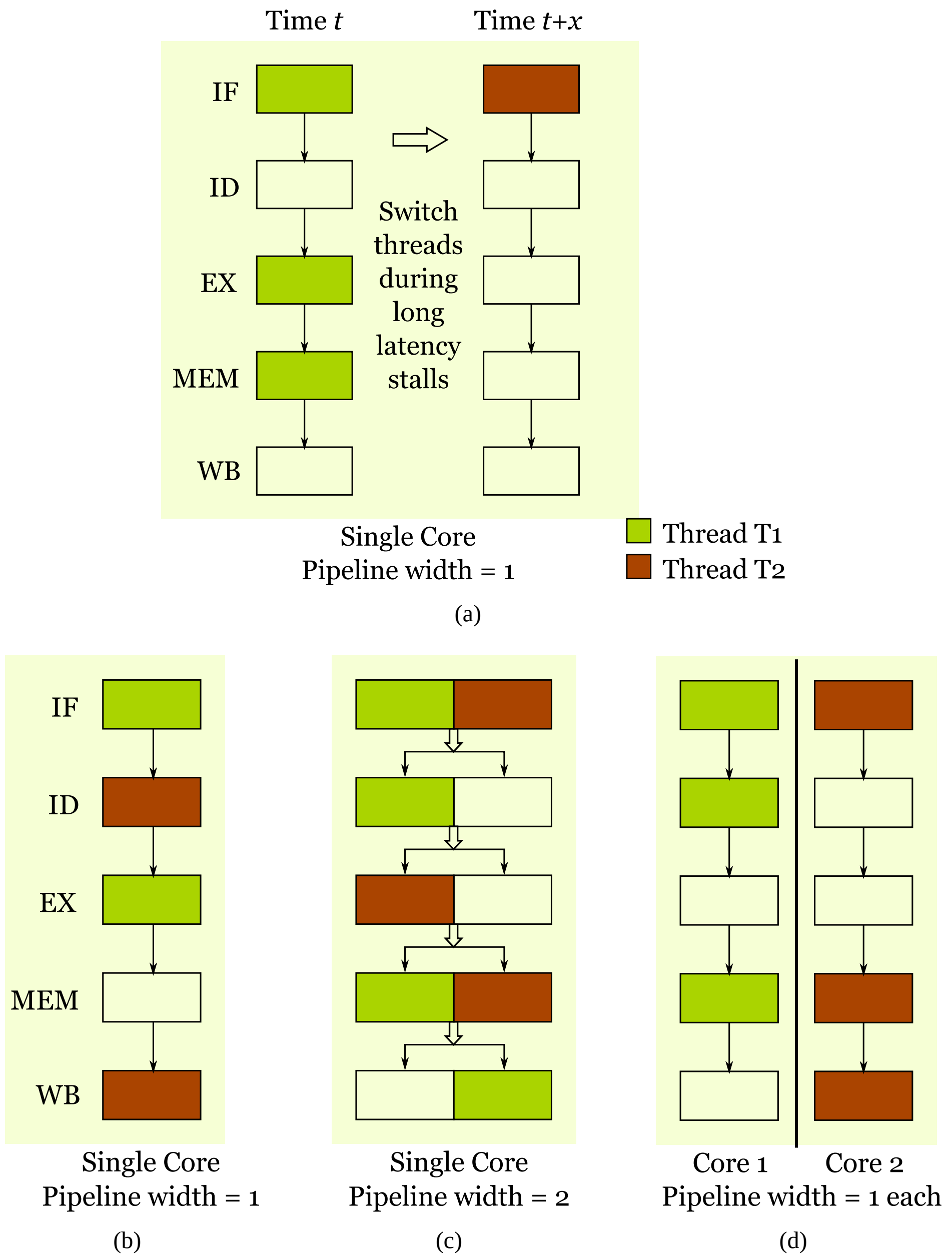


Figure 2.5 Multithreading approaches. (a) coarse-grained multithreading (CGMT) with 2 threads and pipeline width of 1; (b) fine-grained multithreading (FGMT) with 2 threads and pipeline width of 1; (c) simultaneous multithreading (SMT) with 2 threads and pipeline width of 2; (d) chip multiprocessing (CMP) with 2 threads and pipeline width of 1 per core. A 5-stage MIPS pipeline model is shown.

associated with it. This is because, depending on the pipeline stage where the long latency event is detected (e.g., I-cache miss happens in IF-stage but D-cache miss happens in MEM-stage in a MIPS pipeline [26]), the instructions in the preceding stages are squashed, while the instructions in the succeeding stages are allowed to finish before the next thread can be run. Each thread context has a private copy of the register file, instruction fetch buffers, if any, and control logic state, while the rest of the CPU resources are shared. This approach is also known as *blocked multithreading* technique.

- *Fine-grained multithreading (FGMT)*: In this category (Figure 2.5(b)), thread context switching happens at the boundary of one of more clock cycles for ready threads (i.e., threads that are not blocked due to long latency events). Each thread context has a private copy of the register file and control logic state, while the rest of the CPU resources are shared. Instruction fetch-buffers may or may not be shared. FGMT is also known as *interleaved multithreading*.

- *Simultaneous multithreading (SMT)*: In SMT (Figure 2.5(c)), instructions from two or more threads are scheduled simultaneously on different functional units during the same cycle. SMT typically works on superscalar processors that have hardware to support simultaneous execution for two or more instructions in a single cycle. Each thread context has a private copy of the register file, instruction fetch buffers, interstage buffers, and control logic state. The rest of the resources are shared.

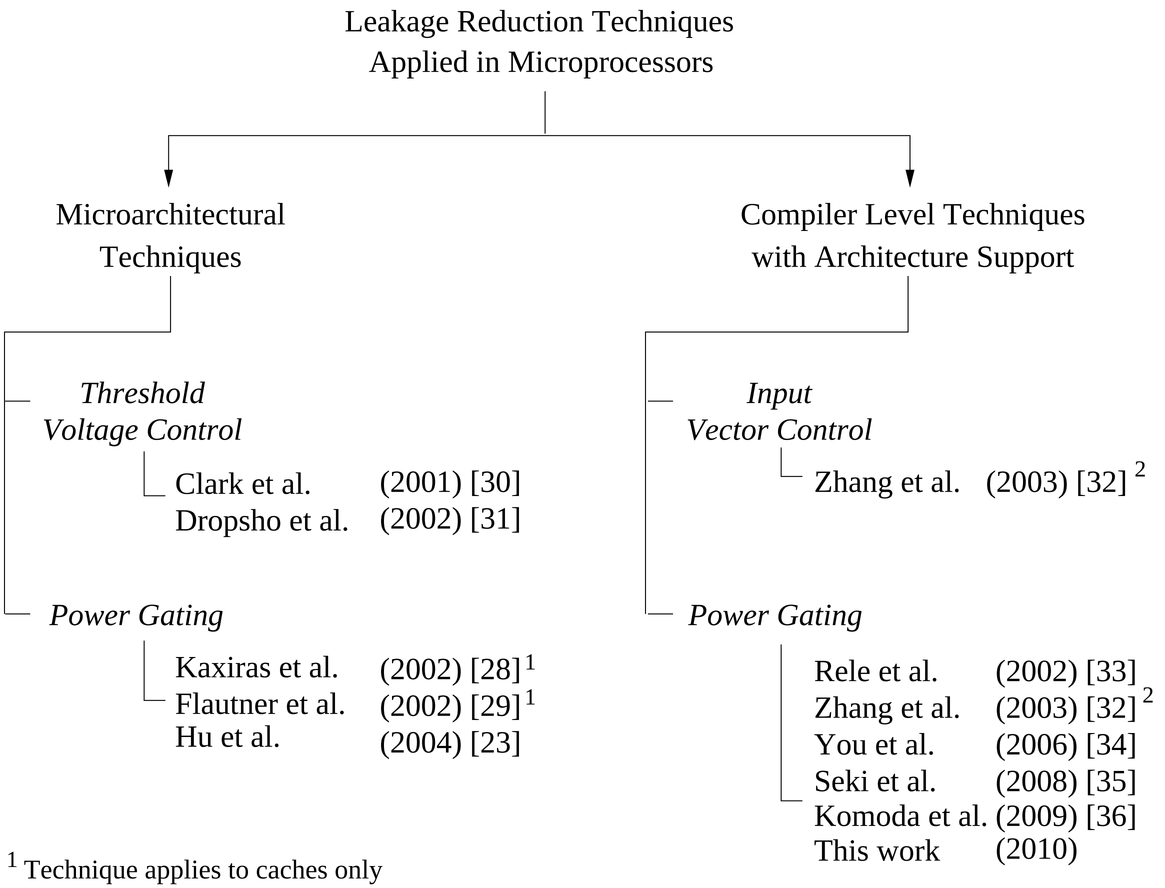
- *Chip multiprocessing (CMP)*: In CMP (Figure 2.5(d)), multiple single-threaded processor cores are instantiated on a die such that they share only the L2 cache and the system interfaces. Each core executes instructions from a different thread independently of the other threads, interacting only through shared memory.

2.5 Related Work

In microprocessors, the methods at the architecture level utilize circuit-level leakage reduction techniques discussed earlier. In such approaches, the microarchitectural subsystems are equipped with multiple leakage reduction techniques that enable putting those subsystems in and out of low-leakage mode. However, the design of the interface of the controls to those techniques classifies these approaches into two distinct categories. They are microarchitectural approaches and compiler-directed architectural approaches. In a microarchitectural approach, the logic to regulate those controls is implemented in hardware. In a compiler-directed approach, the interface of such controls is included in the instruction set in the form of special instructions or hardware directives, which the compiler inserts into the code. During the program execution, these instructions regulate the leakage in the idle subsystems of the processor.

Earlier works [28, 29] on architectural level leakage reduction have concentrated primarily on the memory subsystems (particularly on caches), since they contribute upto 50% of the total leakage of the system. The various works on leakage reduction in microprocessors (Figure 2.6) are classified as (i) microarchitecture level techniques, and (ii) compiler level techniques with architectural support. A circuit level technique based on reverse body-bias was used for leakage reduction in the commercial Intel Xscale microprocessor [30]. An analytical energy model to achieve leakage energy optimization in functional units for superscalar processors is described in [31]. The static energy in the integer functional units is reduced by employing dual threshold voltage domino logic design technique. Power gating of execution units is investigated by Hu et. al [23] in which the activation and deactivation of the functional units are guided by branch prediction decisions. The techniques discussed in [23, 31] use specifically designed control blocks that monitor the sleep periods of functional units. The control blocks involve significant dynamic power overhead and need to be avoided in the design of embedded processors.

Several techniques have been explored at the compiler level to identify opportunities for power gating. In [32], Zhang et. al. investigate the use of input vector control methods and dynamic profiling for leakage reduction. The approach in [33] uses static code analysis to identify power gating opportunities in the program and uses dynamic profiling information to direct the insertion of power gating instructions into the code. The method does not consider power gating opportunities in nested loop structures and requires special architectural support to remove power gating instructions inserted too close to each other which can cause significant performance degradation. In [34], data-flow analysis is used to estimate functional unit requirements in the basic blocks of the program and identify opportunities to insert sleep and wakeup instructions at compiler level. The use of data flow analysis instead of dynamic profiling results in failure to accurately estimate the resource


Leakage Reduction Techniques Applied in Microprocessors
Microarchitectural Techniques
Compiler Level Techniques with Architecture Support
Threshold Voltage Control
Clark et al. (2001) [30]
Dropsho et al. (2002) [31]
Power Gating
Kaxiras et al. (2002) [28] 1
Flautner et al. (2002) [29] 1
Hu et al. (2004) [23]
Input Vector Control
Zhang et al. (2003) [32] 2
Power Gating
Rele et al. (2002) [33]
Zhang et al. (2003) [32] 2
You et al. (2006) [34]
Seki et al. (2008) [35]
Komoda et al. (2009) [36]
This work (2010)

[1] Technique applies to caches only

[2] Investigates both input vector control and power gating

Figure 2.6 Taxonomy of works on leakage reduction in microprocessors. (the works, unless stated otherwise, target functional units)

requirements in both single-level and nested loops in the program. Dynamic profiling provides the runtime characteristics of a program, which can be effectively used in identifying power gating opportunities. Further, the framework in [34] assumes that every branch in the program is equally likely to be taken (or not taken). This assumption is too conservative, since it is well known that the behavior of branch instructions is predictable in a runtime environment, even more so for the branches that represent loop exit conditions. In [35], Seki et al. presented a fine-grained power gating scheme for the MIPS R3000 which was incorporated in a prototype chip, Geyser-0, in 90 nm CMOS technology. Komoda et al. [36] proposed a technique similar to that in [34] but with interprocedural analysis. Roy et al. proposed a framework in [37] that uses both static code analysis and dynamic profiling to identify potential subgraphs in the program during which the units can be kept deactivated. More recently, power gating has also been used as a primary power management technique in modern commercial processors [38, 39].

Although there is significant work reported in the literature on leakage power reduction in functional units, very few works in the literature have tried to address leakage reduction in register files. A multi-banked register file design to improve access speed and reduce total power is presented in [40], while low-leakage register files with dynamic controls have been proposed in [41, 42]. A state-retentive register file designed for the ARM processor was fabricated using 65-nm [43] technology just to study the leakage aspects of a register file in general.

2.5.1 Context and Significance of this Dissertation

The earlier works in leakage reduction in the functional units in microprocessors are primarily focused on superscalar processors. In this work, the existing body of work in this area is supplemented by by a novel framework for compiler-directed power gating, which is particularly geared towards embedded systems. Further, we also investigate the impact of compiler optimizations on the opportunities for power gating and derive a leakage-aware compilation flow, which helps generate code to achieve maximal leakage energy reduction during execution. Finally, while the related works described in the earlier section on leakage reduction techniques applied to register files represent important contributions in this field, its application in the context of multi-core processors has not been addressed in any other work prior to the research effort described in this dissertation.

CHAPTER 3

A FRAMEWORK FOR POWER GATING FUNCTIONAL UNITS IN EMBEDDED MICROPROCESSORS

In this chapter, we present a new framework for power gating the functional units in embedded system microprocessors without degradation in performance. The proposed framework includes an efficient algorithm for idle time estimation, appropriate insertion of sleep instructions within the code, and a method for reactivating the sleeping units only when needed *without the use of wakeup instructions*. We introduce the notion of *loop hierarchy trees* (LHT) to represent the partial ordering of the nested loops within the program. From the control flow graph (CFG) representation of the source program, a forest of loop hierarchy trees is constructed and is used to identify the maximal sub-graphs representing the long idle periods for the functional units. For each sub-graph thus identified, a sleep instruction is introduced in the program with a list of corresponding functional units to be deactivated. When an instruction is decoded, the functional units needed for that instruction are automatically activated by the control unit such that the units are ready before the instruction reaches the execute stage. This eliminates the need for wakeup instructions to be inserted into the object code reducing the overheads. In our implementation, the ARM processor architecture was modified and resynthesized to include power gating by developing a CMOS cell library of functional units with the above capabilities. Experimental results are reported for a set of 12 benchmarks chosen from the MiBench suite, which indicate that, on average, our technique reduces the leakage energy in functional units by 31.1% for integer benchmarks and 26.8% for floating point benchmarks.

3.1 Power Gating in Microprocessors

In microprocessors, power gating is used to reduce leakage in parts of the processor which are not required for sustained periods of time during program execution. However, as discussed earlier, the implementation of power gating using sleep transistors, incurs some latency overhead, called *activation* latency, while activating the circuit. This is the time that the circuit takes to become electrically stable after the power supply has been restored to the circuit. Moreover, the activation and deactivation of the circuit results in dynamic power overhead. The minimum period for which the circuit should remain turned off such that the savings in leakage energy equals the dynamic energy overhead is called the *breakeven* period. The activation latency and the breakeven period are important factors to be taken into account in the implementation of power gating.

In this work, we investigate the above issues in detail and propose a framework for leakage reduction in the functional units of the datapaths in embedded microprocessor cores. The framework uses both static code analysis and dynamic profiling to extract program characteristics useful in determining power gating opportunities for leakage reduction. The identification and analysis of loop hierarchies within code segments is an important objective in the framework. Thus, we focus on the iterative code structures in the programs for detection of long idle regions for functional units. The switches for power gating the functional units are provided at the circuit level, which are controlled with special instructions inserted within the code during compile time based on idle time behavior analysis. The salient features of the proposed framework are:

1. The approach uses both static code analysis and dynamic profiling information to identify power gating opportunities in the program, as well as, to direct the insertion of power gating instructions into the code.

2. The *breakeven* period and the *activation latency* are used in determining power gating opportunities within the code segments.

3. We introduce the notion of *loop hierarchy trees* (LHT) to represent the partial ordering of the nested loops within the program. From the control flow graph (CFG) representation of the source program, a forest of loop hierarchy trees (LHT) is constructed capturing the partial

ordering of the nested loops within the program, which is then used to identify the maximal sub-graphs representing the long idle periods for the functional units. For each sub-graph thus identified, a sleep instruction is introduced in the program with a list of corresponding functional units to be deactivated.

4. The proposed technique inserts only *sleep* instructions into the code, but does not insert any *wakeup* instructions. This is achieved by limiting the *activation latency* of the functional units to a single clock cycle and by extending the decode unit so that, whenever an instruction is decoded, the required functional units are activated just before the instruction reaches the execute stage. This saves significant instruction overhead in implementing power gating.

A cell library consisting of a set of new cells with power gating capability was designed and verified using 70nm CMOS technology. Extensive simulations were performed using the ARM processor as the target architecture model based on the above cell library characterization. Experimental results on the MiBench embedded benchmark suite [44] indicate that significant leakage reduction is possible using the proposed approach.

3.2 Proposed Framework for Leakage Reduction Using Power-Gating

The proposed framework for leakage reduction shown in Figure 3.1 consists of two major components. The hardware component consists of (i) a library of functional units with power gating capability, and, (ii) an embedded processor architecture which supports power gating for functional units. The software component consists of extensions at the compiler level to (i) perform source code analysis, (ii) identify idle times for various functional units, and, (iii) insert sleep instructions for those functional units at appropriate locations in the source code. The application source programs are profiled dynamically for run-time characterization and this information is used to accurately predict long idle times for the functional units so that they can be turned off to save leakage. Finally, in order to study the impact of power gating on microprocessor performance, a cycle-accurate simulation of the proposed power gating methodology was performed and a number of application programs were used for testing.

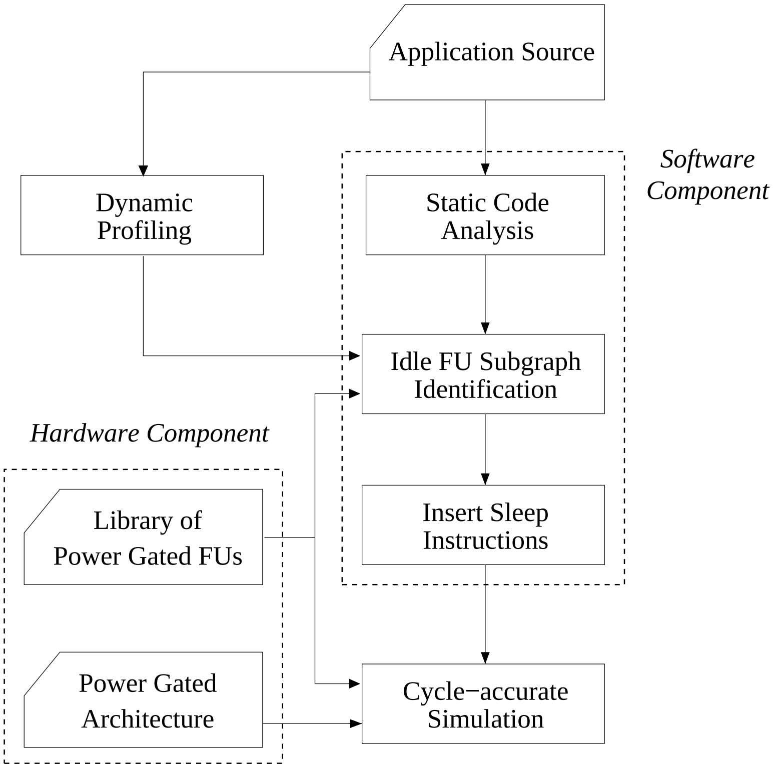


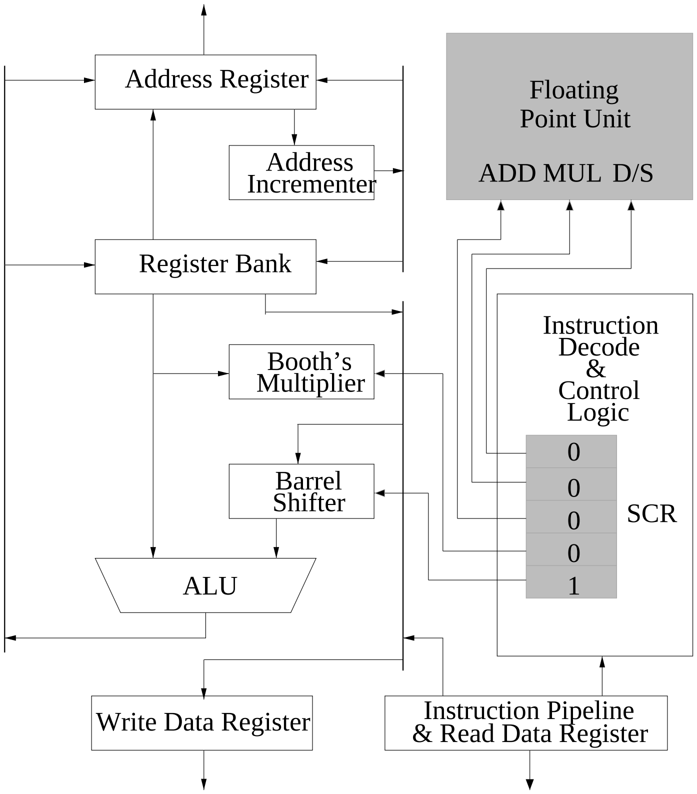
Figure 3.1 Proposed framework for power gating

3.2.1 Hardware Component

The hardware component consists of a library of functional units with sleep transistors and an architecture with control capabilities for those sleep transistors. The architecture and the design of functional units are described here.

3.2.1.1 Embedded Processor Architecture with Power-Gating Support

The target embedded processor architecture, as shown in Figure 3.2, is based on the popular ARMv7 processor core (`www.arm.com`). The ARMv7 processor core has an integer ALU, a barrel shifter, and an integer multiplier. Since the processor does not have a floating point unit (FPU), floating point (FP) operations are emulated using software macros constructed out of integer instructions. Since a significant fraction of embedded systems applications (multimedia applications) operate on FP numbers, a FPU is included in the target architecture. The FPU shown in Figure 3.2, consists of an adder (ADD), multiplier (MUL), and, a division and square root unit (D/S). The func-

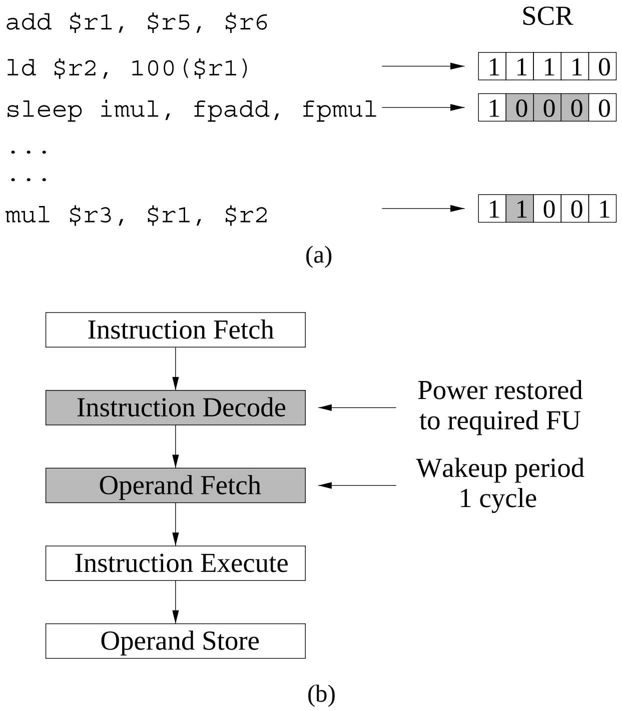


ADD − Adder
MUL − Multiplier
D/S − Division and Square Root
SCR − Sleep Control Register

Figure 3.2 Modified ARM architecture

tional units are designed such that the activation latency is limited to one clock cycle. The details of the integer and FP functional units and their power gated versions are given in Section 3.2.1.2.

A Sleep Control Register (SCR) is added to the instruction decode logic which drives the sleep controls of the various functional units, as shown in in Figure 3.2. The activation and deactivation of the functional units is illustrated in Figure 3.3. The deactivation is carried out using a `sleep` instruction. The functional units that need to be deactivated are specified as operands to the `sleep` instruction, as determined during program analysis. For each functional unit in the `sleep` instruction, a '0' is written at the corresponding cell in the SCR. Figure 3.3(a) shows a sequence of instructions. The contents of the SCR after each instruction is decoded is shown to its right. Before the `sleep` instruction is decoded, the contents of the SCR are "11110" indicating that only the FP division and square root unit is in power gated mode. The operands passed to the `sleep` instruction are - integer multiplier, FP adder, and FP multiplier. When this instruction is decoded, the cells driving the sleep controls for each of these functional units are written with a ‘0’ (indicated by the gray colored cells next to the `sleep` instruction in the figure), initiating their deactivation process.



```
add $r1, $r5, $r6
ld $r2, 100($r1)
sleep imul, fpadd, fpmul
...
...
mul $r3, $r1, $r2
```

(a)

(b)

Figure 3.3 Example of activation and deactivation of functional units

The activation of the functional units takes place at the decode stage. The functional unit gets activated during the cycle following the one in which the instruction enters the operand fetch stage in the pipeline. This is shown in Figure 3.3(b). In the example in Figure 3.3(a), when a subsequent `mul` instruction is decoded, a ‘1’ is written at the SCR position corresponding to the integer multiplier (indicated by the gray colored cell next to the `mul` instruction in the figure), initiating the restoration of the power supply to the integer multiplier. Therefore, by the time the instruction enters the execute stage, the integer multiplier is active to perform computation. This aspect of our design obviates the need for separate wakeup instructions, thereby, reducing the number of overhead instructions significantly.

The ARMv7 architecture reserves some instruction formats for instruction extensions in its later implementations. For example, when the bit 6 of the original ARM multiply instruction is changed from ‘0’ to ‘1’, the instruction is ignored and it does not even generate the undefined instruction trap. This option can be used to define a new instruction requiring minimal changes in the control unit. Based on this knowledge, a possible format of the `sleep` instruction is shown in Figure 3.4(a).

31 28	27 22	21 17	16 8	7 4	3 0
Cond	0 0 0 0 0 0	opvec	X X X X X X X X X	1 1 0 1	X X X X

(a) Generic format

`sleep imul, fpadd, fpmul`

31 28	27 22	21 17	16 8	7 4	3 0
Cond	0 0 0 0 0 0	0 1 1 1 0	X X X X X X X X X	1 1 0 1	X X X X
		opvec			

(b) Sample sleep instruction

Figure 3.4 A possible format of `sleep` instruction.

The generic format of the `sleep` instruction is obtained by altering bit 6 in the original multiply instruction format from ‘0’ to ‘1’, as highlighted by the gray field in the figure. Bits 21 to 17, indicated as *opvec* in the figure, correspond to the functional units that are passed as operands. A ‘1’ in that bit vector indicates that the functional unit needs to be deactivated while a ‘0’ indicates that the sleep control should be left unaltered. Thus, the `sleep` instruction in Figure 3.4(b) with integer multiplier, FP adder, and FP multiplier translate to the bit vector “01110” as shown.

3.2.1.2 Design of Power-Gated Functional Units

The functional units are redesigned according to the specifications of the ARM processor. The integer functional unit comprises of an ALU, a barrel shifter, and a Booth’s multiplier. Since the ALU is frequently used, no power gating is incorporated into it. The floating point unit comprises of an adder, a multiplier, and a divide and square root unit, as in [45]. Structural VHDL descriptions were used to synthesize the functional units using a cell library characterized for latency and power in 70nm Predictive Technology Model (PTM) [46].

There is a tradeoff between the latency of the functional units and the leakage savings achieved. Narrower sleep transistors yield higher leakage savings at the expense of increased latency of the functional units. On the other hand, to incur negligible degradation in the latency of the functional

Table 3.1 Latencies of functional units

Functional unit	Pipeline stages	Latency (cycles)
Integer ALU	0	1
Barrel Shifter	0	1
Integer Multiplier	1	16
FP Adder	4	6
FP Multiplier	3	10
FP Div-Sqrt Unit	4	19

units, the width of the sleep transistors have to be increased. In the results reported later in Section 3.3, leakage savings indicated correspond to keeping the latency degradation minimal or negligible (less than 2%). The leakage savings can be significantly increased if we allow further tradeoff with latency. The area overhead of the power gated functional units is 11% in terms of the width of the NMOS transistors. Based on the specifications of the area optimized ARM cores, the clock period was assumed to be 10 ns for the purpose of characterization. The latencies of the functional units (both regular and power gated versions) for the ARM core in terms of the number of cycles are listed in Table 3.1.

3.2.2 Software Component

The main task of the software component is to analyze the program behavior and predict regions in the program where certain functional units are expected to be idle during the execution of the program so that those functional units may be power gated. This translates to finding out maximal subgraphs in a CFG of a program pertaining to the idleness of each functional unit. Once these subgraphs are found, sleep instructions should be inserted at the entry so that the functional units are deactivated when the program control enters these subgraphs during its execution.

The control flow semantics of a program are represented in the form of a control flow graph (CFG), in which the vertices represent the basic blocks and the edges represent the transfer of control flow between the basic blocks. A basic block is defined as a straight-line code sequence with no control instruction. In the example in Figure 3.5, the code sequence multiplies two matrices

```
flag ← TRUE
for i ← 1 to m    ▷ loop $l_1$
   for j ← 1 to m    ▷ loop $l_2$
      sum ← 0
      for k ← 1 to n    ▷ loop $l_3$
         sum ← sum + A[i,k] * B[k,j]
      end for
      C[i,j] ← sum / norm
   end for
   for j ← 1 to i−1    ▷ loop $l_4$
      if (C[i,j] != 0)
         flag ← FALSE
      end if
   end for
end for
return flag
```

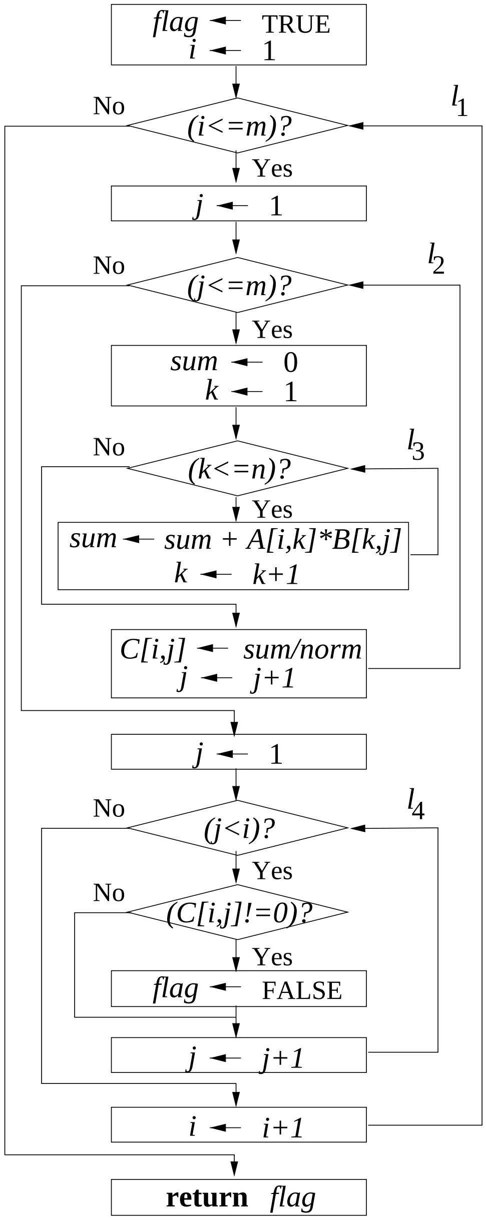


Figure 3.5 A sample piece of code and its CFG

of orders $(m \times n)$ and $(n \times m)$, normalizes the resultant matrix with respect to a parameter, *norm*, and returns a boolean value indicating if the resultant square matrix is an upper triangular matrix.

3.2.2.1 Identifying Potential Power-Gating Regions in the CFG

While generating the CFG for the source program, each vertex is annotated with the functional unit requirement of the basic block represented by that vertex. Consider a subgraph of the run-time trace of a program. The run-time trace is formed by tracing the edges of the CFG during the execution of the program. Let vertex u represent a basic block that does not use a particular functional unit, say r. Let P represent the set of all unique paths starting at vertex u and terminating at the earliest vertices such that the basic blocks represented by them use the functional unit r.

Let t_{clk} be the clock period and $N(p)$ be the number of clock cycles required to execute the instructions in path $p \in P$. Then the total time spent in executing the instructions in p is given by,

$$t_p = N(p)t_{clk} \tag{3.1}$$

Let $L(p)$, the *length* of the path p, be described in terms of the number of instructions in p. If the maximum IPC (instructions per cycle) count, which is determined by the *fetch* and *decode* width of a processor, is λ, then

$$L(p) \leq \lambda N(p) \tag{3.2}$$

The inequality in Equation (3.2) indicates that the IPC count for a code segment can be smaller than λ on account of cache misses, branch mispredictions, pipeline flushes, etc. Eliminating $N(p)$ from Equation (3.1) and Equation (3.2), we obtain,

$$t_p \geq \frac{1}{\lambda}L(p)t_{clk} \tag{3.3}$$

It can be noted, from Equation (3.3), that for an IPC count of at most 1, the execution time for the instructions in p can be lower bounded by the product of the number of instructions in p and the clock period.

Now, let the leakage energy saved per unit time by keeping the functional unit r deactivated is δ_r and that the dynamic energy overhead in activation and deactivation is Δ_r. The breakeven period,

t_{min}, is given by,

$$t_{min} = \frac{\Delta_r}{\delta_r} \tag{3.4}$$

So, for a path p to generate energy savings, the time spent in executing the instructions in p should be more than t_{min}, i.e., $t_p \geq t_{min}$. This is satisfied if,

$$\begin{aligned} & \frac{1}{\lambda}L(p)t_{clk} \geq t_{min} = \frac{\Delta_r}{\delta_r} \\ \Rightarrow \quad & L(p) \geq \frac{\lambda\Delta_r}{\delta_r t_{clk}} = L_{th} \end{aligned} \tag{3.5}$$

The quantity $\lambda\Delta_r/\delta_r t_{clk}$ is called the *threshold* length and is denoted by L_{th}. Equation (3.5) denotes that if the number of instructions in a path, p, is greater than the threshold length for functional unit, r, the energy savings obtained due to the low leakage state of r would be greater than the energy required to activate r, thereby, resulting in overall energy savings. If the relation, specified by Equation (3.5), is satisfied for all $p \in P$, then

$$t_p \geq t_{min}, \ \forall \ p \in P \tag{3.6}$$

which ensures that the leakage energy saved over all the paths in P exceeds the dynamic energy overhead incurred, thereby, giving overall savings.

Although, the above approach ensures energy savings in every path in the subgraph, it is a rather conservative approach. During the execution of the program, some of the paths in P may be greater in length and/or traversed more frequently than the rest. Thus, there is a possibility that the combined energy savings achieved along those paths exceeds the combined energy losses incurred along the rest of the paths, still resulting in overall energy savings. If n_p is the number of times the path $p \in P$ is traversed, then, using Equation (3.5), the leakage energy savings can be expressed as,

$$E_{savings} = \sum_{p \in P} n_p\left(\frac{L(p)\delta_r t_{clk}}{\lambda} - \Delta_r\right) \tag{3.7}$$

If ${}'P \subseteq P$ be the set of paths whose length is greater than L_{th} defined as, ${}'P = \{p \mid p \in P$ and $L(p) \geq L_{th}\}$, i.e., the set of paths whose length is greater than L_{th} then Equation (3.7) can be rewritten as,

$$\begin{aligned} E_{savings} &= \sum_{p \in {}'P} n_p\left(\frac{L(p)\delta_r t_{clk}}{\lambda} - \Delta_r\right) \\ &\quad - \sum_{p \in P - {}'P} n_p\left(\Delta_r - \frac{L(p)\delta_r t_{clk}}{\lambda}\right) \end{aligned} \tag{3.8}$$

From Equation (3.8), it can be observed that if subgraphs are identified in the program CFG which can result in program execution paths of length, at least, L_{th}, such that: (i) a functional unit is not required in those paths; and (ii) the paths are also traversed more often, we find potential regions in the program CFG which are good candidates for power gating.

3.2.2.2 Subgraphs Enclosed Within Loops

For the purpose of identifying regions in a program, not only the functions but also the loops within the functions are considered. Considering program regions enclosed within entire functions will provide a poor granularity in finding opportunities for power gating. Moreover, since the execution of loops usually varies dynamically with input data, a program usually spends a variable fraction of execution time in subgraphs enclosed within loops. In this context, we propose and use the notion of *loop hierarchy trees* (LHTs), which is essentially a tree data structure that captures the partial ordering of nested loops within a function in the source program. At compile time, the LHTs are used to determine long intervals between successive uses of the functional units thus identify-

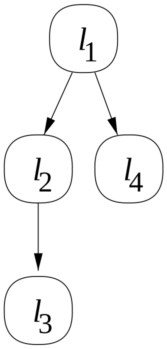


Figure 3.6 LHT for CFG in Figure 3.5

ing opportunities for applying power gating to the functional units. During the static code analysis phase, for each function in the source program, a forest of the LHTs is created. Each vertex of a LHT denotes a loop in the source program and its children denote the loops that are nested immediately within that loop. Each vertex is annotated with the functional unit requirement of the loop corresponding to that vertex. Figure 3.6 shows the LHT for the example in Figure 3.5. Loop l_1 has 2 nested loops at the same level, l_2 and l_4. In the LHT, this translates to l_1 being the root of the tree, and, l_2 and l_4 being its child nodes. Since loop l_2 further has l_3 as a nested loop, l_3 is a child node of l_2 in the LHT. In the remainder of this article, we use the term *child loop* with respect to a parent loop to denote a loop that is immediately nested in the parent loop.

An essential property of the LHT is that it captures a partial ordering of the loops in a program. With respect to the example in Figure 3.6, loop l_1 comprises of a set of basic blocks that is a superset of the set of basic blocks that form loops l_2 or l_4. Therefore, the functional unit requirements of loops l_2 and l_4 are subsets of the functional unit requirements of loop l_1. If the loop l_1 does not require a functional unit, say r, it implies that both loops l_2 and l_4 do not require r. Therefore, if r is deactivated at the entry of l_1 then it will remain deactivated during the executions of loops l_2 and l_4 as well. Similar observation can be made for loops l_2 and l_3 as well. However, since loops l_2 and l_4 do not share any vertices of the CFG among themselves, their functional unit requirements are independent of each other.

During the dynamic profiling of the source program, the vertices in the CFG of the program are annotated with the corresponding basic block execution counts and the vertices in the LHT are annotated with the corresponding loop execution counts. The execution count of a loop is the number of times the loop is entered during the dynamic profiling of the program. This is the same as the execution count of the entry basic block of the loop. Consider that $G_i = (V_i, E_i)$ is the CFG of the i^{th} function in the source program, where V_i is the set of vertices corresponding to the basic blocks in the function and E_i is the set of directed edges indicating the flow of control between the basic blocks. Let l_i be a loop in G_i, such that $S(l_i) \in V_i$ denote the set of vertices in l_i and C_i denote the set of child loops of l_i. The *total length* of the loop l_i, which is defined as the total number of instructions executed over all iterations of l_i during the dynamic profiling stage, can be represented by the recursive relation,

$$L_{tot}(l_i) = \sum_{c_i \in C_i} L_{tot}(c_i) + \sum_{v \in S(l_i) - \bigcup_{c_i \in C_i} S(c_i)} L_{bb}(v) g(v) \tag{3.9}$$

where, $L_{tot}(c_i)$ is *total* length of nested loop c_i, $L_{bb}(v)$ is length of basic block corresponding to vertex v, and $g(v)$ is the execution count of the basic block corresponding to vertex v. In words, the *total length* of a loop is calculated as the sum of the number of instructions executed over all iterations of its child loops and the number of instructions executed as part of the basic blocks in the loop, which are not part of the child loops. Then, the *average length* of loop l_i is defined as the average number of instructions executed during one iteration of l_i and is given by,

$$L_{avg}(l_i) = \frac{L_{tot}(l_i)}{f(l_i)} \tag{3.10}$$

where, $f(l_i)$ is the execution count of loop l_i. Similarly, the *average length* of any of the child loops c_i, of l_i, is given by

$$L_{avg}(c_i) = \frac{L_{tot}(c_i)}{f(c_i)} \tag{3.11}$$

where, $f(c_i)$ is the execution count of loop c_i. Substituting Equation (3.10) and Equation (3.11) in Equation (3.9), we get the following recursive relation for $L_{avg}(l_i)$,

$$\begin{aligned} L_{avg}(l_i) &= \frac{1}{f(l_i)} \Big(\sum_{c_i \in C_i} L_{avg}(c_i) f(c_i) \\ &\quad + \sum_{v \in S(l_i) - \bigcup_{c_i \in C_i} S(c_i)} L_{bb}(v) g(v) \Big) \end{aligned} \tag{3.12}$$

The average lengths of the loops are used to make decisions about inserting sleep instructions in cases where the loop requires a functional unit but only a subset of the nested loops within that loop have the same functional unit requirement. The L_{avg} values for all the loops in the CFG can be calculated by running a *breadth first search* from the root of each tree in the LHTs. For the functions that are called from within a loop, the entire function is considered as a basic block in the above formulation in Equation (3.10). Also, each function in the source program is separately analyzed for insertion of sleep instructions.

3.2.2.3 Insertion of Sleep Instructions

To find the locations in the program to insert sleep instructions, a *depth first traversal* of the nodes in each LHT is performed starting at its root. This traversal is done once for each functional unit and is terminated as soon as it is found that the entire loop corresponding to the node does not use the functional unit. Since dynamic profiling at the basic block level can only acquire the total execution count of each loop in the program, we normalize the execution count of a child loop with respect to its parent loop to quantify the *iterative degree* of the child loop. Iterative degree of a loop really means the number of executions of the loop per execution of its parent loop. We define the normalized average length of loop x, $L_{norm}(x)$, as,

$$L_{norm}(x) = L_{avg}(x)\frac{f(x)}{f(parent(x))} \tag{3.13}$$

where, $f(x)$ is the execution count of loop x, $f(parent(x))$ is the execution count of the parent loop of loop x. Refering to the CFG in Figure 3.5, loop l_3 does not have a division operation while its parent loop, l_2, has a division operation. Therefore, loop l_2 becomes a potential subgraph for power gating the divider. However, the amount of savings obtained by keeping the divider powered down depends on the number of times loop l_3 executes each time loop l_2 is executed. It can be observed that loop l_3 iterates m^2n times, whereas, loop l_2 iterates m^2 times. Therefore, if $m = 8$ and $n = 4$, $f(l_3) = 8^2 \cdot 4 = 256$, while $f(l_2) = 8^2 = 64$. On the other hand, if $m = 4$ and $n = 16$, $f(l_3) = 4^2 \cdot 16 = 256$, but $f(l_2) = 4^2 = 16$. Thus, although the total number of times loop l_3 is executed is the same (256) in both cases, the latter case is more favorable for power gating the divider since there are 4 times as many instructions executed between two successive division operations in this case compared to those in the former. Thus, the normalization process considers nested loops that are small but iterative enough to be potential subgraphs for power gating.

Algorithm 1 INSERT-SLEEP(F) ▷ Algorithm to insert sleep instructions

```
1: $S \leftarrow \Phi$
2: for all tree $T \in F$ do
3:     for all functional unit $r \in R$ do
4:         INSPECT($root(T), r$)
5:     end for
6: end for
7: UNIQIFY($S$)
```

Algorithm 1 represents the pseudocode for the routine INSERT-SLEEP. The set S contains the sleep instruction locations. It is set to a NULL set at the beginning (line 1). In lines 2-6, the two *for* loops iterate over all the LHTs in the LHT forest F for each functional unit in R and calls the routine INSPECT at the root of each tree. After all the LHTs are inspected, in line 7, the routine UNIQIFY is called to uniqify all the sleep instructions in set S. It should be noted that the normalized length of all the loops, $L_{norm}()$, in the program and the threshold number of instructions for each functional unit, $L_{th}()$, are already known before the routine INSERT-SLEEP is called.

Algorithm 2 INSPECT(x, r) ▷ Algorithm to inspect loops

```
1: if $r \notin res(x)$ then
2:     if $L_{norm}(x) \geq L_{th}$ then
3:         $S \leftarrow S \cup \{e(x), r\}$
4:     end if
5: else
6:     for all $y \in C_x$ do
7:         INSPECT($y, r$)
8:     end for
9: end if
```

Algorithm 2 represents the pseudocode for the routine INSPECT which takes a vertex of a LHT, x, and a functional unit, r, as the arguments. In lines 1-4, it checks whether r is used in x or not. The functional unit requirement for loop x is given by $res(x)$. If r is not used in loop x, it checks the normalized length of x, $L_{norm}(x)$. If $L_{norm}(x)$, is greater than the threshold number of instructions, L_{th}, the location is marked (added to the set S) for insertion of a sleep instruction for deactivating functional unit, r. The location is a two element tuple consisting of the basic block leading to the loop, $e(x)$, and the functional unit, r. If, however, r is used in the loop x, it calls

INSPECT recursively on all the child vertices of x (lines 5-8). In other words, the loops nested in x are inspected to explore the possibility of power gating r.

Algorithm 3 UNIQIFY(S) $\triangleright$ Algorithm to uniqify the sleep instructions

```
1: Sort the elements in $S$
2: for all unique locations do
3:     Merge all the FUs to form one sleep instruction
4: end for
```

The routine UNIQIFY, shown in Algorithm 3, replaces separate sleep instructions which might have been inserted for each functional unit at the same location with a single sleep instruction with all those functional units as its operands. In line 1, the elements in S are sorted in ascending order by their locations. In lines 2-4, a unique sleep instruction is generated for all the functional units which have the same location.

3.2.2.4 Time Complexity

Since the routine INSPECT implements a DFS traversal in T, its worst-case time complexity is $O(|V|+|E|)$. However, since for a rooted tree, $|E|=|V|-1$, the worst-case time complexity of INSPECT is $O(|V|)$. Note that $|V|$ denotes that number of loops in a function. Therefore, the worst-case time complexity of INSPECT is linear in the number of loops in the function corresponding to the LHT T. The time complexity of routine UNIQIFY is $O(|S|\log|S|)$, where S is the number of sleep instructions inserted. The routine INSERT-SLEEP makes calls to INSPECT for all the LHTs for each functional unit. Since there is one LHT for each function in the program, the worst-case time complexity of lines 1-6 in INSERT-SLEEP is $O(n*|R|)$, where n is the total number of loops in the entire program across all functions and $|R|$ is the total number of power gating enabled functional units. However, $|R|$ is constant since the number of functional units is constant. Thus, the worst-case time complexity of lines 1-6 in INSERT-SLEEP is $O(n)$, which is linear in the number of loops in the program. However, the number of sleep instructions that can be inserted can be a maximum of the number of loops present and, therefore, $|S| \leq n*|R|$. Thus, the worst-case time complexity of the call to the routine UNIQIFY in line 7 of INSERT-SLEEP is $O(n\log n)$. Therefore, the worst-case time complexity of routine INSERT-SLEEP is $O(n\log n)$.

3.3 Experimental Setup and Results

3.3.1 Energy Component Calculations

Figure 3.7 shows the instantaneous power graph superimposed on the instantaneous voltage graph at the virtual ground for a footer sleep transistor configuration. The graph depicts the significant time intervals for the calculation of the various energy components of a power gated functional units. V_{vrgnd} refers to the voltage at the virtual ground. P_{inst} refers to the instantaneous power dissipated by the circuit. At time, t_0, when the sleep transistor is switched OFF, V_{vrgnd} rises to V_{dd} by time t_1. From time t_1 to t_2, all the capacitances in the circuit reach their final charge. During this interval, the circuit still dissipates instantaneous power which slowly approaches the steady state leakage power in its OFF state, $P_{L_{OFF}}$. Thus, the overhead energy, $E_{t_0-t_2}$, in deactivating the circuit is the total energy dissipated during the interval t_0 to t_2, and is given by the area under the curve for P_{inst} during the interval t_0 to t_2 (indicated by shaded area in dark gray in Figure 3.7):

$$E_{t_0-t_2} = E_{t_0-t_1} + E_{t_1-t_2} \tag{3.14}$$

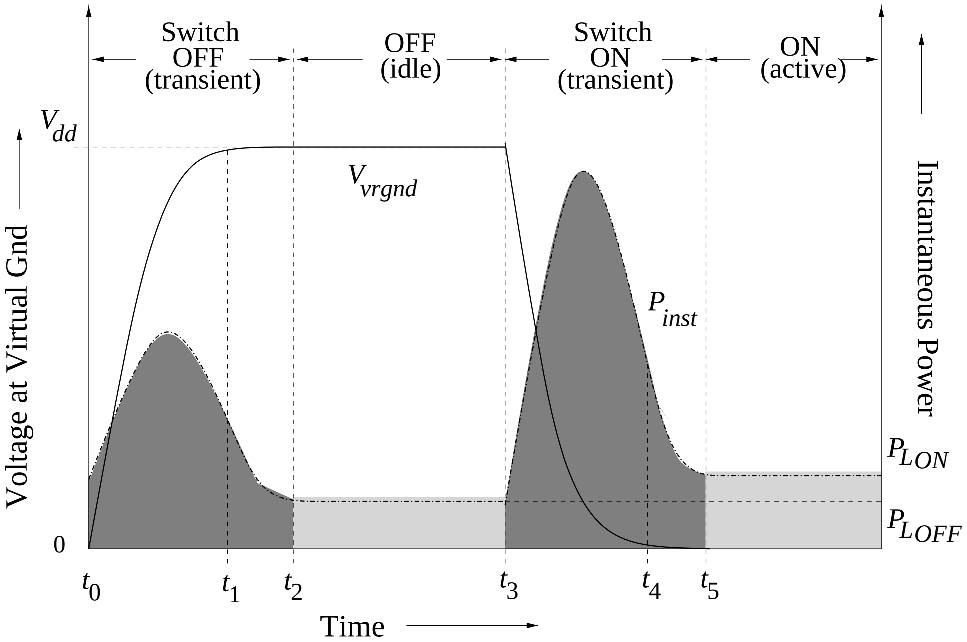


Figure 3.7 Superimposed voltage and power graphs as functions of time

It is only after time t_2 that the circuit starts to dissipate $P_{L_{OFF}}$, which is the steady state leakage power dissipation of the circuit in sleep state. The overhead energy required while activating the circuit is considered next. At time t_3, the sleep transistor is switched ON, V_{vrgnd} falls to V_{gnd} by time t_4. From time t_4 to t_5, all the capacitances reach their final charge. The overhead energy in activating the circuit is the total energy dissipated during the interval t_3 to t_5 (indicated by shaded area in dark gray in Figure 3.7, denoted by $E_{t_3-t_5}$.

$$E_{t_3-t_5} = E_{t_3-t_4} + E_{t_4-t_5} \tag{3.15}$$

After time t_5, the circuit starts to dissipate $P_{L_{ON}}$. This is the steady state leakage power dissipation of the powered circuit when it is idle.

The calculations of the energy components of the library components are performed using HSPICE simulations with the 70nm PTM files. The activation and deactivation overhead energies (areas indicated by dark gray shades) are calculated using piecewise linear approximation of the curves. The dynamic energy components are computed by averaging the power reported by HSPICE over pseudorandomly generated inputs. During the calculation of the dynamic energy components, the sleep transistors were turned on. The energy components of each functional unit is estimated as the sum of the energy components of its constituent component instances. For a functional unit r, the dynamic energy overhead incurred during activation and deactivation of r, Δ_r, is calculated as,

$$\Delta_r = E_{t_0-t_2} + E_{t_3-t_5} \tag{3.16}$$

Table 3.2 shows the energy values and the threshold lengths that are estimated for each of the functional units.

Table 3.2 Average energy components of functional units

Functional Unit	Leakage No power gating (nJ/cycle)	Leakage OFF (nJ/cycle)	Leakage ON (nJ/cycle)	Overhead activation (nJ)	Overhead deactivation (nJ)	Dynamic component (nJ/operation)	L_{th}
Int ALU*	1.26E-04	-	-	-	-	-	-
Barrel Shifter	3.01E-04	1.64E-04	3.12E-04	5.68E-02	2.59E-04	6.13E-02	415
Int Multiplier	2.61E-04	1.46E-05	2.65E-04	6.07E-02	2.34E-04	7.23E-02	530
FP Adder	9.18E-04	4.93E-04	9.20E-04	1.91E-01	8.82E-04	2.16E-01	453
FP Multiplier	8.13E-04	4.23E-04	8.17E-04	1.66E-01	6.73E-04	1.93E-01	428
FP Div-Sqrt Unit	1.60E-03	8.74E-04	1.64E-03	3.80E-01	1.34E-03	4.16E-01	523

*Since the integer ALU is not power gated, its entries are omitted from the cells pertaining to the power gated version.

3.3.2 Cycle-Accurate Simulation

We used the SimpleScalar-ARM distribution [47] for modeling the proposed embedded architecture and MiBench embedded benchmark suite [44] for experimentation. We chose two benchmarks from each of the categories of applications in MiBench which are: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. The object code for the program was disassembled using the `gcc` tools for ARM, available with the distribution. Static code analysis was performed on the CFG generated for the functions in the source program. Although sleep instructions were not inserted into the standard library functions, they were analyzed for their functional unit requirements. The profiling tool with the SimpleScalar distribution, `sim-profile`, was used to perform dynamic profiling of the program and `sim-outorder` to perform cycle-accurate simulation after the insertion of the sleep instructions.

The embedded processor configuration is based on the Intel StrongARM SA-1 processor (Table 3.3). All the experiments were performed on a set of benchmarks from the MiBench suite, whose details are tabulated in Table 3.4. For each benchmark, the leakage energy consumption during its execution on a processor with power gated functional units was compared with the leakage energy consumption on a processor without any power gated functional units. The savings in leakage energy for integer benchmarks and the floating point benchmarks are plotted in Figure 3.8 and Figure 3.9, respectively.

The leakage energy savings achieved for the integer benchmarks range from 25.6% to 37.4%, resulting in an average of 31.1%. For the FP benchmarks, the leakage energy savings range from 22.6% to 30.4%, resulting in an average of 26.8%. Further, the fraction of total simulation cycles

Table 3.3 ARM processor configuration. [44]

Fetch Queue (instructions)	2
Branch Predictor	Not-taken
Fetch & Decode Width	1
Issue width	1
Instruction L1 Cache	16K, 32-way
Data L1 Cache	16K, 32-way
L2 Cache	None
Memory (bus width, first block latency)	4-byte, 12 cycle

Table 3.4 Benchmark details

Benchmark Name	Category	Dynamic Instruction Count	Benchmark Type
bitcount	Automotive	49.6	Integer
qsort	Automotive	43.6	Integer
jpeg decode	Consumer	6.7	Integer
lame	Consumer	97.2	Floating Point
dijkstra	Network	64.9	Integer
patricia	Network	103.9	Integer
rsynth	Office	57.9	Floating Point
ispell	Office	8.4	Integer
fft	Telecomm.	52.7	Floating Point
fft-inverse	Telecomm.	65.8	Floating Point
rijndael	Security	30.7	Integer
sha	Security	13.6	Integer

for which each of the units was power gated is shown in Figure 3.10 (integer units) and Figure 3.11 (floating point units). Since there was less than 0.1% performance degradation in terms of the additional cycles required to execute the sleep instructions, those numbers are not included in the table.

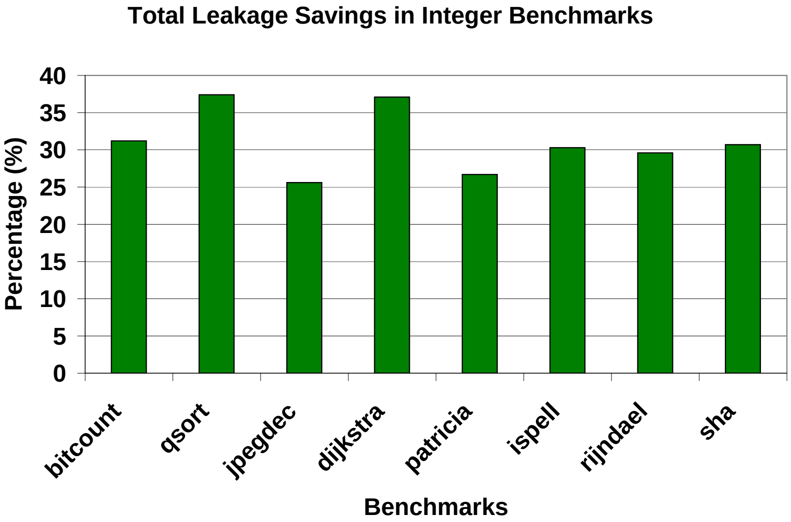


Figure 3.8 Total leakage energy savings in integer benchmarks. These numbers are reported considering only the integer units (the shifter and the integer multiplier).

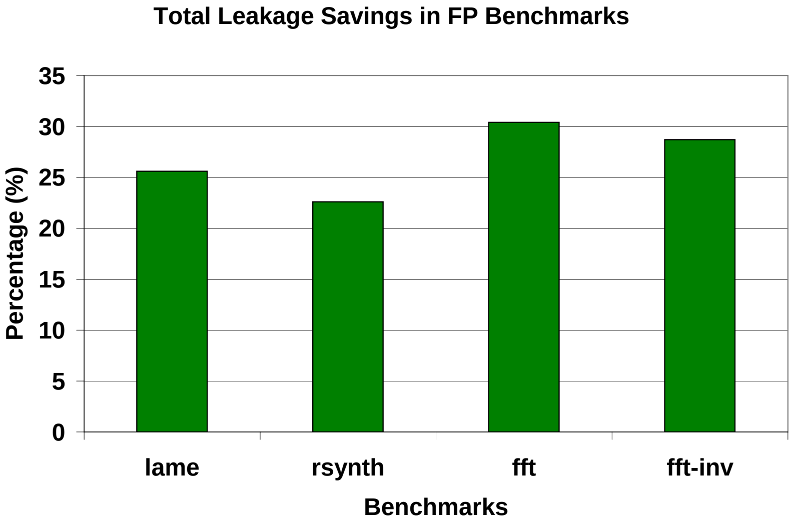


Figure 3.9 Total leakage energy savings in floating point benchmarks. These numbers are reported considering both integer and floating point functional units.

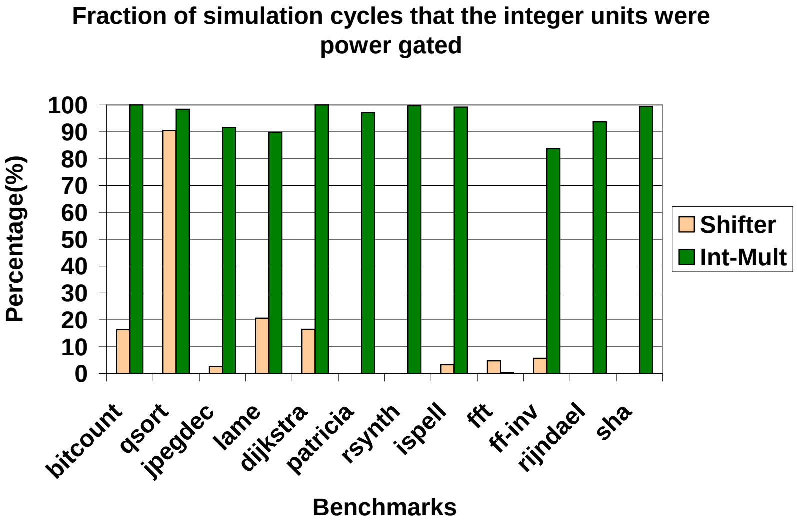


Figure 3.10 Fraction of total simulation cycles for which the integer units were power gated. Since both integer and floating points units use integer units, these numbers are reported for all the benchmarks.

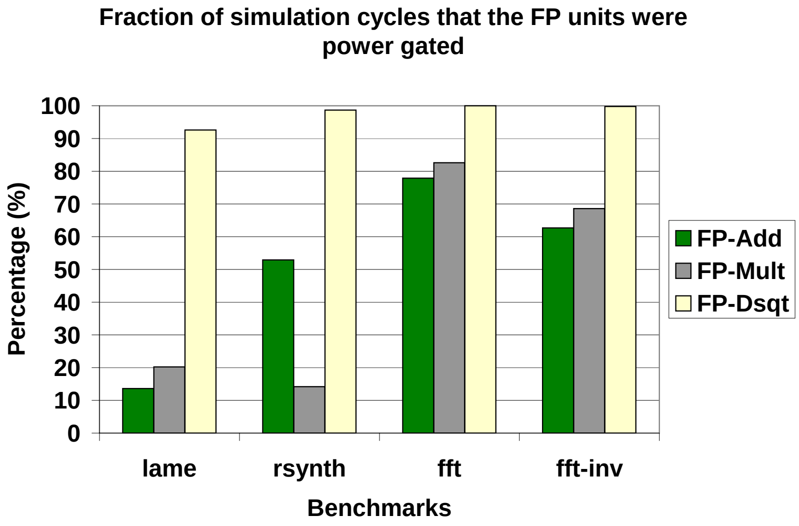


Figure 3.11 Fraction of total simulation cycles for which the floating point units were power gated. These numbers are reported only for the floating point benchmarks because integer benchmarks do not use floating point units.

3.4 Discussion

In this chapter, a framework for power gating the functional units in the datapaths of embedded processor architectures in order to achieve leakage reduction at the compiler level was described. The proposed scheme is based on the fact that most embedded processors operate with longer clock cycles than the superscalar processors. The longer clock cycle helps in activation of the needed functional units by using control signals from the decode stage. The use of more sophisticated techniques such as dual sleep transistors and charge recycling devices in the design of the functional units need to be investigated for improving the power gating opportunities during program execution.

CHAPTER 4

IMPACT OF COMPILER OPTIMIZATIONS ON POWER GATING

In this chapter, we investigate the impact of compiler optimization techniques on power gating implemented with compiler and architectural support to reduce leakage in the arithmetic functional units of microprocessors. Power gating can be implemented using sleep transistors to selectively deactivate functional units in the datapath that would remain idle for sustained periods of time during program execution. During compile time, the idle times for various functional units can be identified and sleep instructions can be inserted within the code to deactivate those units during program execution. They can be subsequently awakened through use of hardware control signals when their need is detected during instruction decode stage. In this context, the effectiveness of power gating for leakage reduction depends on the ability of the compiler to identify the program regions in which all or some of the functional units are expected to remain idle for periods long enough to provide leakage savings considering the overheads involved. However, the usage of resources in a program is dependent not only on its source code description but also on the code transformations frequently performed by the compiler for improving the performance of the executable binary. Therefore, the effectiveness of compiler-directed power gating could be largely dependent on the compiler optimizations performed during code generation. In this paper, we develop a leakage aware compilation flow based on a comprehensive study of the combined effects of various compiler optimization techniques on power gating through analysis and simulations. While GCC is used as the compiler framework for our study, the embedded processor is modeled based on the ARM architecture modified to include the hardware support needed for power gating. Simplescalar-ARM is used for evaluating energy and performance results for the benchmarks chosen from MiBench and MediaBench suites. Experimental results indicate that, through the use of different compiler optimization techniques, while the leakage energy savings due to power gating in individual functional units could be increased by 15% in benchmarks from MediaBench, the same for benchmarks from Mibench could be increased by up to 9 times.

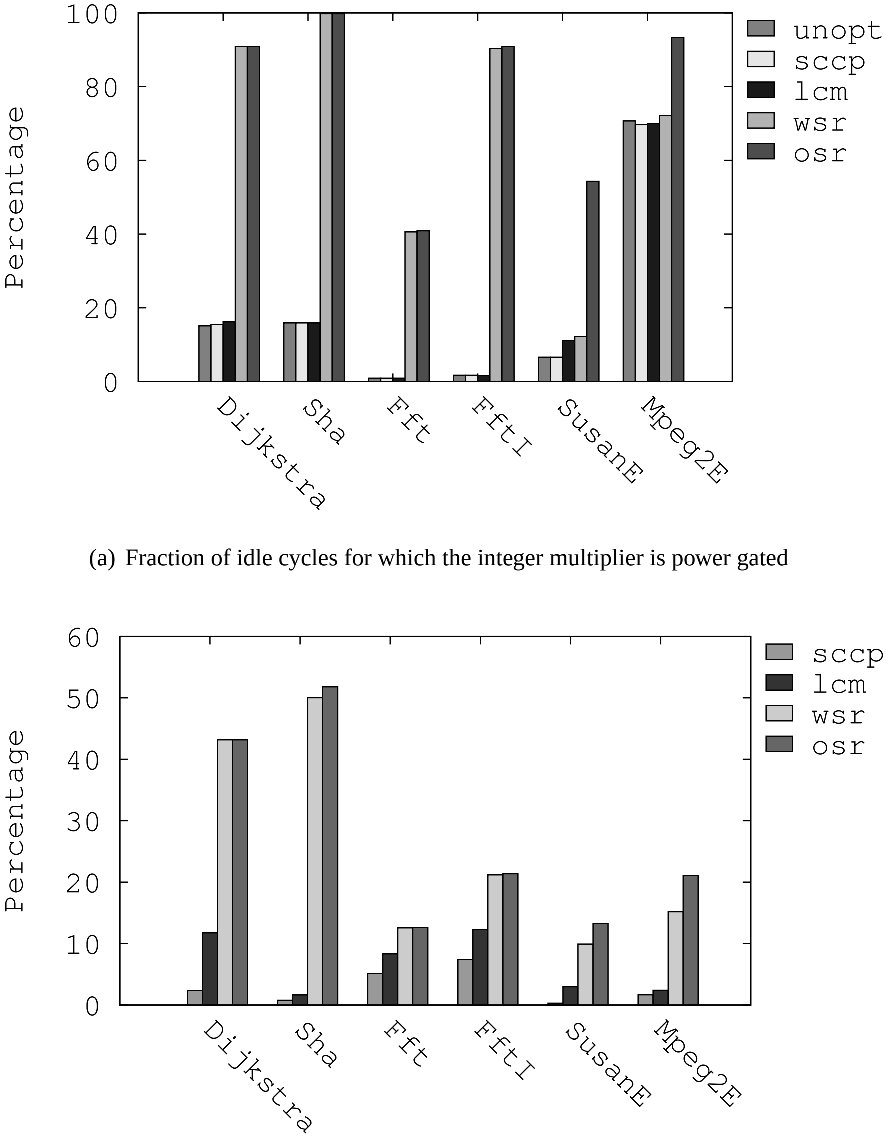
4.1 Motivation

An important task in compiler-directed power gating is to identify program regions in which functional units are expected to be idle for long periods of time. However, the idleness or usage of functional units is directly influenced by the code transformations performed by the compiler during code generation. Such transformations are often performed by a compiler to improve the performance of the program executable. The plots in Figure 4.1 show the impact of certain compiler optimizations for performance improvement on power gating of the integer multiplier in an embedded microprocessor. Table 4.1 describes the legends used in the plots. The benchmarks `Dijkstra` and `Sha` are integer benchmarks, whereas the rest are floating point benchmarks. For each benchmark, the fraction of idle cycles for which the multiplier is kept deactivated during the entire program runtime is plotted in Figure 4.1(a). It can be seen that the strength reduction optimizations, which remove redundant integer multiplications in the program, improve power gating of the multiplier significantly. The savings in the total energy dissipated due to leakage in the functional units is shown in Figure 4.1(b). This suggests that compiler optimizations play a critical role in the effectiveness of compiler-directed power gating techniques. Therefore, it is of immense interest to investigate the effect of a larger spectrum of performance-improving code optimizations on the opportunities for power gating of functional units.

Table 4.1 Description of the legends in Figure 4.1

Legend	Description
`unopt`	No optimizations
`sccp`	Sparse Conditional Constant Propagation (SCCP) [49]
`lcm`	SCCP + Lazy Code Motion (LCM) [50]
`wsr`	SCCP + LCM + Weak Strength Reduction (WSR) [51]
`osr`	SCCP + LCM + Operator Strength Reduction (OSR) [52] + WSR

In [53], Kandemir et al. studied in detail the effects of a few high level compiler optimizations on dynamic power and energy consumption in microprocessors. Since leakage power has



(a) Fraction of idle cycles for which the integer multiplier is power gated

(b) Leakage energy saved due to power gating after code optimizations over that in the unoptimized case

Figure 4.1 Impact of a few compiler optimizations on power gating [48]

become a critical aspect of low power VLSI design in the recent years, it is important to study the factors that influence leakage power consumption of a system directly or indirectly. In this work, we use the open source production quality compiler, GNU Compiler Collection (GCC) [54], as the compiler framework to study the influence of performance enhancing compiler optimizations on compiler-directed power gating in embedded processors. The main contribution of our work is a leakage aware compilation flow that can be used to selectively apply compiler optimization techniques on an application with an objective to improve power gating opportunities of the functional units in an embedded microprocessor during runtime, thereby increasing leakage energy savings. The Simplescalar-ARM distribution [55] is used to perform extensive simulations on a set of floating point benchmarks from Mibench [44] and MediaBench [56] suites.

4.2 Impact of Compiler Optimizations on Power Gating

In this section, the analyses of various compiler optimizations are presented from the perspective of power gating [57]. The optimizations that apply locally to a procedure are categorized as intraprocedural optimizations and those that apply across procedures are categorized as interprocedural optimizations.

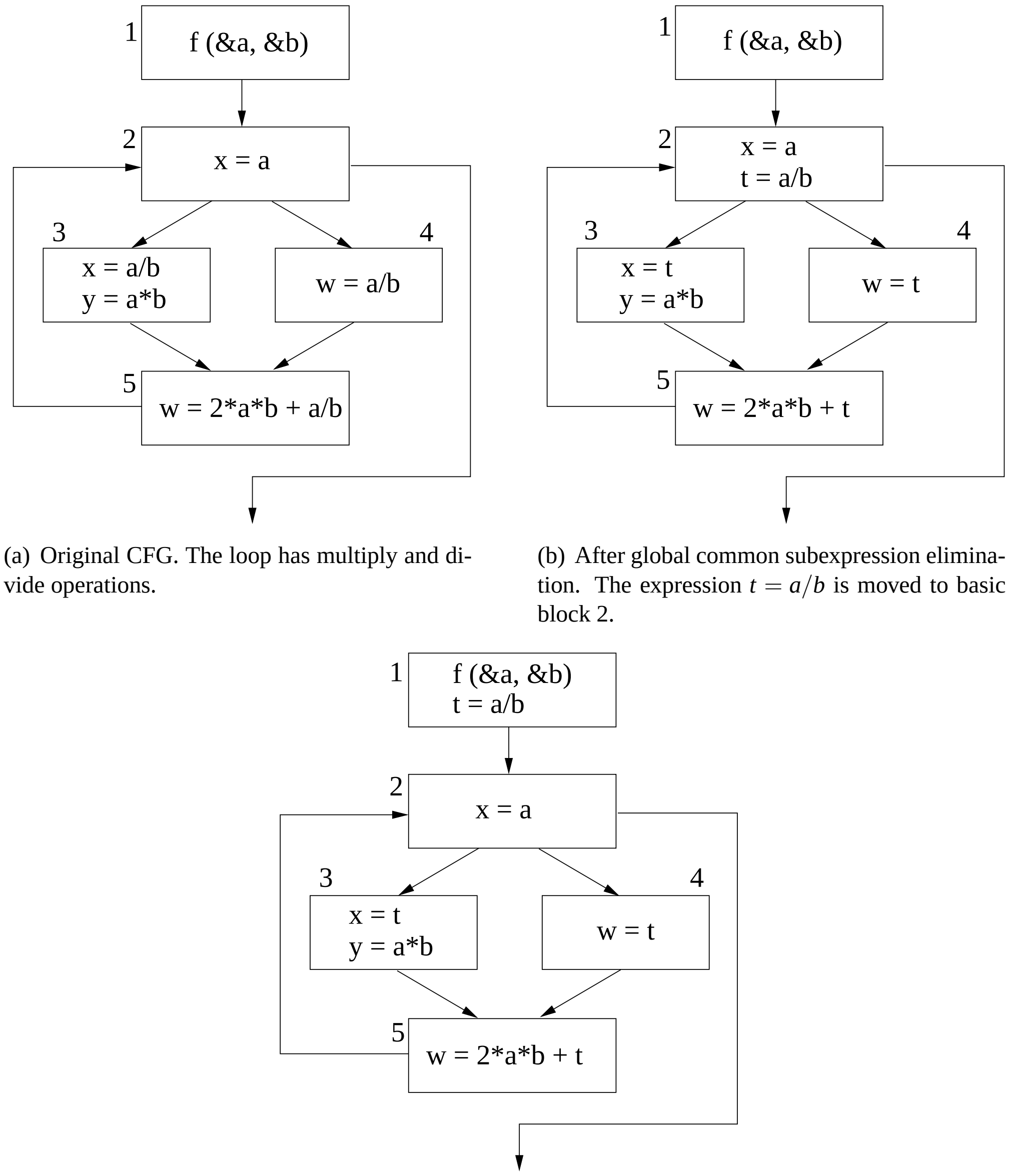
4.2.1 Intraprocedural Optimizations

The optimizations discussed here are performed on the control flow graph of a procedure and they deal with the removal of redundant arithmetic operations within the code. Such optimizations directly influence the opportunities for keeping one or more functional units deactivated in a program region and, therefore, can be extremely effective in generating code that is conducive to power gating.

4.2.1.1 Dominator Optimizations

These optimizations use the dominance information [57] of the control flow graph for the procedure to perform various optimization tasks. Common dominator optimizations include *dead code elimination*, *copy and constant propagation*, *common subexpression elimination* (CSE), etc.

Figures 4.2 and 4.3 illustrate the impact of CSE techniques on the power gating opportunities in a procedure. In Figure 4.2(a), the expression a/b (procedure $f()$ writes to a and b) is *fully* redundant in the basic block 5 because it is computed in both of its predecessor blocks, 3 and 4. However, the expression $a \cdot b$ is *partially* available in basic block 5 because it is not computed



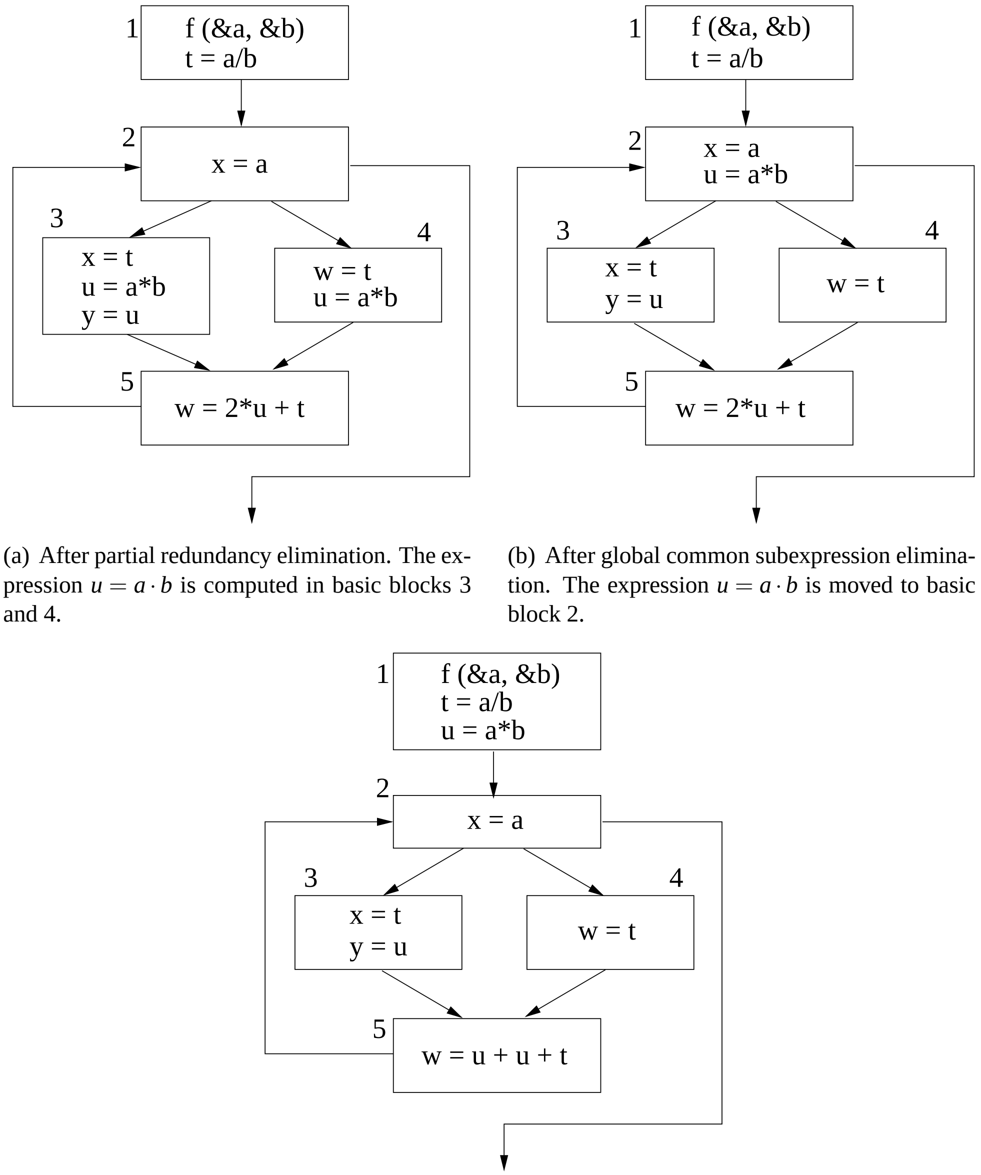
(a) Original CFG. The loop has multiply and divide operations.

(b) After global common subexpression elimination. The expression $t = a/b$ is moved to basic block 2.

(c) After loop invariant code motion. The expression $t = a/b$ is moved out of the loop. Now the loop does not have any divide operations.

Figure 4.2 Example to illustrate the impact of global common subexpression elimination on the usage of functional units

in basic block 4. Global CSE (GCSE) moves the computation of a/b to basic block 2 (the loop header) (Figure 4.2(b)). When *loop invariant code motion* (described later in Section 4.2.1.2) is performed (Figure 4.2(c)), the expression is moved out of the loop, thereby making the divider



(a) After partial redundancy elimination. The expression $u = a \cdot b$ is computed in basic blocks 3 and 4.

(b) After global common subexpression elimination. The expression $u = a \cdot b$ is moved to basic block 2.

(c) After loop invariant code motion and weak strength reduction. The expression $u = a \cdot b$ is moved out of the loop and $2 \cdot u$ is replaced with $u + u$. Now the loop does not have any multiply operations.

Figure 4.3 Example to illustrate the impact of partial redundancy elimination on the usage of functional units

idle in the loop. Another optimization, known as *partial redundancy elimination* (PRE), makes the expression $a \cdot b$ fully redundant in basic block 5 by introducing the statement $u = a \cdot b$ in basic block 4 (Figure 4.3(a)). GCSE can now move the expression $a \cdot b$ to basic block 2 (Figure 4.3(b)). When code motion and weak strength reduction (described in Section 4.2.1.3) are performed subsequently, this expression is moved out of the loop, thereby making even the multiplier idle in the loop (Figure 4.3(c)). It should be noted, however, that GCSE and PRE are not always performed by the compiler because the former may increase register pressure, while the latter may increase code size. If the IEEE or ISO floating point precision rules are relaxed during compilation, such arithmetic transformations can be applied to floating point types as well.

4.2.1.2 Loop Optimizations

The optimizations described in this category are performed on loops to remove redundant operations from the body of the loops. Figure 4.4 illustrates how *loop invariant code motion* and *strength reduction* on induction variables are effective in doing so, thereby improving power gating opportunities in the loops of a procedure. In Figure 4.4(a), the expression $c[i] * k$ evaluates to the same value in every iteration of the inner for-loop. Therefore, this computation is moved out of that loop and a new temporary t is used to hold the result of that computation, which is subsequently used in the inner for-loop. Therefore, a sleep instruction, putting the multiplier to sleep, can be inserted before the entry of the inner for-loop. In Figure 4.4(b), the result of the computation $i * k$ across the iterations of the loop form an arithmetic progression with a common difference of k. This is because k is constant across all the iterations of the loop, while i gets incremented by 1 in each iteration of the loop. The transformed code, therefore, does not have any multiply operation and the multiplier can be turned off at the entry of the loop.

4.2.1.3 Machine Dependent Optimizations

The primary purpose of these optimizations is to improve the pipeline efficiency by performing local code transformations with the knowledge of the target machine. Figure 4.5 illustrates two peephole optimizations that eliminate the integer multiply instruction from a code sequence, thereby increasing opportunities for power gating. Figure 4.5(a) describes the generation of complex addressing operands while performing a load operation. The sequence of instructions on the

Original code

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

Transformed code

```
for (i = 0; i < n; i++)
{
  t = c[i] * k;
  /*Turn off Multiplier */
  for (j = 0; j < n; j++)
    a[j] = b[j] + t;
}
```

(a) Loop invariant code motion

Original code

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

Transformed code

```
t = 0;
/*Turn off Multiplier */
for (i = 0; i < n; i++)
{
  a[i] = b[i] + t;
  t = t + k;
}
```

(b) Strength reduction

Figure 4.4 Examples of loop optimizations that improve the opportunities for power gating. In (a), the multiply operation is moved from the inner for-loop to the outer one. The multiplier can now be turned off at the entry of the inner for-loop. In (b), the multiply operation is eliminated in the for-loop by replacing it with an equivalent series of add operations. If IEEE or ISO floating point precision rules are relaxed, strength reduction can also be applied to eliminate floating point multiply operations.

left column (to load an element from an array storing elements of size 4 bytes) is replaced by a single instruction on the right in which the base address and the offset can be directly specified. Figure 4.5(b) illustrates weak strength reduction [51, 58] in which multiplication with a constant operand (119) may be replaced by a sequence of addition, subtraction, and shift instructions. This transformation, however, should be carried out only if it is estimated during compilation that it

```
   Original code                     Optimized code

mov      r1, #4               ldr      r3, [r0, r1, asl #2]
mul      r3, r2, r1
add      r1, r0, r3
ldr      r3, [r4]
```

(a) Loads with complex addressing

```
   Original code                     Optimized code

mov      r1, #119               mov      r1, r2, asl #4
mul      r3, r2, r1             add      r1, r1, r2
                                mov      r3, r1, asl #3
                                sub      r3, r3, r1
```

(b) Weak strength reduction

Figure 4.5 Examples of machine dependent optimizations in ARM that can improve opportunities for power gating. In the examples above, the multiply instructions are eliminated in the optimized codes.

would be beneficial (in terms of both performance and power) to do so considering the overhead of fetching the extra instructions and executing them.

4.2.2 Interprocedural Optimizations

These optimizations rely on the analyses performed by the compiler by considering the entire compilation unit as a whole. All the procedures are analyzed and a *call graph* is created. In a call graph, each vertex represents a procedure and an edge (u, v) implies that there is a call from procedure u to procedure v. Interprocedural analyses are performed on the call graph to compute the set of variables and memory locations that are modified and referenced by each procedure. This makes it possible for more effective intraprocedural optimizations to be performed. We discuss two

```
int scale(int s, int* x)
{
 sum = 0;
 /*Turn off Int. Mult. */
 for (i = 0; i < n; i++)
   sum = sum + x[i];
 /*Turn on Int. Mult. */
 return (s*sum);
}

void foo(void)
{
 ...
 for (i = 0; i < m; i++)
   b[i] = scale(5, a[i]);
 ...
}
```

(a) Original source in which the procedure, `scale()`, performs a multiply operation on its first formal argument, *s* and the sum of the elements in the array pointed to by *x*.

```
int scale(int* x)
{
 /*Turn off Int. Mult. */
 sum = 0;
 for (i = 0; i < n; i++)
   sum = sum + x[i];
 return ((sum ≪ 2) + sum);
}

void foo(void)
{
 ...
 for (i = 0; i < m; i++)
   b[i] = scale(a[i]);
 ...
}
```

(b) After interprocedural constant propagation is performed on (a), the first formal argument is removed and the arugment *s* is replaced with the constant value of 5 in the return statement. Weak strength reduction eliminates the multiply operation.

```
void foo(void)
{
 ...
 /*Turn off Int. Mult. */
 for (i = 0; i < m; i++)
 {
   sum = 0;
   for (j = 0; j < n; j++)
     sum = sum + a[i][j];
   b[i] = (sum ≪ 2) + sum;
 }
 ...
}
```

(c) After procedure inlining is performed on (a), the *for* loop within the procedure `scale()` becomes a nested loop in the *for* loop in callee procedure `foo()`. Weak strength reduction eliminates the multiply operation. The instruction to turn off the integer multiplier can now be inserted before entering the outer for-loop.

Figure 4.6 Example illustrating the impact of interprocedural optimizations on power gating opportunities of the integer multiplier

optimizations in this category, *interprocedural constant propagation* and *procedure inlining*, and their impact on power gating (Figure 4.6).

The original code (Figure 4.6(a)) has two procedures, `scale()` and `foo()`. The first procedure computes the sum of the elements in the array pointed to by its formal argument, *x*, and then scales the computed value by its second formal argument, *s*. The second procedure calls `scale()` for every row of an $(m \times n)$ array, *a*, with a constant scaling parameter, 5, and stores the values returned by `scale()` in array, *b*. In this case, the integer multiplier can be turned off at the entry of the for-loop and turned back on at the exit of the for-loop in `scale()`. However, since `scale()` is called *m* times, the performance and energy overheads involved in power gating increases as *m* increases.

Figure 4.6(b) shows the interprocedural constant propagation technique conceptually. The procedure definition for `scale()` is reduced to one with only a single argument. The argument *s* in the body of the procedure is replaced with the constant 5. Weak strength reduction can now replace $(5 \cdot sum)$ with $((sum \ll 2) + sum)$, thereby eliminating the multiply operation completely. Interprocedural constant propagation is beneficial in the cases where macros are used in the source code that assume constant values after preprocessing. When procedure inlining is performed (Figure 4.6(c)), the body of `scale()` is integrated into `foo()`, following which the multiply can be replaced with shift and add operations. Thus, only a single sleep instruction is required at the entry of the outer for-loop putting the integer multiplier to sleep. This significantly reduces the performance and energy overheads in implementing power gating.

4.3 Compiler-Directed Power Gating

Figure 4.7 describes the framework for compiler-directed power gating of functional units along with code optimizations. A compiler usually has multiple intermediate representations (IR) to represent a program during the entire compilation process. The source code of an application is translated into a high-level IR by the compiler front-end. High level tranformation, which include interprocedural optimizations, are performed at the high-level IR. Most of the machine-independent intraprocedural optimizations, like dominator and loop optimizations, are performed on an intermediate-level IR. Finally, the machine-dependent optimizations are performed on a low-level IR. All the

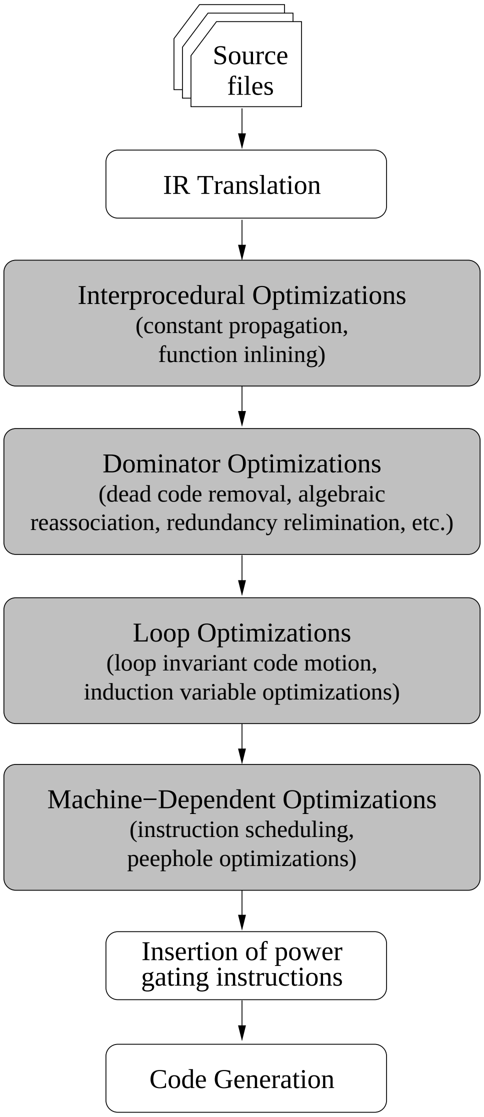


Figure 4.7 Framework for compiler-directed power gating of functional units with code optimizations

optimizations are organized as compiler passes which can be selectively turned ON or OFF with the help of optimization flags specified during compilation. The insertion of the sleep instructions is performed after all of the code optimizations have been performed and is, therefore, shown as part of machine-dependent optimizations. After code generation, the assembly code is assembled and linked to generate a machine executable that is simulated on a cycle accurate simulator for performance and leakage energy evaluation. In this work, we use the GNU Compiler Collection (GCC

4.2.1) [54], GNU Binary Utilities (Binutils 2.17) [59], and GNU C Library (Glibc 2.3.6) [60] to build the compiler toolchain for the ARM Linux platform. The Simplescalar ARM distribution [55] is used to perform cycle-accurate simulation for performance and power calculations.

The internal compiler pipeline in GCC [61] is shown in Figure 4.8. GCC uses three intermediate representations - GENERIC, GIMPLE, and RTL. The compiler front end parses the C source code and converts it into GENERIC representation. This is lowered into GIMPLE representation, where all tree-based code transformations are performed. Lower level optimizations, including all or most of the machine-dependent optimizations (instruction scheduling, peephole optimizations etc.), are performed at the RTL level. The control flow graph (CFG) information for a procedure is retained till late in the RTL passes. Since insertion of power gating instructions into the code requires its control flow information, the pass to do so is added right before the CFG is purged and the procedure is described only as an instruction list.

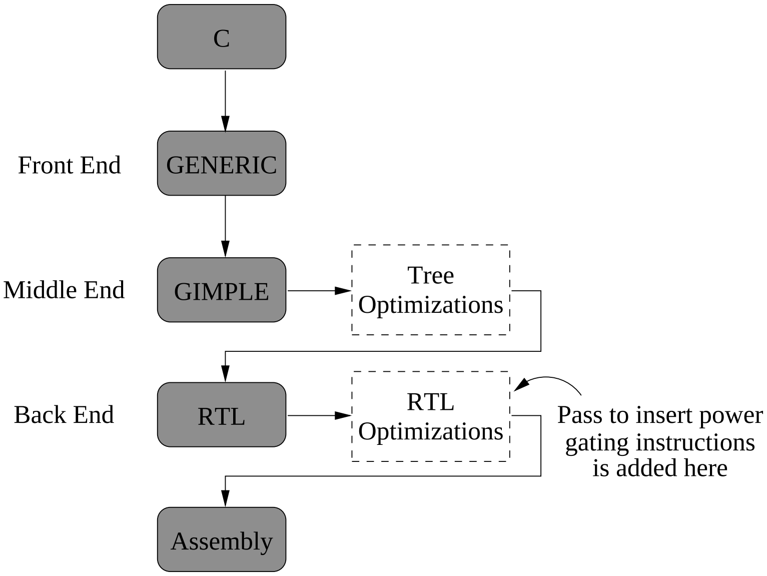


Figure 4.8 GCC Compiler Pipeline [62]. The pass to insert power gating instructions is added as an RTL optimization pass.

4.3.1 Architecture Support for Power Gating

The architecture support for power gating is based on the ARM processor architecture and is proposed in [37]. The instruction set architecture (ISA) provides an explicit *sleep* instruction, whose argument is an immediate integer that encodes the list of functional units that are to be deactivated. The ISA, however, does not provide any explicit *wakeup* instructions. The activation of the functional units is carried out automatically by the decode logic of the ARM pipeline for those functional units that are needed by the decoded instruction. The processor core has an integer ALU, which cannot be power gated; and a barrel shifter, an integer multiplier (IMUL), a floating point adder (FADD), a floating point multiplier (FMUL), and a floating point divide and square root unit (FDSQ), all of which can be power gated. The library of functional units with power gating support are designed for 1 cycle wakeup latency for a clock period of 10 ns (100 MHz clock) and are characterized for latency and power [37]. The barrel shifter, however, is not equipped with any power gating support in this work because of the frequent usage of shift instructions in the code. Shift instructions are generated during strength reduction of multiply operations with constant operands [51] and during generation of load and store instructions with complex addressing modes [54].

As shown in Figure 4.9, a sleep control register (SCR), comprising of SR flip-flops, is added to the decode logic of the ARM core. The outputs of these flip-flops drive the gates of the sleep transistors of the functional units. Since NMOS transistors are used for the sleep transistors, a `'0'` at their gate switches them OFF while a `'1'` switches them ON. In Figure 4.9, the contents of the SCR indicate that while IMUL and FDSQ are in active mode, FADD and FMUL are in sleep mode. When a sleep instruction is decoded, signal *slp* is asserted and bits 11-8 of the instruction are used to reset the flip-flops. When an instruction, requiring a particular functional unit, is decoded, a *wakeup* signal (`wkp_*`) is asserted to set the flip-flop. The wakeup signals are datapath flags that are generated by the decode logic to latch the operands in the input latches of the functional units.

The *rst* input to each flip-flop is gated with the output of that flip-flop to prevent unnecessary switching at the output of the corresponding AND gate. Therefore, the switching at the outputs of the AND gates and the SR flip-flops carry out the switching ON or OFF of the functional units. If

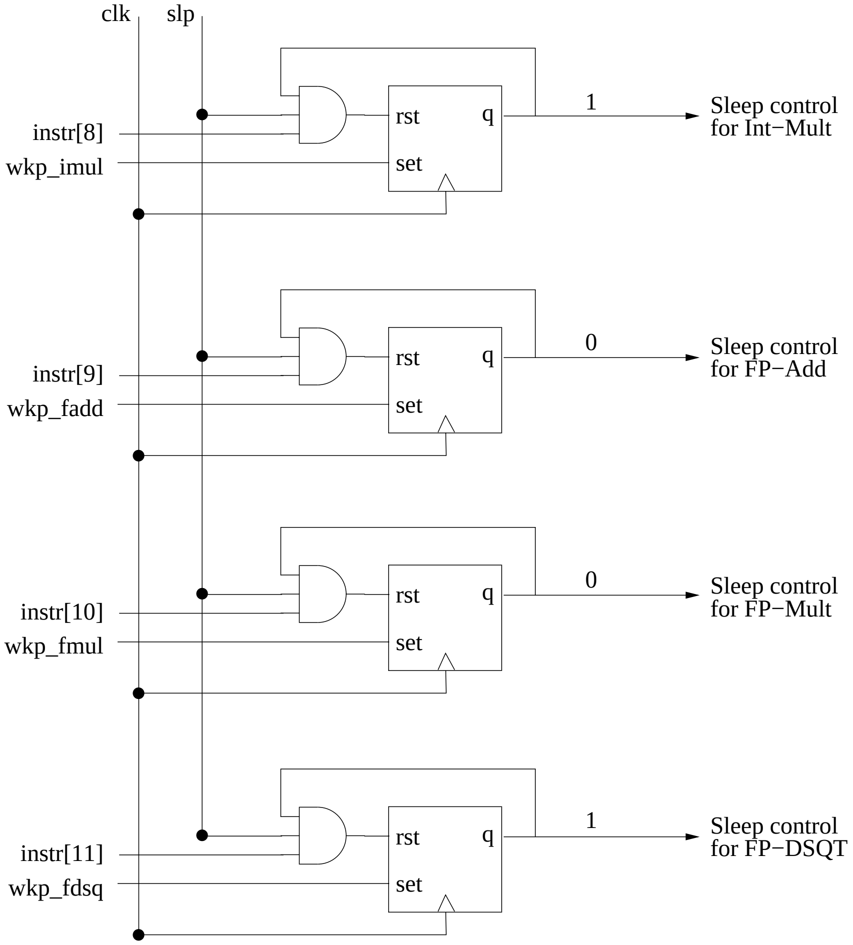


Figure 4.9 Gate level schematic of the Sleep Control Register (SCR). SCR is the extension to the ARM decode logic that enables the support to power gate functional units.

the output of an AND gate switches, it will switch the content of the corresponding flip-flop from `1′ to `0′. The functional unit whose sleep transistors the AND gate is controlling is, thereby, put to sleep. Similarly, if a wakeup (wkp_*) signal is asserted and the output of the corresponding flip-flop switches from `0′ to `1′, the functional unit whose sleep transistors it is controlling is woken up from sleep. Thus, the dynamic power overhead involved in the implementation of the SCR logic is linearly proportional to the number of times the functional units are switched ON and OFF. However, this overhead power is negligible when compared to the power overhead involved in activation and deactivation of the functional units.

Assembly format of sleep instruction

```
slp <Arg> /* 4 bit argument */
```

Arg bit 0 : Int−Mult
Arg bit 1 : FP−Add
Arg bit 2 : FP−Mult
Arg bit 3 : FP−DSQT

Machine code format of sleep instruction

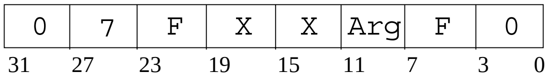


Figure 4.10 Assembly and machine code formats of the sleep instruction

The assembler support for translating the sleep instructions into machine code is added to the GNU ARM assembler, which is part of the Binutils package. The format of the machine code for the sleep instruction is chosen from the domain of exceptional opcodes described in the ARM reference manual and is shown in Figure 4.10. The assembly opcode for the sleep instruction is `slp`. It takes a four-bit immediate integer as an argument that encodes the list of the functional units that are to be deactivated. As shown in the figure, bits 0-3 of the immediate integer argument are for deactivating IMUL, FADD, FMUL, and FDSQ, respectively. The machine code for the `slp` instruction has bits 7-0 as `'F0'` and bits 31-20 as `'07F'`. Instruction bits 11-8 are reserved for the four-bit immediate integer argument that is passed to the sleep instruction. The definitions of the semantics of the `slp` instruction and the extensions to the semantics of the rest of the instructions (wakeup logic) are added to the machine description file. The machine definition for the Simplescalar ARM port maps each instruction to a functional unit in the processor core. This information is used to implement the wakeup logic. The ARM processor configuration used is that of the StrongArm processor core and is tabulated in Table 3.3.

4.3.2 Insertion of Sleep Instructions

The sleep instructions are inserted at discrete regions in a procedure. These regions are the entry block of the procedure itself and the basic blocks that are predecessors to the entry blocks of the loops in that procedure. Due to this reason, the pass to insert power gating instructions, first identifies the loops in the procedure, and then probes them for the usage of functional units. GCC provides a comprehensive library for representing loops and analyzing them. A *loop tree* is defined as a tree in which each node represents a loop in the procedure and the children of a node, say *L*, represent the loops immediately contained inside *L*. The loop tree captures the nesting structure of the loops within the procedure. The important attributes of a loop involved in performing the task of insertion of sleep instructions are *loop header*, *loop latch*, and *loop body*. Figure 4.11 illustrates these components. *Loop header* is the basic block that forms the entry block for the loop. *Loop latch* is a back edge that connects a basic block from inside the loop to the loop header. Finally, *loop body* is the set of basic blocks that are dominated by its header, and are reachable from its latch against the direction of edges in CFG. The shaded region in Figure 4.11 indicates the loop body. Natural loops, defined as the loops that have one loop header and possibly multiple loop latches, are the most commonly occurring loop structures in the C programming language and are, therefore, of interest to us.

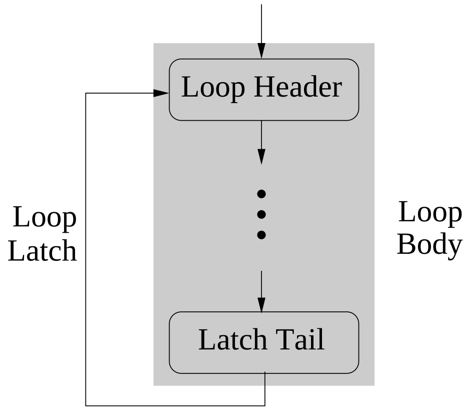


Figure 4.11 Components of a loop

Algorithm 4 toplevel-driver()

```
Construct the call graph
Perform interprocedural optimizations
for each procedure $x \in$ call graph in postorder sequence do
    expand ($x$)
end for
```

Algorithms 4-7 describe the technique of inserting power gating instructions during the compilation process. The top level procedure that drives the entire compilation process is described in Algorithm 4. First, the call graph is constructed after the source file is parsed. Then, interproce-

Algorithm 5 expand(f)

```
Perform intraprocedural optimizations on $f$
pgi-insert ($f$)
Output the assembly code for $f$
```

Algorithm 6 pgi-insert(f)

```
/* Gather functional usage information */
$FU(f) \leftarrow \Phi$
**for each** basic block $b \in$ CFG **do**
    $FU(b) \leftarrow \Phi$
    **for each** instruction $i \in b$ **do**
        **if** $i$ is a CALL instruction **then**
            $FU(b) \leftarrow$ usage (callee)
        **else**
            $FU(b) \leftarrow FU(i)$
        **end if**
    **end for**
    $FU(f) \leftarrow FU(f) \cup FU(b)$
**end for**
Compute the loop hierarchy tree, $L$, in $f$
**for each** loop $l \in L$ **do**
    $FU(l) \leftarrow \cup FU(b), \forall\, b \in l$
**end for**
/* Insert power gating instructions */
**for each** region $r \in L$ in preorder sequence **do**
    **if** $FU(r) \subset FU(parent(r))$ **then**
        /* $r$ needs fewer functional units than $parent(r)$ */
        Insert sleep($FU(parent(r) - FU(r))$) at the entry of $r$
    **end if**
**end for**
```

dural analysis and optimizations are performed across the procedures. Finally, the procedures are expanded in the postorder sequence of the vertices. This is done to enable transfer of data from a callee procedure to its caller, which may lead to better optimization opportunities during code generation of the caller procedure. From the perspective of the task of inserting power gating instructions, this ensures that a callee procedure is inspected for power gating opportunities before any of its callers. Algorithm 5 describes the expansion steps for a procedure, which involve performing the desired intraprocedural optimizations, inserting power gating instructions, and generating its assembly code. Algorithm 6 describes the technique of inserting sleep instructions within a procedure. The instructions in a basic block are inspected (lines 3-13) to compute the functional unit usage of that basic block. If an instruction is not a CALL instruction, then its functional unit usage is computed by parsing the instruction expression. Otherwise, the value returned by the procedure `usage()` (Algorithm 7) is used. The loop tree is computed (line 14) and the functional unit usage of all the loop bodies are determined (lines 15-17). Next, the loop tree is traversed (lines 18-24) in preorder sequence for regions whose entries can be gated with sleep instructions. The root of the loop tree represents the entire procedure.

Algorithm 7 usage(f)

```
/* lookup the call graph for $f$ */
if $f \in$ call graph then
    return $FU(f) - \cup FU(l), \forall$ loop $l \in f$
end if
/* $f$ is a standard library procedure */
return functional usage based on policy P1, P2, or P3
```

Algorithm 7 describes the routine that returns a set of functional units corresponding to the callee procedure symbol that is passed to it as an argument. If the callee procedure is part of the compilation unit then the call graph has an entry for it. In this case, the functional unit usage of the region of the code that is not part of any loops in the callee is returned. This is because any functional unit that is needed in this region will be needed as many times as the procedure is called. If the callee has loops, then those loops already have sleep instructions inserted at their entries which turn off functional units that are not required within their bodies. Finally, if it is determined that the callee procedure does not belong to the compilation unit, then it must be a C standard library call and the return value is determined by the policy that is used to handle the C standard library routines. This is described the next subsection. Although the technique described here does not use dynamic profiling information of the applications to direct the insertion of sleep instructions in the code, the techniques that do so ([33, 37]) can be readily incorporated into the compilation framework described in this work.

4.3.3 Policies for Handling C Standard Library Routines

The most frequently used standard library routines in the benchmarks from MediaBench and MiBench are those defined in `stdio.h`, `stdlib.h`, `string.h`, and `math.h`. Based on the Glibc implementation of the C standard library, while the routines defined in `math.h` use FP units commonly, those defined in the first three headers do not use any FP units (except the string to FP type conversion routines such as `atof()`, etc.). Moreover, a large number of the low level file I/O routines, memory allocation/deallocation routines, and string manipulation routines do not even use the integer multiplier. Considering this knowledge about the standard library routines, in this work, we apply three policies to handle them. These policies are as follows:

1. P1 - It is assumed that none of the standard library routines use any of the functional units.
2. P2 - It is assumed that the routines belonging to the math library use all the functional units, whereas the other routines do not use any functional units.
3. P3 - The routines defined in the math library are profiled for their functional unit usage and a lookup table is created with this information. This lookup table is referred to during the power gating stage. For the rest of the routines, it is assumed that they do not use any functional units.

4.3.4 Proposed Leakage Aware Compilation Flow

Based on the work and discussions in the preceding subsections, we propose a leakage aware compilation flow that is shown in Figure 4.12. The application source, which is comprised of one or more compilation units (set of procedures that the compiler generates code for in a single run), is the input to this flow. A call graph is constructed for each compilation unit, which is inspected

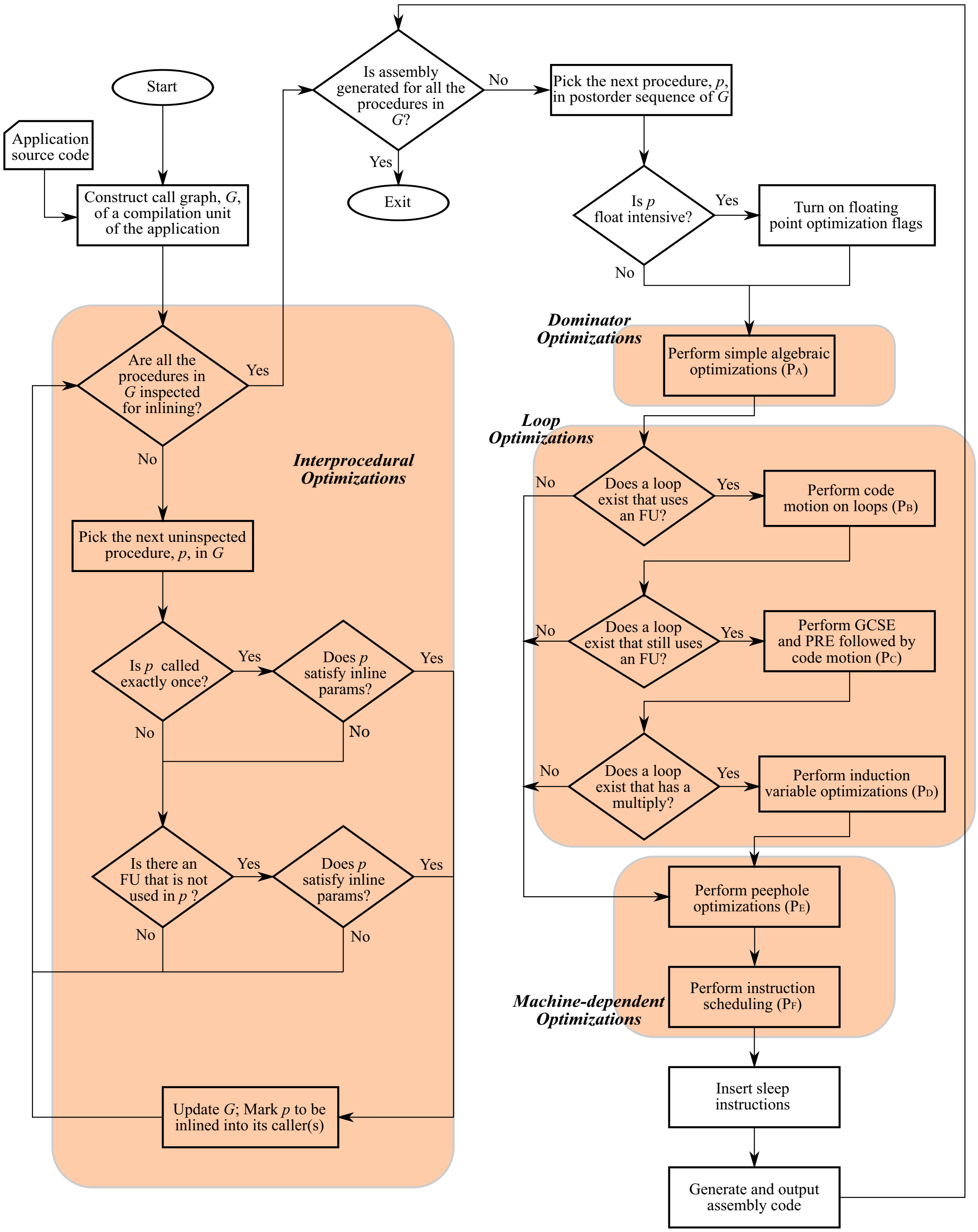

Start
Application source code
Construct call graph, G, of a compilation unit of the application
Is assembly generated for all the procedures in G?
No
Yes
Exit
Pick the next procedure, p, in postorder sequence of G
Is p float intensive?
Yes
No
Turn on floating point optimization flags
Dominator Optimizations
Perform simple algebraic optimizations (P_A)
Loop Optimizations
Does a loop exist that uses an FU?
Yes
No
Perform code motion on loops (P_B)
Does a loop exist that still uses an FU?
Yes
No
Perform GCSE and PRE followed by code motion (P_C)
Does a loop exist that has a multiply?
Yes
No
Perform induction variable optimizations (P_D)
Perform peephole optimizations (P_E)
Perform instruction scheduling (P_F)
Machine-dependent Optimizations
Insert sleep instructions
Generate and output assembly code
Are all the procedures in G inspected for inlining?
Yes
No
Interprocedural Optimizations
Pick the next uninspected procedure, p, in G
Is p called exactly once?
Yes
No
Does p satisfy inline params?
Yes
No
Is there an FU that is not used in p ?
Yes
No
Does p satisfy inline params?
Yes
No
Update G; Mark p to be inlined into its caller(s)

Figure 4.12 Proposed compilation flow to generate leakage optimized code with compiler-directed power gating. Each vertex in a call graph is a procedure, which is decribed as a control flow graph. Each vertex in a control flow graph is a basic block. The regions enclosed in grey rectangles correspond to the optimization phases shown in the framework in Figure 4.7. The labels for the intraprocedural optimization passes, P_{A-F}, are used to annotate the optimization configurations studied in the experimental section (Table 4.4).

for procedure inlining. The decision to inline a procedure primarily depends on its impact on the increase in code size. GCC has builtin heuristics to decide if a procedure should be inlined or not. The relevant parameters that control these heuristics and their default values are tabulated in Table 4.2. A procedure that is declared `static` and is called only once in the compilation unit is always considered for inlining because it is not likely to impact the code size. However, for the rest of the procedures, only those procedures that have at least one functional unit that can be power gated in its body are considered to be inlined. As mentioned earlier in Section 4.2.2, this has a significant potential to reduce the number of overhead sleep instructions during program execution. Although not explicitly shown in the figure, interprocedural constant propagation is also performed during this stage.

Table 4.2 Procedure inlining parameters. [54]

Parameter	Description	Value
max-inline-insns-auto	Size of procedures considered for inlining in number of instructions (counted in GCC's internal representation)	90
large-function-insns	Limit specifying large procedures in number of instructions (counted in GCC's internal representation)	2700
large-function-growth	Relative growth of large procedures after inlining with respect to original size	200%
inline-call-cost	Relative cost of a call instruction compared to a simple arithmetic instruction (e.g. add)	16

Once all the procedures are inspected for procedural inlining, they are picked in a postorder sequence of the call graph (explained earlier in Section 4.3.2) for intraprocedural optimizations and assembly code generation. The decision to perform floating point optimizations is taken based on the knowledge of the applications. If an application has floating point operations, then the IEEE or ISO floating point precision rules are relaxed so that arithmetic optimizations can be performed on floating point data types as well. Simple dominator-based arithmetic optimizations like *dead code elimination, constant and copy propagation, local subexpression elimination* and *full redundancy elimination* are performed. Following this, if there are loops in which one of more functional units are being used, then code motion is performed on the loop bodies. As mentioned earlier in Section

4.2.1.1, Global subexpression elimination (GCSE) and partial redundancy elimination (PRE) are techniques that always may not result in performance optimized code because the former increases register pressure, while the latter has the potential to increase code size. However, these optimizations may aid code motion in removing requirements of functional units out of loops (illustrated in Section 4.2.1) and, therefore, are evaluated if code motion alone is not entirely effective in removing computations out of the loops. Induction variable optimizations are performed further if there are any multiply operations in the loops (for both integer and floating point types). Finally, the backend optimizations, peephole transformations, and instruction scheduling are performed before inserting sleep instructions and finally generating assembly code.

4.4 Experimental Setup and Results

For the purpose of experimentation, we report results for the following benchmarks from Mibench and Mediabench suites:

- *Susan* (Smallest Univalue Segment Assimilating Nucleas) is a set image processing benchmarks from the automotive category in MiBench. It comprises of three programs - edge detection *SusanE*, corner detection *SusanC*, and image smoothing *SusanS*.
- *Epwic* (Embedded Predictive Wavelet Image Color) is part of MediaBench which implements a wavelet image encoder (*Epwic*) and a decoder (*Unepwic*).
- *Mpeg2* (Moving Pictures Experts Group) benchmarks comprise of *Mpeg2E* and *Mpeg2D*, which are video encoding and decoding programs, respectively. These benchmarks are also part of the Mediabench suite.

All the benchmarks described above, except *SusanS* are floating point benchmarks, with *Mpeg2D* having the least composition of floating point operations (less than 0.5% in C0 configuration). Table 4.3 tabulates, for each benchmark built in C0, the number of procedures and the number of simulation cycles reported by Simplescalar-ARM for the StrongARM configuration. All the experiments were performed on a Linux server with four 2.6 GHz AMD processors and 16 GB

Table 4.3 Benchmark details. (in C0 configuration)

Benchmark	Number of procedures	Number of cycles (Millions)
SusanC	19	4.05
SusanE	19	8.13
Epwic	130	1745.60
Unepwic	130	179.69
Mpeg2E	95	549.02
Mpeg2D	114	28.89

The configurations are described in Table 4.4

RAM running Red Hat Enterprise Linux AS release 4. The code modifications to GCC, including the pass to insert sleep instructions, resulted in less than 1300 additional lines in the source code. The simulation time of the benchmarks with Simplescalar ranged from a few seconds, for *SusanC* benchmark, to several minutes for *Epwic* benchmark.

4.4.1 Optimization Configurations

Table 4.4 Optimization configurations

Optimization Label	Description	Pass Labels wrt Figure 4.12
C0	Base configuration (only machine specific instruction scheduling is performed)	P_F
C1	C0 + Machine dependent peephole optimizations	P_{E-F}
C2	C1 + All dominator optimizations (dead code elimination, copy and constant propagation, full and partial redundancy elimination, local and global common subexpression elimination)	P_A, P_{E-F}
C3	C1 + Simple dominator optimizations (except GCSE and PRE) + Loop invariant code motion	P_{A-B}, P_{E-F}
C4	C3 + All dominator optimizations + Loop invariant code motion	P_{A-C}, P_{E-F}
C5	C4 + Induction variable optimizations	P_{A-F}

The optimizations in each of the configurations above may be performed with interprocedural and floating point optimizations

Table 4.4 describes the various optimization configurations defined for the purpose of experimentation in this work. The third column of this table lists the labels of the intraprocedural optimization passes from Figure 4.12 for each of these configurations. C0 is the base configuration in which only machine specific instruction scheduling is performed. In C1, machine dependent peephole optimizations are also enabled. In C2, all dominator optimizations are performed along with those in C1. In C3, simple dominator optimizations (excluding GCSE and PRE) and loop invariant code motion are performed along with those in C1. More aggressive dominator optimizations, including GCSE and PRE, are added to C3 to define C4. In C5, induction variable optimizations are performed in addition to the ones performed in C4. All the intraprocedural optimizations are performed after interprocedural optimizations are performed except for the *Susan* benchmarks. The floating point arithmetic optimization flag `-ffast-math` is turned on for all benchmarks except *SusanE* and *Mpeg2D*.

4.4.2 Results

The library of functional units developed in [37] is used for the leakage energy calculations for all the benchmarks. The leakage savings for the units in sleep mode is about 50%, which indicates

Table 4.5 Description of metrics

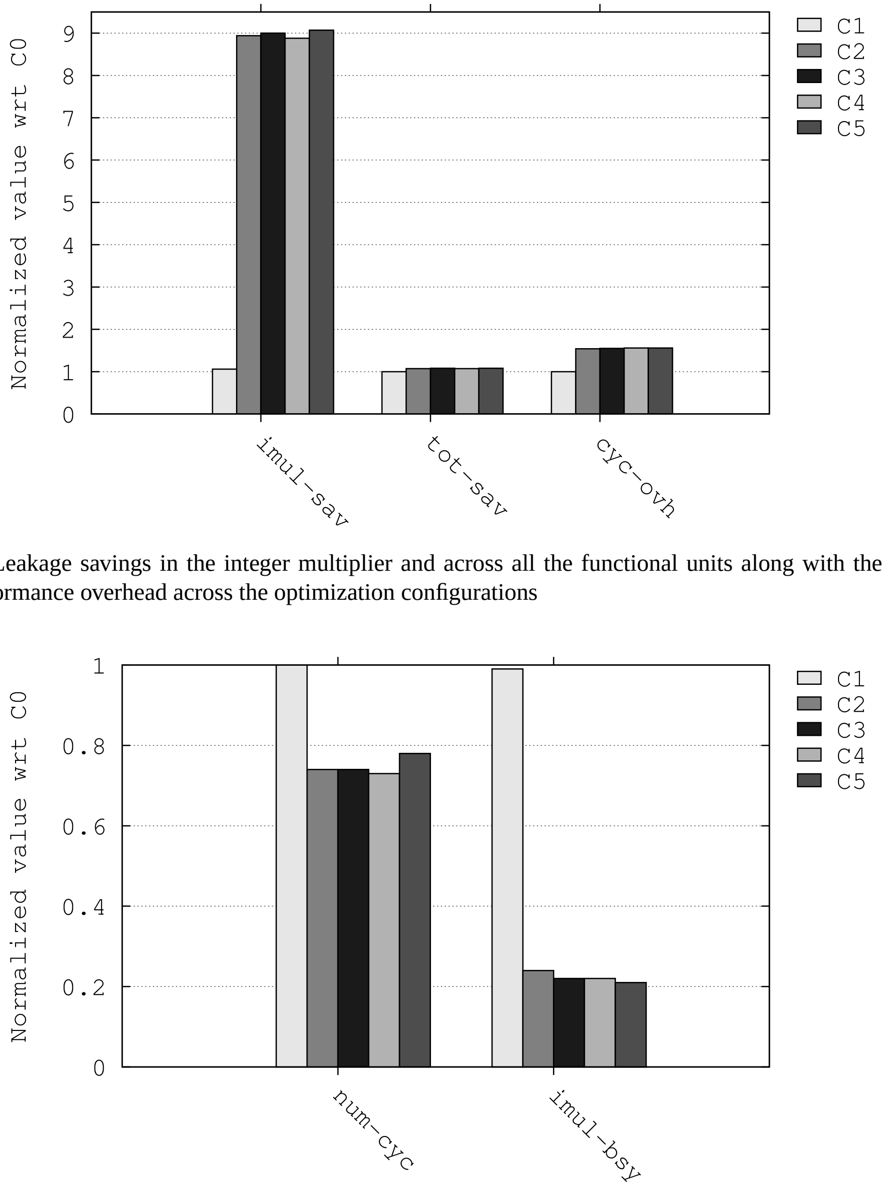
Metric label	Description
num-cyc-nopg	Number of simulation cycles without any sleep instructions
num-cyc-pg	Number of simulation cycles with sleep instructions
cyc-ovh	Performance penalty incurred in executing the sleep instructions (in percentage), computed as (num-cyc-nopg/num-cyc-pg) − 1
unit-bsy	Number of cycles for which *unit* is busy
unit-engy-nopg	leakage energy for *unit* without power gating
unit-engy-pg	leakage energy for *unit* with power gating
unit-sav	Percentage savings in leakage energy in *unit* due to power gating, computed as 1 − (*unit*-engy-pg/*unit*-engy-nopg)
total-sav	Percentage savings in total leakage energy across all units due to power gating, computed as $1 - \left(\sum_{\forall \text{units}} \textit{unit}\text{-engy-pg} / \sum_{\forall \text{units}} \textit{unit}\text{-engy-nopg}\right)$

that the theoretical upper bound on the maximum achievable savings is 50%. Table 4.5 enumerates the metrics that are computed and reported in this section to demonstrate the effectiveness of power gating. All the results described next are reported for policy P3 used to handle the C standard library routines. Also, the legend ‘num-cyc’ in all the plots indicates ‘num-cyc-nopg’, as described in Table 4.5. The FP optimizations are also turned on for all the benchmarks in configurations C1-C5.

4.4.2.1 Susan Benchmarks

Figures 4.13(a) and 4.13(b) show the statistics for *SusanE*. For this benchmark, the integer multiplier is again the most frequently used unit (busy for 17.3% of the cycles in C0), whereas the FP units are busy for roughly 1% of the cycles. Again, in C2-C5, large leakage savings are observed for the integer multiplier when compared to those in C0. In these configurations, the savings increase by 9X (from 3.3% to almost 30%). This can be attributed to the fact that the number of busy cycles for this unit reduces by 4X (0.24 vs. 1.00) in C2 and by almost 5X (0.21 vs. 1.00) in C5. In C5, strength reduction on induction variables is performed which is very effective in replacing multiply operations with equivalent add or subtract operations. The savings in the FP units are almost uniform across all the configurations. The total leakage savings across all the functional units increases by less than 10% because the leakage component of the integer multiplier is a small fraction of those of the FP units.

For *SusanC*, the integer multiplier is busy 14% of the cycles, whereas the FP units are used in less than 2% of the cycles in C0. In the rest of the configurations (all built with floating point optimizations), significant improvements in the leakage savings are observed for the integer multiplier and the FP divider. Figures 4.14(a) and 4.14(b) show the statistics for *SusanC*. In C2-C5, the number of busy cycles for the integer multiplier drops by 8X (0.12 vs. 1.00) to 12X (0.08 vs. 1.00) giving an increase in savings by 7% to 2.1X. For the FP divider, the leakage savings improve by more than 2.2X in configurations C3-C5. This is because the number of busy cycles for this unit drops by almost 6X (0.17 vs. 1.00). The savings in the rest of the units are uniform over all the optimization configurations. The total leakage savings (across all the functional units) is 1.7X than

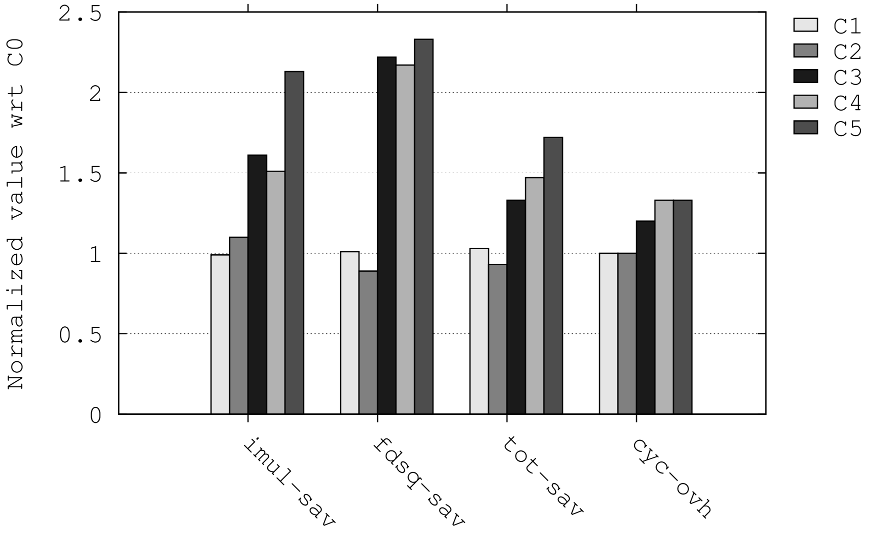


(a) Leakage savings in the integer multiplier and across all the functional units along with the performance overhead across the optimization configurations

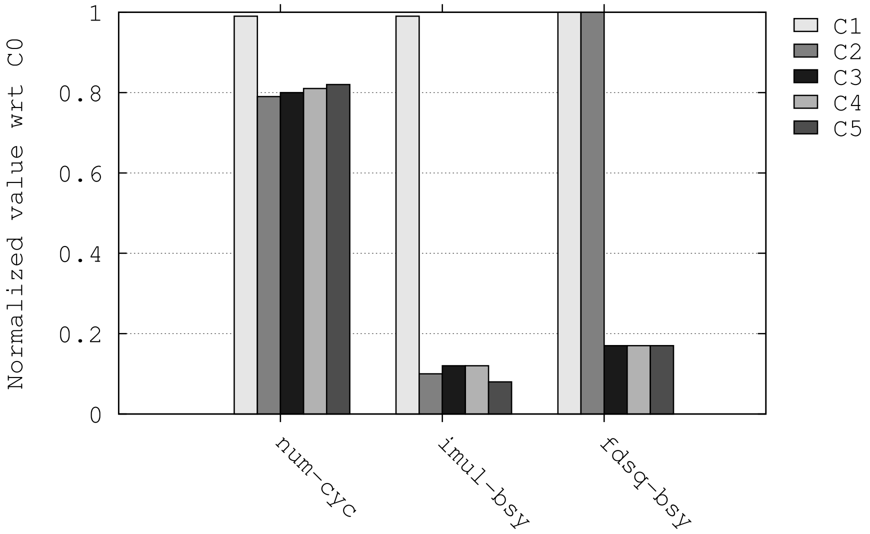
(b) Total number of execution cycles (without sleep instructions) and the number of busy cycles for the integer multiplier across the optimization configurations

Figure 4.13 Leakage savings and sleep overhead for *SusanE*

that in C0. The overhead in terms of the number of extra cycles executed increases by 1.3X in C5 but is below 0.6% of the total numer of execution cycles. The number of execution cycles for this benchmark reduces by 20% in C2-C5 wrt that in C0. Figure 4.15 illustrates the impact of weak strength reduction on the leakage savings achieved for the integer multiplier across all the optimiza-



(a) Leakage savings in the integer multiplier, FP divider, and across all the functional units along with the performance overhead across the optimization configurations



(b) Total number of execution cycles (without sleep instructions), the number of busy cycles for the integer multiplier and the FP divider across all the optimization configurations

Figure 4.14 Leakage savings and sleep overhead for *SusanC*

tion configurations. For C5, the savings increase by almost 2X when weak strength reduction is performed.

SusanS is an integer benchmark that spends almost the entire time in a loop performing integer multiplications on memory elements. Therefore, during the entire time, the integer multiplier is

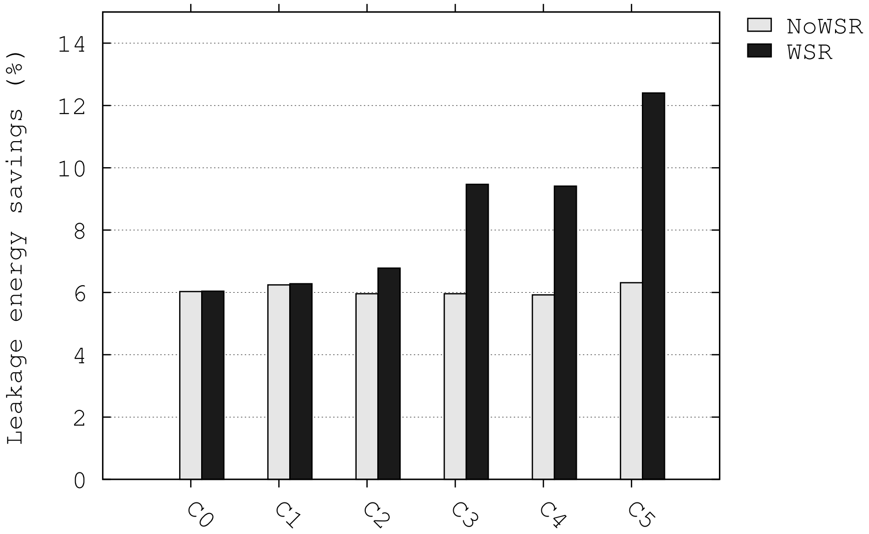
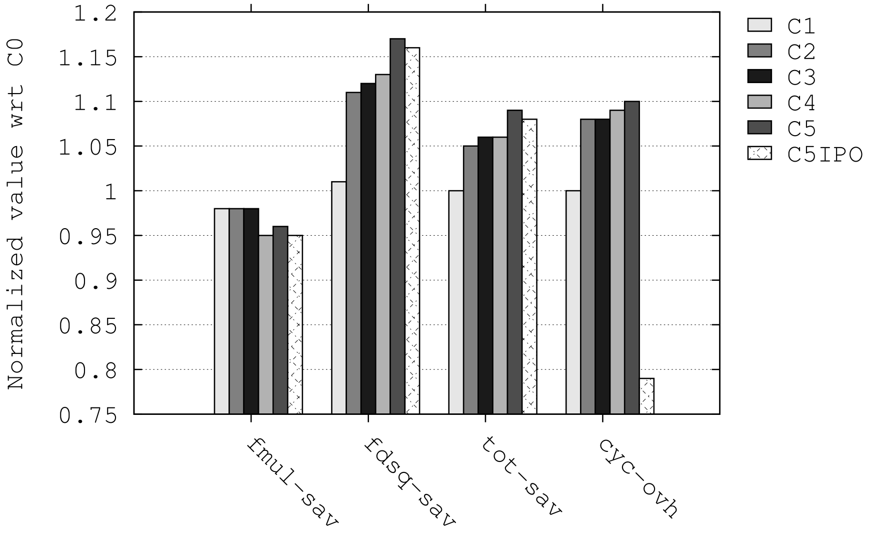


Figure 4.15 Impact of weak strength reduction on leakage savings of the integer multiplier for *SusanC*. NoWSR indicates that no weak strength reduction is performed.

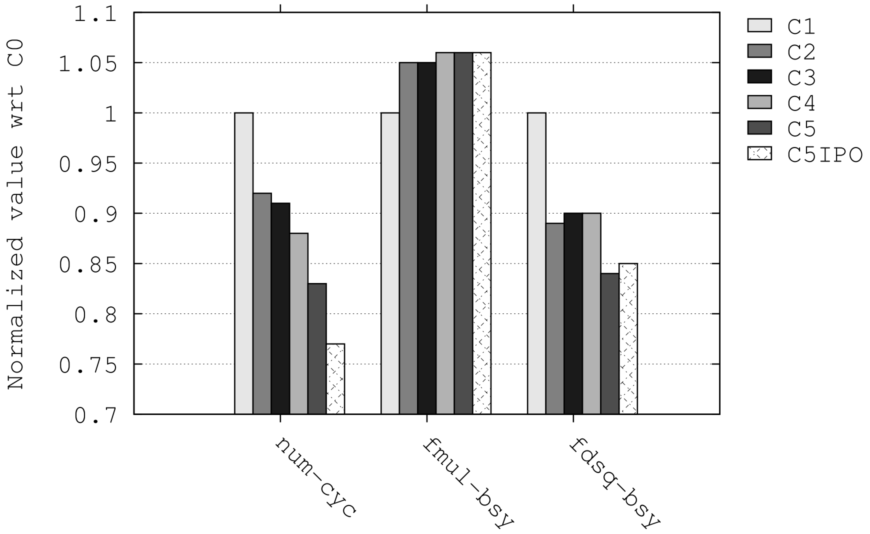
awake, whereas the floating point units are asleep. Interprocedural optimizations are not effective on any of the benchmarks from the *Susan* set owing to the non-modular description of the source code. Therefore, interprocedural optimizations are not performed on them.

4.4.2.2 Epwic Benchmarks

Figure 4.16 shows the plots for *Epwic* benchmark. In this benchmark, the FP divider and FP multiplier tradeoff leakage savings with each other as the various optimizations are performed. The number of busy cycles for the FP divider drops by 10%-16% in C2-C5, while that for the FP multiplier increases by 5%-6%. This gets reflected in the leakage savings for these two units. While the savings in the FP divider increases by 16% (28% vs. 32.5%) in C5, the savings for the FP multiplier decreases by 5% (30% vs. 28.5%). Since the leakage in FP divider is almost twice as much as that in the FP multiplier, increased savings in FP divider contribute towards higher total savings across all the units (which increases by 5%-9%). The performance improves steadily as we go from C1-C5IPO (C5IPO is C5 configuration with interprocedural optimizations). The cycle overhead in C5IPO reduces by more than 20% compared to that in C0. Since *Unepwic* shares



(a) Leakage savings in the FP multiplier, FP divider, and across all the functional units along with the performance overhead for various optimization configurations. C5IPO indicates that interprocedural optimizations are performed in C5

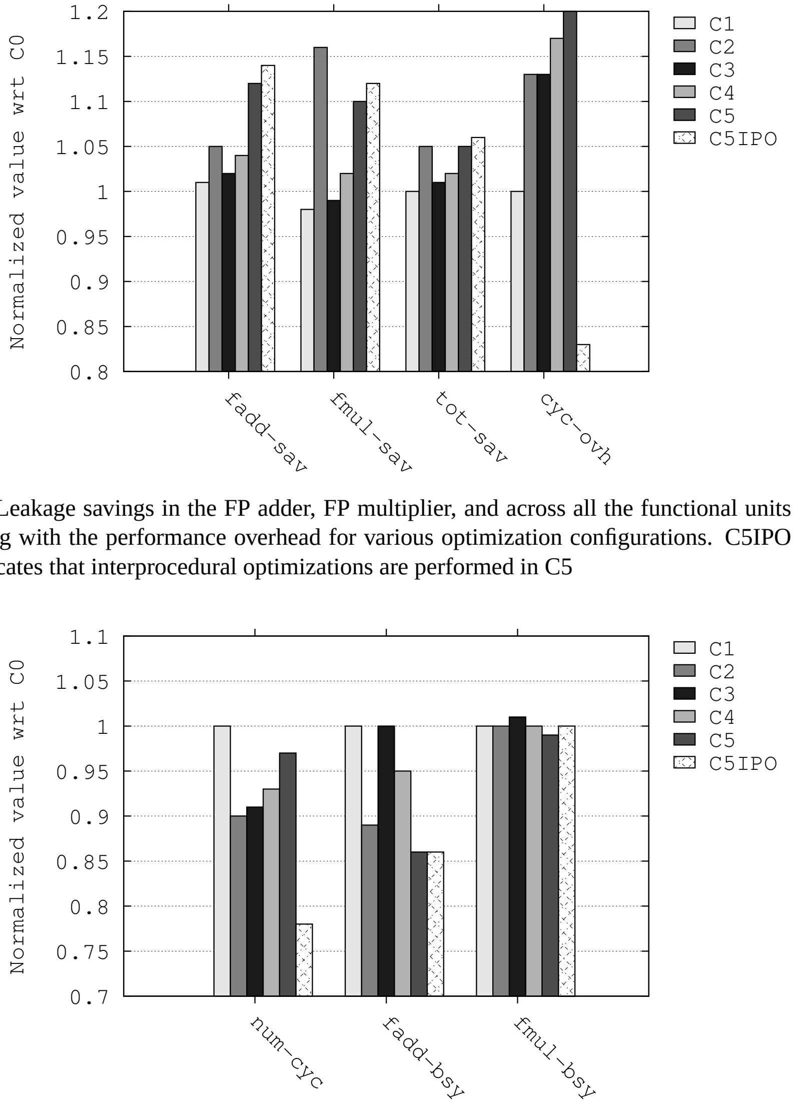


(b) Total number of execution cycles (without sleep instructions) and the number of busy cycles for the FP multiplier and FP divider for the optimization configurations

Figure 4.16 Leakage savings and sleep overhead for *Epwic*

majority of the source files with *Epwic*, similiar tradeoff is observed between the FP divider and the FP multiplier as in *Epwic*. However, the leakage savings in the FP adder and the FP multiplier are much lower (8% and 12%) than those in *Epwic*. Therefore, the total savings in *Unepwic* is lower when compared to those in *Epwic*.

4.4.2.3 Mpeg2 Benchmarks



(a) Leakage savings in the FP adder, FP multiplier, and across all the functional units along with the performance overhead for various optimization configurations. C5IPO indicates that interprocedural optimizations are performed in C5

(b) Total number of execution cycles (without sleep instructions) and the number of busy cycles for the FP adder and FP multiplier for the optimization configurations

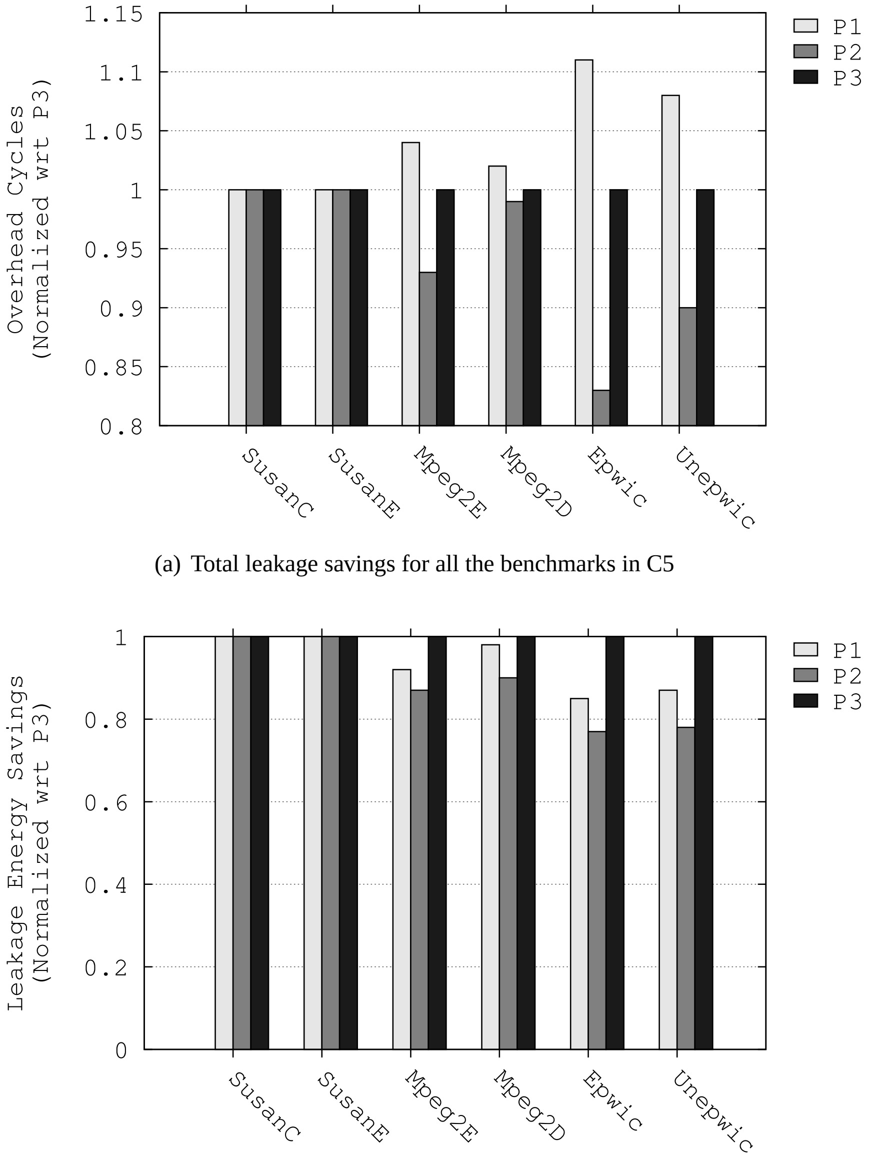
Figure 4.17 Leakage savings and sleep overhead for *Mpeg2Encode*

In *Mpeg2E*, the FP multiplier is the most frequently used unit (busy for 4.4% of the total cycles in C0), whereas the FP divider is the most infrequently used unit (busy for less than 0.01% in C0). The integer multiplier and FP adder are busy for 3.3% and 2.6% of the cycles in C0. When interprocedural optimizations are performed in C5 (CFIPO), the leakage savings in the FP adder increases by almost 14% (36% compared to 41%). This is a significant improvement in savings considering the fact that the maximum achievable savings is less than 50%. The number of cycles for which the FP adder is busy drops by 14% in C5 and C5IPO. Also, in C5IPO, the total simulation cycles for this benchmark reduces significantly (by more than 20%) and the number of overhead cycles due to power gating also drops by more than 15%. For the FP multiplier, 16% increase in leakage savings are observed in C2 (44% compared to 38%). Across all the optimization configurations, the leakage savings in the integer multiplier and the FP divider are almost uniform (31% and almost 50%, respectively). This is shown in the plots in Figure 4.17.

In *Mpeg2D*, the functional units other than the integer ALU are used very infrequently. Due to this the leakage savings in all the units are almost uniform (almost 50%) for all the optimization configurations.

4.4.3 Impact of Policies for Handling Standard Library Routines

The impact of the three policies that are used to handle the C standard library routines on total leakage savings and overhead cycles in C5 (with interprocedural and FP optimizations) is shown in Figure 4.18. These values are normalized with respect to those in P3. For the *Susan* benchmarks, there is no difference in either the leakage savings or the overhead cycles across the different policies because the math routines are not used in the most frequently executed regions. However, for the *Epwic* and *Mpeg2* benchmarks, both the leakage savings and the overhead cycles reduce in policy P2 with respect to those in P3 because none of the functional units are put to sleep in the regions where the math library routines are called. The leakage savings for P1 are higher than those in P2 but lower than those in P3 because the functional units that are put to sleep right before a math routine are woken up when that routine is executing before the unit stays asleep for less than breakeven number of cycles. This results in lower leakage savings in those units. The most often used math routines in these benchmarks are `exp()`, `log()`, and `sqrt()` all of which do not use the FP divider unit, which has the largest component of leakage (more than 45%) among all the units. Therefore, the leakage savings in P1 are more than in P2.



(a) Total leakage savings for all the benchmarks in C5

(b) The overhead cycles for all the benchmarks in C5

Figure 4.18 Impact of various policies for handling the standard library routines during insertion of sleep instructions

Integer benchmarks like *dijkstra*, *patricia*, *quicksort*, and *sha* do not have any integer multiply operations. Therefore, when peephole optimizations (particularly, weak strength reduction) are performed, all overhead multiply operations (introduced by the compiler to compute array addresses) are eliminated. The benchmarks - *Rsynth*, *Fft*, and *Ffti* - spend almost the entire time performing arithmetic operations on memory elements and, therefore, none of the compiler optimizations are able to eliminate such operations.

4.5 Conclusions

In this work, we studied the impact of various compiler optimizations on the opportunities for power gating various functional units in an embedded processor core. To perform this study we use the ARM Linux toolchain comprising of the GNU compiler collection (GCC), which is a an open source production quality compiler, along with the Simplescalar-ARM processor simulator. We build a set of benchmarks for embedded systems chosen from Mibench and MediaBench suites in various predefined optimization configurations and perform extensive simulations with the Simplescalar-ARM tool to evaluate the effect of compiler optimizations on power gating for functional units. The results of the experiments indicate that the optimizations that remove computation redundancy from the program or assist such optimizations are instrumental in improving power gating opportunities in a program. In quantitative terms, depending on the benchmark, the leakage energy savings due to power gating in individual functional units can be increased by 15% to 9 times through the use of different compiler optimization techniques. The penalty of executing the sleep instructions is at most 0.1% for all benchmarks, except for *SusanC*, for which the penalty is at most 0.6%. Therefore, the energy overhead involved in executing the sleep instructions is very minimal compared to the leakage savings achieved in the functional units. With shrinking technologies and ever increasing demand of portable devices, reducing the total energy consumption of processors will continue to be be one of the primary goals for processor designers and compiler writers. This work thoroughly investigates reducing the energy dissipated by the processors due to leakage power at the compiler level with the help of architectural support. Since the leakage component in CMOS circuits is increasing every technology generation, it is imperative that compilers will need to generate code to not only improve performance but also reduce energy consumption in microprocessors. The work presented in this paper serves as an important step towards that requirement.

CHAPTER 5

STATE-RETENTIVE POWER GATING OF REGISTER FILES IN MULTI-CORES

We investigate state-retentive power gating of register files for leakage reduction in multi-core processors supporting multithreading in this chapter. In an in-order core, when a thread gets blocked due to a memory stall, the corresponding register file can be placed in a low leakage state through power gating for leakage reduction. When the memory stall gets resolved, the register file is activated for being accessed again. Since the contents of the register file are not lost and restored on wakeup, this is referred to as state-retentive power gating of register files. While state-retentive power gating in single cores has been studied in the literature, it is being investigated for multi-core architectures for the first time in this work. We propose specific techniques to implement state-retentive power gating for three different multi-core processor configurations based on the multithreading model: (i) coarse-grained multithreading, (ii) fine-grained multithreading, and (iii) simultaneous multithreading. The proposed techniques can be implemented as design extensions within the control units of the in-order cores. Each technique uses two different modes of leakage states: *low leakage savings and low wake-up latency* and *high leakage savings and high wake-up latency*. The overhead due to wake-up latency is completely avoided in two techniques while it is hidden for most part in the third approach, either by overlapping the wake-up process with the thread context switching latency or by executing instructions from other threads ready for execution. The proposed techniques were evaluated through simulations with multiprogrammed workloads comprised of SPEC 2000 integer benchmarks. Experimental results show that in an 8-core processor executing 64 threads, the average leakage savings were 42% in coarse-grained multithreading, while they were between 7% and 8% for fine-grained and simultaneous multithreading.

5.1 Motivation

The motivation for our work arises from the simple observation that, in an in-order CPU, when an instruction from a particular thread encounters a pipeline stall, no further instructions from that thread (i.e., following instructions in program order) may be executed till that stall gets resolved. Thus, the thread gets blocked and the hardware units that are private to that thread could be placed in a low-leakage state. When the pipeline stall is resolved, the thread gets ready to run and those hardware units need to be brought back to the active state from the low-leakage state so that they are functional again.

As discussed earlier, the datapath components that are replicated to support hardware multithreading are the register files and buffer structures such as, instruction fetch buffers, load-store buffers, and, in some cases, pipeline registers. Among these, the register files are the largest in area and at the same time are the leakiest. For example, the SPARC architecture uses windowed integer register files with eight windows [8]. In the Niagara processor, each thread requires 128 registers for the eight windows (with 16 registers per window) and another 32 global registers, which makes a total of 160 registers per thread. Since, each SPARC core supports four hardware threads, there are a total of 640 registers in each SPARC core. Each register is 64 bits in size and there are additional bits for implementing error correcting codes (ECC). This makes each integer register file in the Niagara processor a 5 KB storage structure. If this is compared with the L1 data cache, which is private to each core in the Niagara processor and is 8 KB in size, the register file has more than 60% of the storage size of the L1 data cache. Thus, placing parts of the register files in a low-leakage state during pipeline stalls appears to be a very attractive option for saving over all leakage energy in a processor core.

Another observation is that when multiple CPU cores are required to be accommodated to build a multi-core processor, the caches that are private to each core are shrunk in size to fit the chip within a given area limit. This results in an increase in the cache miss rates experienced by each core. For instance, as mentioned earlier, each core in the Niagara processor features an 8 KB private L1 data cache which results in average miss rates of around 10% [8]. However, having four threads to run on a single core hides the latencies in stalls due to access misses from the L1 and L2 caches very effectively. Thus, as the number of cores and the number of threads per core are increased, the fraction of time for which the integer register files for each thread stays idle due to memory stalls also increases.

It is important that any approach to reduce leakage based on the stalls incurred by hardware threads meet two important requirements:

1. The leakage reduction technique that is chosen to put the register file to a low-leakage state should be state retentive so that, when it is brought back to the active state, its contents are restored.

2. Every dynamic leakage reduction technique has a performance overhead when transitioning between the active and the low-leakage states. Thus, it is important to ensure that the overhead does not negatively impact the overall performance of the processor.

In this work, we apply state-retentive power gating to save leakage in integer register files during memory stalls in multithreaded processor cores. Figure 5.1 illustrates the schematic view of this approach for a core which supports execution of four hardware threads. The fundamental idea is that when a memory stall (cache miss) is detected, the running thread either gets blocked or gets

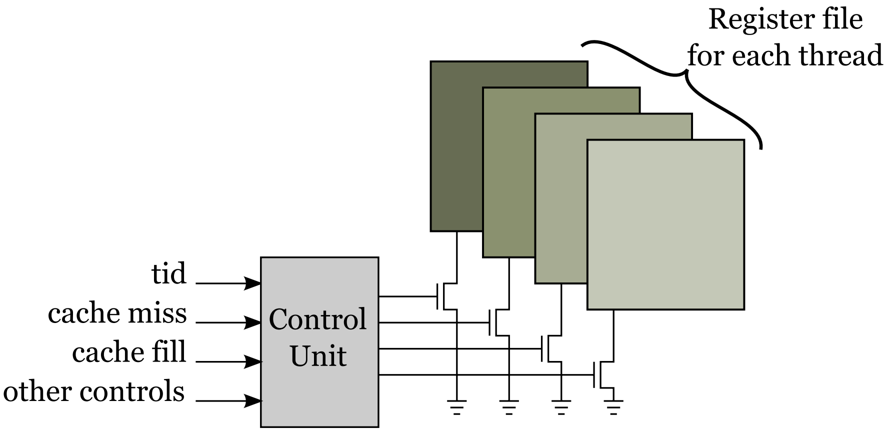


Figure 5.1 Schematic view of the proposed approach for power gating register files in in-order cores that support multithreading. When a thread (specified by *tid*) experiences a cache miss, its register file is placed in a low-leakage state. When the thread gets ready to run again after the cache miss completes, the register file is activated. The existing control unit in the core may be extended to incorporate this scheme very easily.

switched out and, therefore, its register file is placed in a low-leakage state. Eventually, when the stall gets resolved (following a cache line fill) the register file is put back in the active state.

An intermediate strength power gating technique presented in [63] is applied to characterize 32 entry 64-bit integer register file for leakage savings. The technique is also fine-grained in that the wake-up latencies are between three and five clock cycles (for a clock frequency of 1 GHz). We ensure that wake-up latency associated with this technique is effectively hidden by virtue of more than one thread sharing the CPU pipeline. Further, the register files have two distinct low-leakage states - one with lower leakage savings and lower wake-up latency; and the other with higher leakage savings and higher wake-up latency. Depending on the duration of the stall and the time between when the stall gets resolved and the register file is accessed again, it is placed is one of those low-leakage states. This is elaborated in Figure 5.2. The register file is designed to have two low-leakage states, *sleep1* and *sleep2*. When an L1 miss is incurred by an instruction from a particular thread, that thread's register file is placed in *sleep1* state. If the L1 fill request further experiences an L2 miss, then the register file is placed in the higher leakage savings state, *sleep2* state. When the L2 miss completes, the register file is brought back to the *sleep1* state. The wake-up latency, t_{w2}, is overlapped with the L1 fill latency and, thus, gets hidden. If, however, an L2 miss is not experienced, it continues to stay in *sleep1* state. When the L1 miss completes, the register file is brought back to the active state. The wake-up latency, t_{w1}, is hidden very effectively due to multiple threads running on the core.

The main contributions of this work are as follows:

- We propose specific techniques to implement state-retentive power gating for three different multi-core processor configurations based on the multithreading model: (i) coarse-grained multithreading (CGMT), (ii) fine-grained multithreading (FGMT), and (iii) simultaneous multithreading (SMT).

- The proposed techniques can be implemented as design extensions to the control units of the in-order processor core, with negligible control overhead.

- The overhead due to wake-up latency is completely avoided in two techniques while it is hidden for most part in the third approach, either by overlapping the wake-up process with the thread context switching latency or by executing instructions from other threads ready for execution.

5.2 Register File Power Gating in CGMT Processors

In the CGMT approach, whenever a thread is switched, there is a multiple cycle penalty incurred due to the context switching process. The penalty is due to either squashing (or rolling back) of instructions from the pipeline or draining of the pipeline following an event that triggers the context switch. For instance, when a thread encounters a data load miss, all the instructions in the pipeline following the load instruction are squashed before a ready thread could be switched in. Conversely, in the case of an instruction fetch miss, all the leading instructions in the pipeline are allowed to

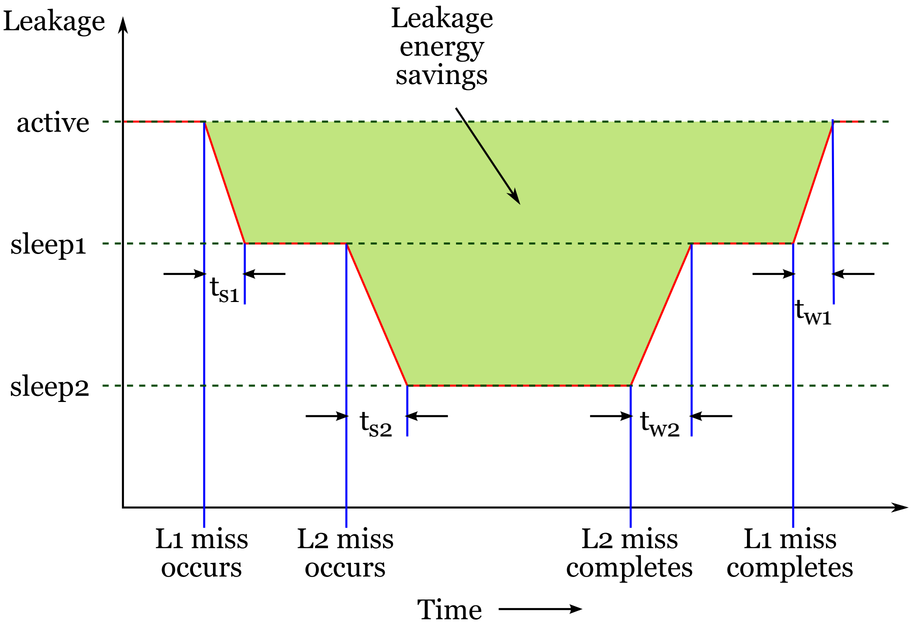


Figure 5.2 Intermediate strength power gating applied during a cache miss. When an L1 cache miss occurs, the register file is placed in the lower savings state (*sleep1*). Further, if L2 access also results in a miss, then it is placed in a higher savings state (*sleep2*). Otherwise, it stays in *sleep1* state. The wake-up latency t_{w2} is overlapped with the L1 fill latency. The wake-up latency t_{w1} is overlapped during thread context switching. The total leakage savings is the shaded/colored area in the figure.

finish before the next ready thread is switched in. In both cases, bubbles are inserted into the pipeline that negatively affects the pipeline performance. A direct approach to avoid this switch penalty is to have copies of the pipeline registers at each stage, which results in increased area and complexity of the CPU core. However, for short pipelines, the context switch penalty is only a few cycles and the additional hardware does not justify the small improvement in performance (for e.g., IBM Northstar/Pulsar [26]).

In this section, we describe the timing details involved in putting an integer register file in and out of low-leakage state following a memory stall. The wake-up latency of the register file is completely overlapped with the thread switch latency discussed above. We consider a CGMT model in which thread context switching happens only during instruction fetch misses (referred to as *fetch misses* in the remainder of the text) and data load misses (referred to as *load misses* in the remainder of the text). Also, the CPU pipeline is modeled around a MIPS pipeline with instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and writeback (WB) stages. However, the register file is read during the first cycle in the EX-stage and then dispatched to the arithmetic functional units.

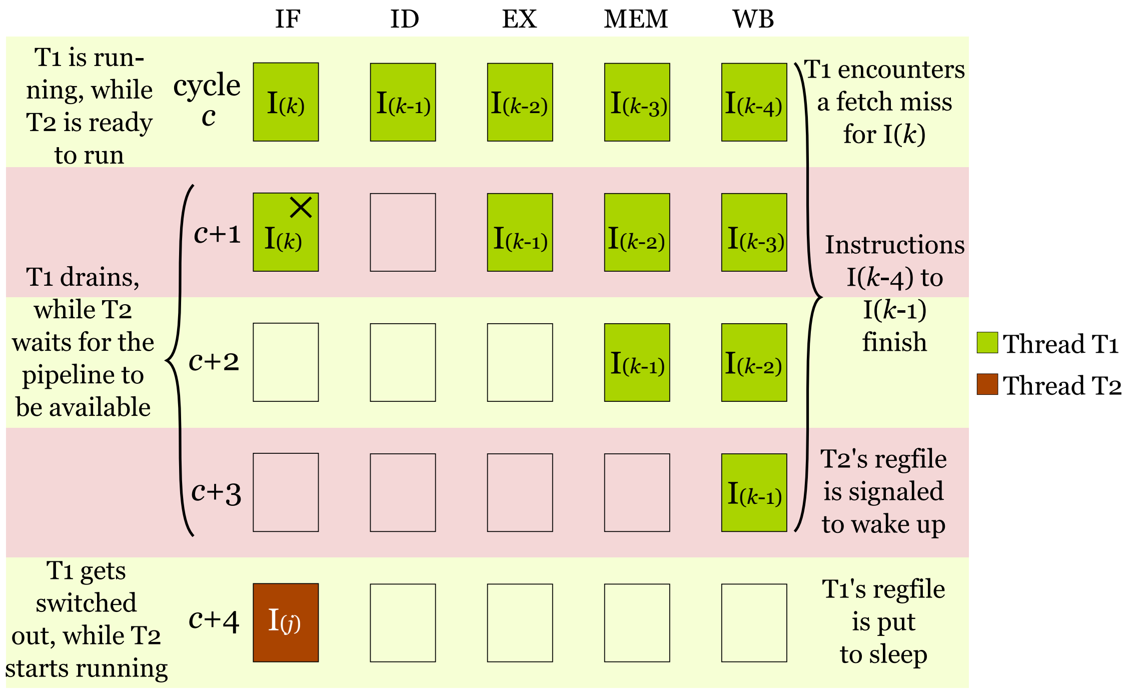
5.2.1 Power Gating Control During Fetch Miss

Figure 5.3 illustrates the scenario and explains the timing details of putting a register file to sleep following a fetch miss. The figure shows the state of the pipeline during clock cycle c, when thread T_1 is running while thread T_2 is in the ready state. The scenario described in this figure assumes that:

1. T_2 was switched out earlier following a fetch miss and was eventually put back in the ready state after its fetch miss completed.

2. T_2's register file is currently in a low-leakage state and needs 3 cycles to wake up before it can be accessed.

3. All the stages of the pipeline are busy executing T_1's instructions, I_{k-4} to I_k.


IF
ID
EX
MEM
WB
T1 is running, while T2 is ready to run
cycle c
I(k)
I(k-1)
I(k-2)
I(k-3)
I(k-4)
T1 encounters a fetch miss for I(k)
c+1
I(k)
I(k-1)
I(k-2)
I(k-3)
T1 drains, while T2 waits for the pipeline to be available
Instructions I(k-4) to I(k-1) finish
c+2
I(k-1)
I(k-2)
c+3
I(k-1)
T2's regfile is signaled to wake up
T1 gets switched out, while T2 starts running
c+4
I(j)
T1's regfile is put to sleep
🟩 Thread T1
🟥 Thread T2

T2's regfile will be accessed earliest in cycle *c*+6 in EX-stage (3 cycle wakeup)

Figure 5.3 Timing details for putting register files to sleep following an instruction fetch miss

While fetching I_k, an instruction fetch miss is encountered following which T_1 starts to drain in cycle $(c+1)$. This is done so that instructions I_{k-4} to I_{k-1} finish before a thread context switch happens. During this draining period, T_1's register file needs to be active so that reads and writes may be performed to it. Assuming that no other instruction in the pipeline gets stalled, the last instruction, I_{k-1}, finishes in cycle $(c+3)$. During this cycle, T_2's register file is signaled to wake up. In cycle $(c+4)$, thread T_2 gets switched in and it starts to fetch instruction I_j, while T_1's register file is put to sleep. By the time I_j reaches the EX-stage in cycle $(c+6)$ and accesses T_2's register file, it is already in active state. The wake-up latency is overlapped with the context switching latency and, therefore, the pipeline does not incur any stalls due to the unavailability of the register file.

Eventually, as shown in Figure 5.4, T_1's fetch miss completes at cycle $(c+m)$ and T_1 switches in to the ready state. Then, we can consider waking up T_1's register file.

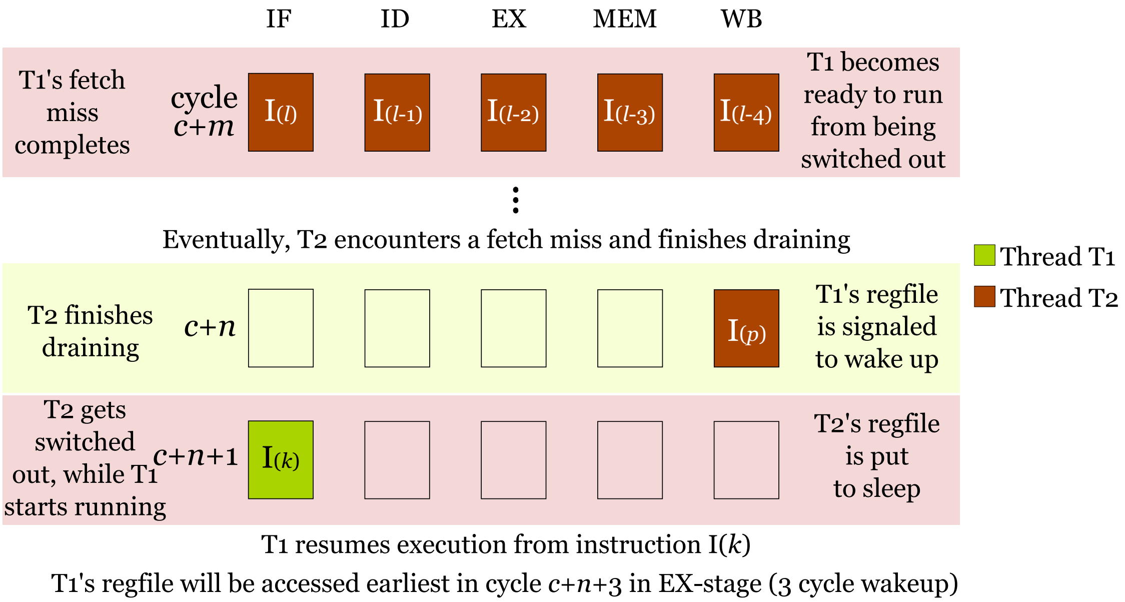

IF
ID
EX
MEM
WB
T1's fetch miss completes
cycle c+m
I(l)
I(l-1)
I(l-2)
I(l-3)
I(l-4)
T1 becomes ready to run from being switched out
⋮
Eventually, T2 encounters a fetch miss and finishes draining
T2 finishes draining
c+n
I(p)
T1's regfile is signaled to wake up
T2 gets switched out, while T1 starts running
c+n+1
I(k)
T2's regfile is put to sleep
T1 resumes execution from instruction I(k)
T1's regfile will be accessed earliest in cycle c+n+3 in EX-stage (3 cycle wakeup)
🟩 Thread T1
🟥 Thread T2

Figure 5.4 Wake-up details of T_1's register file if the pipeline is busy when its fetch miss completes. Resuming from the IF-stage means that the instruction is either fetched again or placed in the instruction buffer from the cache line following a cache line fill.

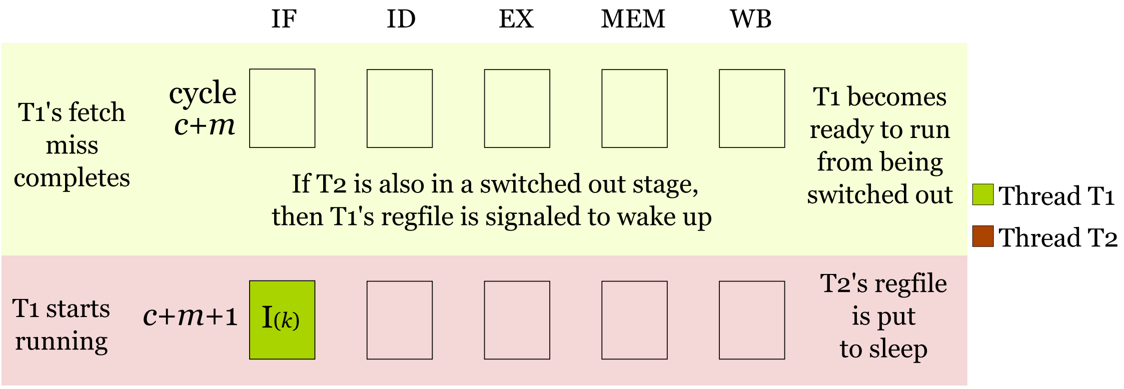
Two possible scenarios occur:

1. T_2 is currently running.

2. T_2 is switched out and the pipeline is currently idle.

In scenario 1, there is no need to wake up T_1's register file because T_1 will get to run only after T_2 gets switched out following a memory stall (assume a fetch stall again). This situation is shown in Figure 5.4 when at cycle $(c+n)$, T_2 finishes draining following a fetch stall and gets switched out. T_1's register file is signaled to wake-up during this cycle. During the next cycle, T_1 resumes execution from I_k, which would need to access the register file earliest during the EX-stage. This allows the 3-cycle wake-up period to for T_1's register file.

In scenario 2 (Figure 5.5), however, T_1 gets to run right after the fetch miss completes because T_2 is switched out and the pipeline is idle. Therefore, T_1's register file is signaled to wake up at cycle $(c+m)$. Since T_1 resumes execution (from instruction I_k) at cycle $(c+m+1)$, T_1's register file does not get accessed till cycle $(c+m+3)$ when it reaches the EX-stage.


IF
ID
EX
MEM
WB
T1's fetch miss completes
cycle c+m
If T2 is also in a switched out stage, then T1's regfile is signaled to wake up
T1 becomes ready to run from being switched out
🟩 Thread T1
🟥 Thread T2
T1 starts running
c+m+1
I(k)
T2's regfile is put to sleep

T1 resumes execution from instruction I(k)
T1's regfile will be accessed earliest in cycle c+m+3 in EX-stage (3 cycle wakeup)

Figure 5.5 Wake-up details of T_1's register file if the pipeline is idle after its fetch miss completes

In 1, T_1's register file stays in a low-leakage state for $(n-5)$ cycles, while in 2 it stays in a low-leakage state for $(m-5)$ cycles.

5.2.2 Power Gating Control During Load Misses

During data store misses, the thread need not stall as long as the result of the store instruction can be placed in a store buffer (unless the store is part of a specialized atomic instruction). However, during a data load miss, the thread gets stalled and it starts the transition process towards being switched out. At the same time, its register file is placed in a low-leakage state. In case, a newly switched in thread always starts from the IF-stage (as in Niagara), then the wake-up latency of the register file can be hidden with the number of cycles that the instruction takes to traverse from the IF-stage to the EX-stage. However, if the load instruction that encounters the load miss is placed in a load buffer in the MEM-stage so that it may resume execution as soon as the load miss gets processed, then efforts are needed to hide the wake-up latency. Load instructions write into the register file during the W-stage, and, therefore, the register file needs to be in active state before it can be written into.

Figure 5.6 shows the register file sleep strategy following a load miss. As in Figure 5.3, this figure also shows the state of the pipeline during clock cycle c, when thread T_1 is running and thread T_2 is in the ready state. The scenario described in this figure assumes that:

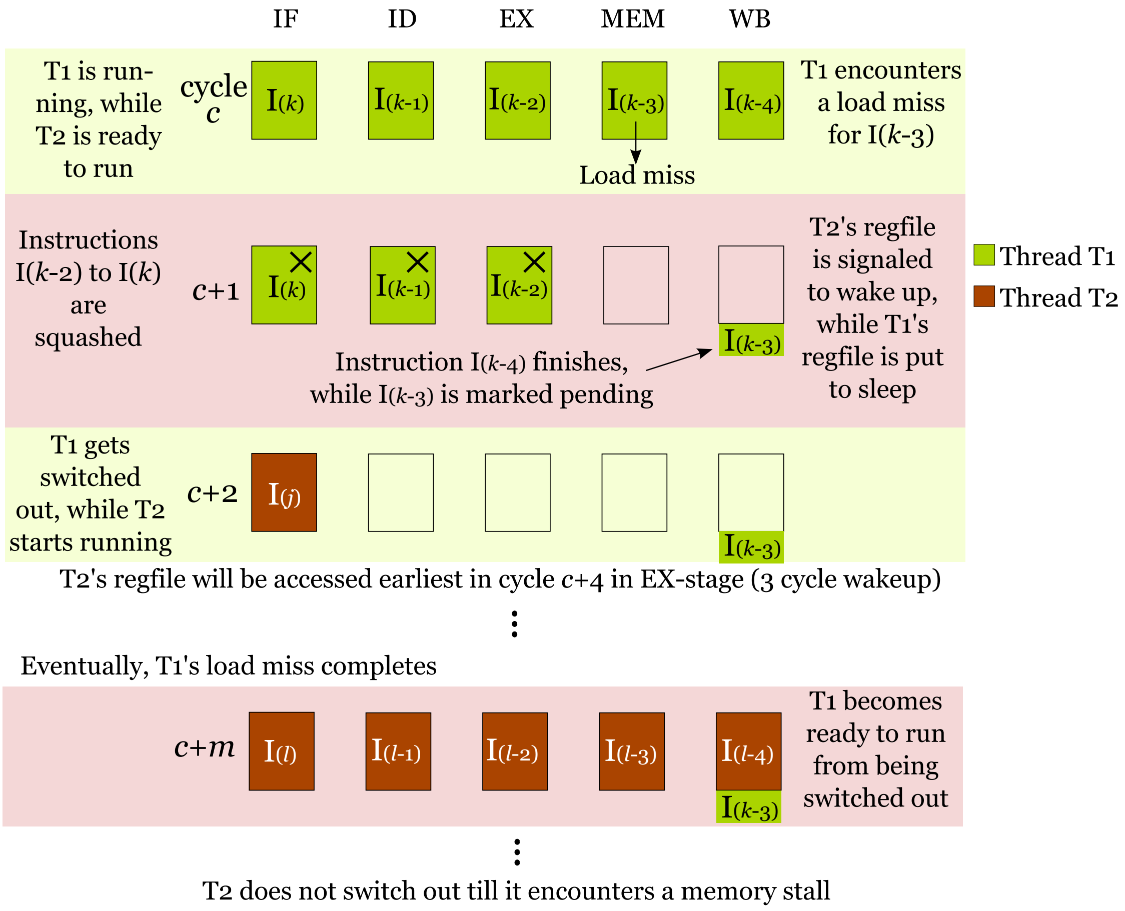

IF
ID
EX
MEM
WB
T1 is running, while T2 is ready to run
cycle c
I(k)
I(k-1)
I(k-2)
I(k-3)
I(k-4)
T1 encounters a load miss for I(k-3)
Load miss
Instructions I(k-2) to I(k) are squashed
c+1
I(k)
I(k-1)
I(k-2)
I(k-3)
T2's regfile is signaled to wake up, while T1's regfile is put to sleep
Instruction I(k-4) finishes, while I(k-3) is marked pending
Thread T1
Thread T2
T1 gets switched out, while T2 starts running
c+2
I(j)
I(k-3)
T2's regfile will be accessed earliest in cycle c+4 in EX-stage (3 cycle wakeup)
Eventually, T1's load miss completes
c+m
I(l)
I(l-1)
I(l-2)
I(l-3)
I(l-4)
I(k-3)
T1 becomes ready to run from being switched out
T2 does not switch out till it encounters a memory stall

Figure 5.6 Timing details for putting a register file to sleep following a data load miss. An instruction marked pending implies that it is placed in the load buffer.

1. T_2 was switched out earlier following a *fetch miss* and was eventually put back in the ready state after its fetch miss completed.

2. T_2's register file is currently in a low-leakage state and needs 3 cycles to wake up before it can be accessed.

3. All the stages of the pipeline are busy executing T_1's instructions, I_{k-4} to I_k.

Following a load miss encountered by thread T_1 while executing instruction I_{k-3} in the MEM-stage, instructions I_{k-2} to I_k are squashed in cycle $(c+1)$, while I_{k-3} is placed in the load buffer. Since T_2 will be switched in to be executed during the next cycle, its register file is signaled to wake up, while T_1's register file is put to sleep. In cycle $(c+2)$, T_2 resumes execution by fetching instruction I_j, which does not access the register file till cycle $(c+4)$, thereby giving it the adequate number of cycles to become active.

Eventually, when T_1's load miss completes during a later cycle, say $(c+m)$, T_1 transitions from switched out state to the ready state. Again, the decision to wake up its register file depends on whether T_2 is running (condition shown in Figure 5.7) or is switched out. In the former case, the register file is signaled to wake up when T_2 eventually encounters a stall (a *load miss* this time) and gets switched out (cycle $(c+n)$ in Figure 5.7). In the following cycle, cycle $(c+n+1)$ as shown in the figure, T_1 resumes execution from I_{k-2} in IF-stage and I_{k-3} in W-stage. Since a load instruction needs to write into the register file in the W-stage, it is stalled for three cycles to allow the 3-cycle wake-up latency needed by the register file. This timing also coincides with the earliest cycle that T_1's register file need to be accessed by I_{k-2} (when it reaches the EX-stage). If, however, there is a read-after-write (RAW) data dependency between I_{k-3} and I_{k-2}, then the result of the load operand is forwarded to the functional unit that needs it as a operand to execute instruction I_{k-2}.

5.3 Register File Power Gating in FGMT Processors

In contrast to the CGMT approach, FGMT and SMT approaches do not typically suffer from multiple cycle thread switch penalties. This is because, in these approaches, each pipeline stage processes one or more instructions from multiple threads. If an instruction from a thread encounters a stall, no further instructions from that thread are fetched to be dispatched to the pipeline. Instead, instructions from one or more of the ready threads are fetched and processed. The policy to select a thread to fetch from may vary across designs. For instance, Niagara uses round-robin (RR) policy to select one thread among a list of ready threads, while Niagara2 implements a least recently fetched (LRF) policy to do the same. As long as there are ready threads available to the CPU, no bubbles are inserted into the pipeline. This very idea is utilized to hide the wakeup latency of the register files when they are transitioning from the low-leakage state to the active state.

5.3.1 Power Gating Control During Fetch Miss

A CPU that is designed to support the FGMT approach, fetches an instruction from a new ready thread each cycle and dispatches it to the pipeline. Figure 5.8 illustrates the timing details of putting a thread's register file in and out of low-leakage state following an instruction fetch miss. It is assumed that the CPU has 4 hardware threads, T_{1-4}, and a round-robin fetch policy is implemented. The pipeline contents are shown for clock cycle c. Instruction from all four threads are currently being processed by the different stages in the pipeline when an instruction from T_1 encounters a fetch miss. Therefore, in the next cycle, $(c+1)$, one of the ready threads is selected (T_2 in the figure) and an instruction from that thread is fetched. The decision to assert the sleep control to

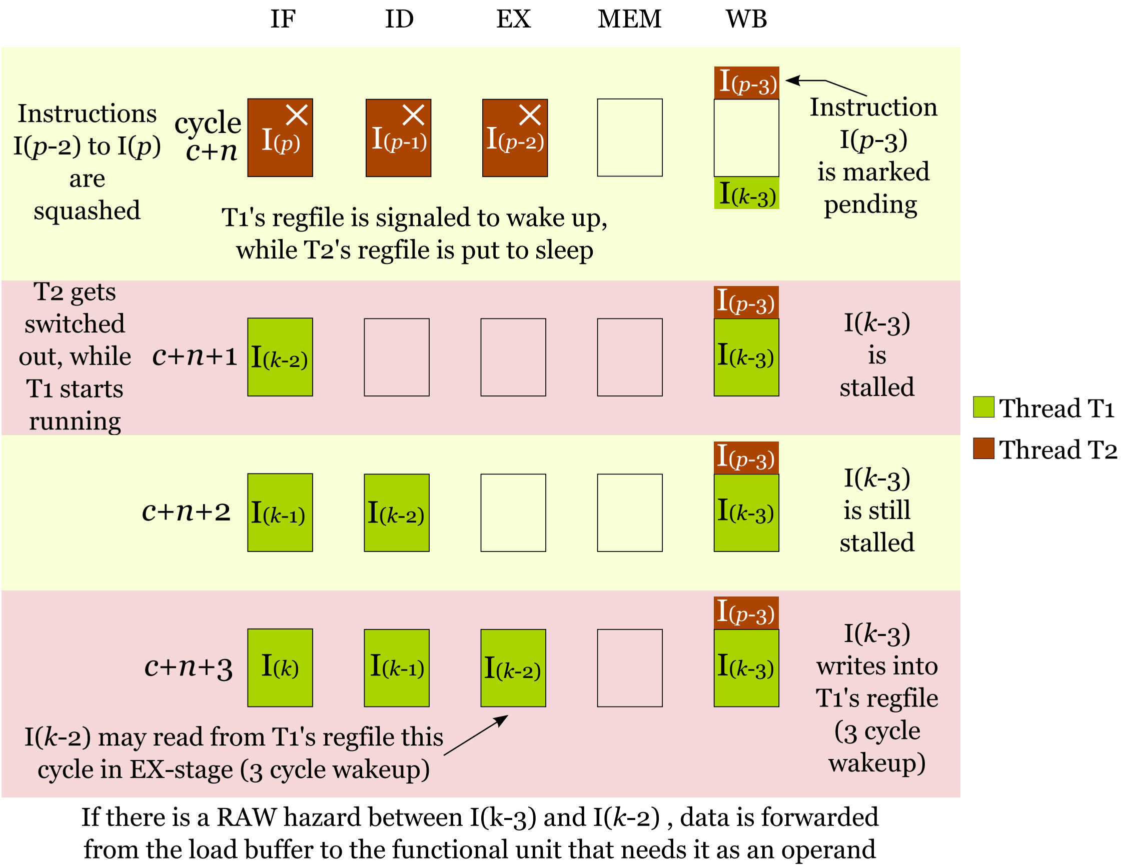

IF
ID
EX
MEM
WB
Instructions I(p-2) to I(p) are squashed
cycle c+n
I(p)
I(p-1)
I(p-2)
I(p-3)
I(k-3)
Instruction I(p-3) is marked pending
T1's regfile is signaled to wake up, while T2's regfile is put to sleep
T2 gets switched out, while T1 starts running
c+n+1
I(k-2)
I(p-3)
I(k-3)
I(k-3) is stalled
c+n+2
I(k-1)
I(k-2)
I(p-3)
I(k-3)
I(k-3) is still stalled
c+n+3
I(k)
I(k-1)
I(k-2)
I(p-3)
I(k-3)
I(k-3) writes into T1's regfile (3 cycle wakeup)
I(k-2) may read from T1's regfile this cycle in EX-stage (3 cycle wakeup)
🟩 Thread T1
🟥 Thread T2
If there is a RAW hazard between I(k-3) and I(k-2) , data is forwarded from the load buffer to the functional unit that needs it as an operand

Figure 5.7 Timing details for waking up T_1's register file from sleep after its load miss completes and it gets ready to run. Resuming from the W-stage indicates that the result of the load is filled into the load buffer from the cache line.

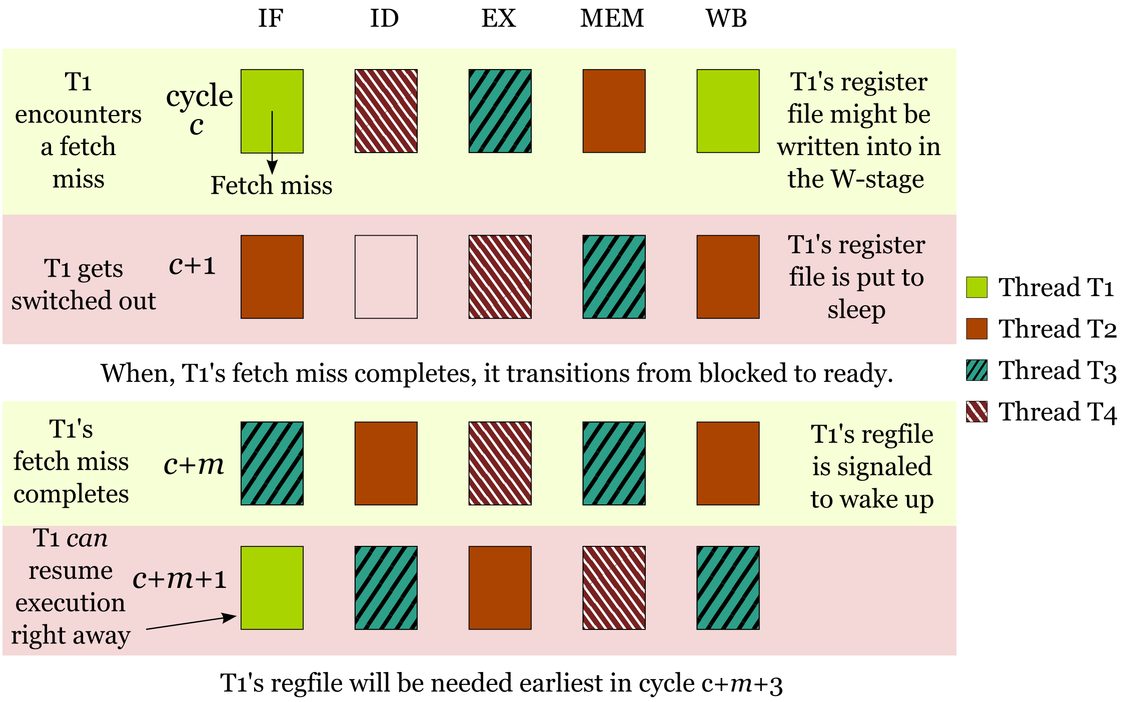

IF
ID
EX
MEM
WB
T1 encounters a fetch miss
cycle c
Fetch miss
T1's register file might be written into in the W-stage
T1 gets switched out
c+1
T1's register file is put to sleep
When, T1's fetch miss completes, it transitions from blocked to ready.
T1's fetch miss completes
c+m
T1's regfile is signaled to wake up
T1 can resume execution right away
c+m+1
T1's regfile will be needed earliest in cycle c+m+3
Thread T1
Thread T2
Thread T3
Thread T4

Figure 5.8 Timing details for putting a thread's register file in and out of low-leakage state following a fetch miss in FGMT

T_1's register file depends on whether there are any instructions belonging to T_1 in the pipeline and require to access the register file. For instance, as shown in the figure, one of T_1's instructions is in the W-stage in cycle c. Therefore, it is imperative that the register file be active till that instruction finishes writing into the register file. In this case, T_1's register file is put to sleep at the end of cycle $(c+1)$.

Eventually, when T_1's fetch miss completes at cycle $(c+m)$, it becomes ready to run. T_1's register file is signaled to wake up right away. Assuming that it is indeed selected by the thread scheduler in the next cycle, i.e., cycle $(c+m+1)$, it will access T_1's register file earliest in cycle $(c+m+3)$, thereby providing for the 3-cycle wake-up latency.

5.3.2 Power Gating Control During Load Miss

In FGMT processors, when a load miss is detected for an instruction from a thread, all the instructions following the load instruction in the pipeline are squashed (or rolled back to the in-

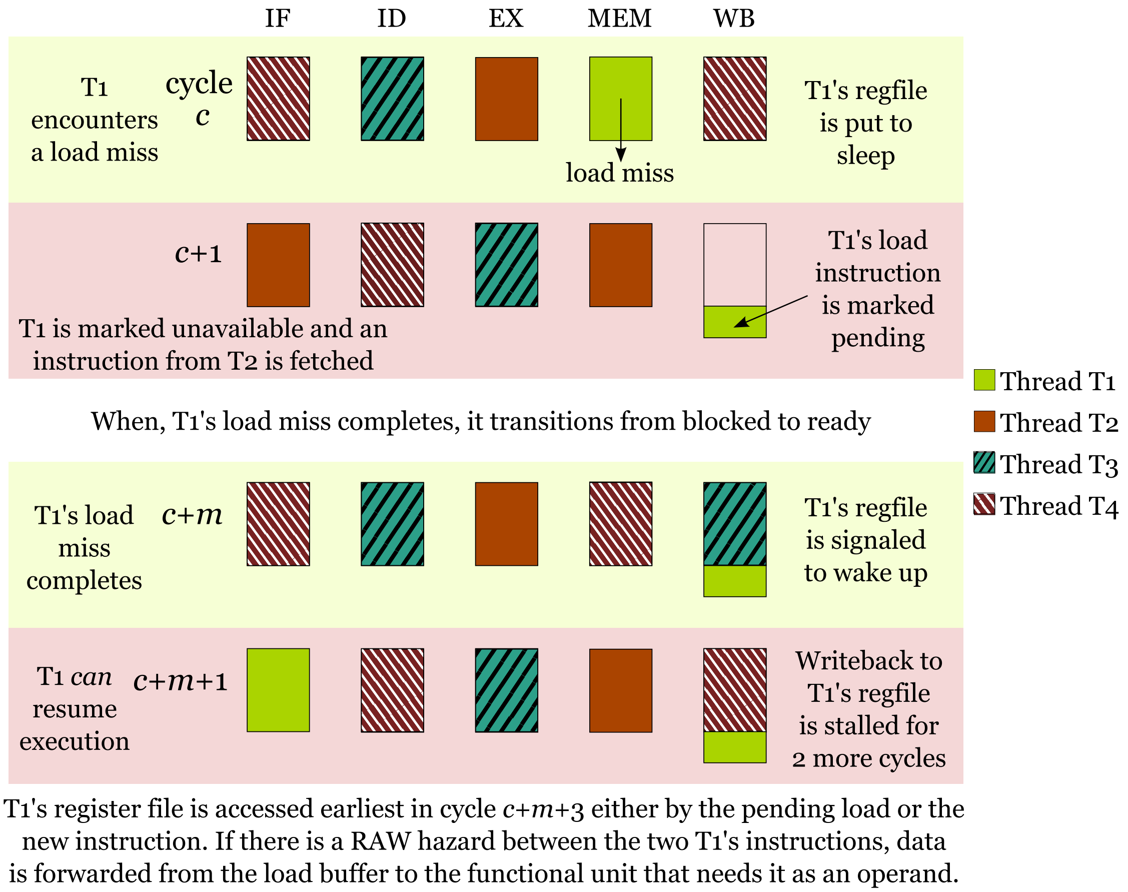

IF
ID
EX
MEM
WB
T1 encounters a load miss
cycle *c*
load miss
T1's regfile is put to sleep
c+1
T1's load instruction is marked pending
T1 is marked unavailable and an instruction from T2 is fetched
Thread T1
Thread T2
Thread T3
Thread T4
When, T1's load miss completes, it transitions from blocked to ready
T1's load miss completes
c+*m*
T1's regfile is signaled to wake up
T1 *can* resume execution
c+*m*+1
Writeback to T1's regfile is stalled for 2 more cycles
T1's register file is accessed earliest in cycle *c*+*m*+3 either by the pending load or the new instruction. If there is a RAW hazard between the two T1's instructions, data is forwarded from the load buffer to the functional unit that needs it as an operand.

Figure 5.9 Timing details for putting a thread's register file in and out of low-leakage state following a data miss in FGMT

struction buffer). However, since in each cycle a different thread is dispatched to the pipeline in an FGMT approach, the number of instructions squashed is expected to be much smaller than that in the case of CGMT. The load instruction itself may also be squashed or it may be marked pending at the MEM-stage (in a load buffer). The choice of implementation here impacts the wake-up strategy applied to wake up the register file for a stalled thread when its load miss completes. In the former case (as in Niagara), the wake-up latency of the register file is overlapped with the number of cycles that the instruction takes to reach the EX-stage from the IF-stage (described earlier in Section 5.2.2). However, in the latter case (shown in Figure 5.9), the writeback to the register file is deferred for additional cycles (2 cycles in the figure) to account for the wake-latency. If there is a RAW hazard between the two T_1's instructions in the pipeline, then data is forwarded from the load buffer to the consuming instruction when it reaches the EX-stage.

5.4 Register File Power Gating in SMT Processors

We model a generic simultaneous multithreading in-order core processor, in which each pipeline stage is capable of processing multiple instructions from *distinct threads* during the same clock cycle. To support this capability, each pipeline stage is equipped with stage buffers for each thread context. Once an instruction is processed by the stage, it is placed in the stage buffer for its thread context for the next stage to process it in a subsequent clock cycle. However, if an instruction gets stalled at a stage due to a multicycle latency operation, like an integer multiplication or a memory stall, it is marked blocked or unavailable till the operation finishes. Along with that, all the instructions from that thread in all the preceding stages are also marked blocked. Each stage picks up instructions from only ready threads to process during a clock cycle.

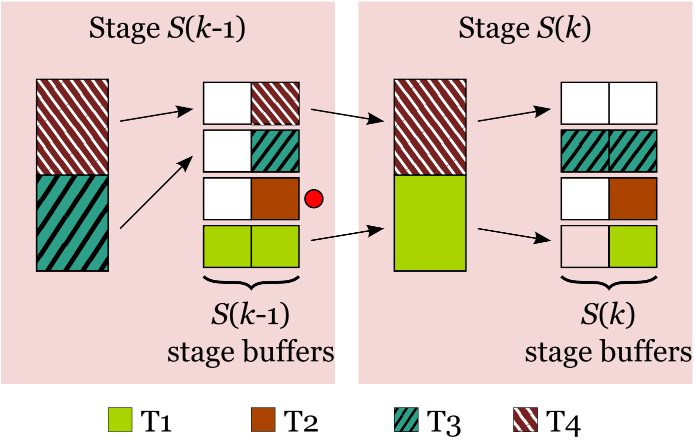


Figure 5.10 Schematic view of a pipeline organization to support SMT in in-order cores. Each pipeline stage has stage buffers for each thread context. Instructions from these stage buffers are chosen to be processed by the next stage. During a pipeline stall, the thread is marked blocked in all the preceding stages (indicated by the circular shape). Only instructions from the ready threads are picked and processed by each stage.

Figure 5.10 shows the schematic view of this structure. It shows a snapshot of two back to back stages of the pipeline, S_{k-1} and S_k. S_{k-1} processes instructions from threads T_3 and T_4 and places them in their respective thread buffers in that stage. S_k, on the other hand, picks up instructions from threads T_1 and T_4 from S_{k-1}'s stage buffers, processes them, and places them in their respective thread buffers in this stage. Thread T_2 is shown to have been marked as unavailable (the circular shape in the figure) or stalled in S_{k-1} because it is currently performing a multicycle latency operation. Also, an instruction from T_3 cannot be processed by S_k since the buffer for thread T_3 is full in this stage.

In this SMT core, placing the register file in and out of the low-leakage state during both fetch misses and data misses is very similar. Once a miss is detected, the thread is marked blocked in *all* the stages in the pipeline so that the register file may be put to sleep immediately. Note that this is different from the regular case in which the thread is marked blocked only in the preceding stages. When the miss completes, the register file is signaled to wake up but the thread is marked ready only after a few additional cycles to account for the wake-up latency of the register file. As long as there are instructions from other ready threads in the pipeline, the additional blocked cycles do not result in any performance degradation.

5.5 Summary of the Proposed Techniques

A summary of the proposed techniques for CGMT, FGMT, and SMT cores is presented in Table 5.1. In the CGMT approach, the wake-up latency of the register files is overlapped with the latency associated with the latency of thread context switching. It can also be observed that, since no more than one thread is active simultaneously, the register files for all the other threads, irrespective of whether they are stalled or ready, may be kept in low-leakage states. Thus, as the number of threads are increased in a CGMT approach, it is expected that the leakage savings in the register files also increase linearly. On the other hand, in the FGMT and SMT approaches, multiple threads are simultaneously active in the CPU at the same time. Therefore, the leakage savings achieved in these approaches are not expected to scale with the number of hardware thread contexts supported by the CPU. Instead, they are expected to be proportional to the fraction of the time that the threads spend waiting on memory stalls.

Table 5.1 A summary of the proposed techniques

Control Feature	CGMT	FGMT	SMT
Sleep after a fetch miss	Wait till the pipeline drains	Wait till there are no instructions from the target thread in the pipeline	Immediately; block all the instructions belonging to this thread in the entire pipeline
Wake up after the fetch miss completes	When the thread gets switched in (its state changes from ready to run)	As soon as the fetch miss completes	As soon as the fetch miss completes
Performance degradation due to wake-up overhead	Zero cycles; the thread resumes execution from the IF-stage	Zero cycles; the thread resumes execution from the IF-stage	Best case is zero; there are instructions available from other threads. Worst case is wake-up latency number of cycles; otherwise.
Sleep after a load miss	Immediately; any following instructions in the pipeline are squashed	Immediately; any following instructions from that thread in the pipeline are squashed	Immediately; block all the instructions belonging to this thread in the entire pipeline
Wakeup after the load miss completes	When the thread gets switched in (its state changes from ready to run)	As soon as the load miss completes	As soon as the load miss completes
Performance degradation due to wake-up overhead	Zero cycles; either the load instruction or the instruction following the load resumes execution from the IF-stage.	Zero cycles; either the load instruction or the instruction following the load resumes execution from the IF-stage.	Best case is zero; there are instructions available from other threads. Worst case is wake-up latency number of cycles; otherwise.

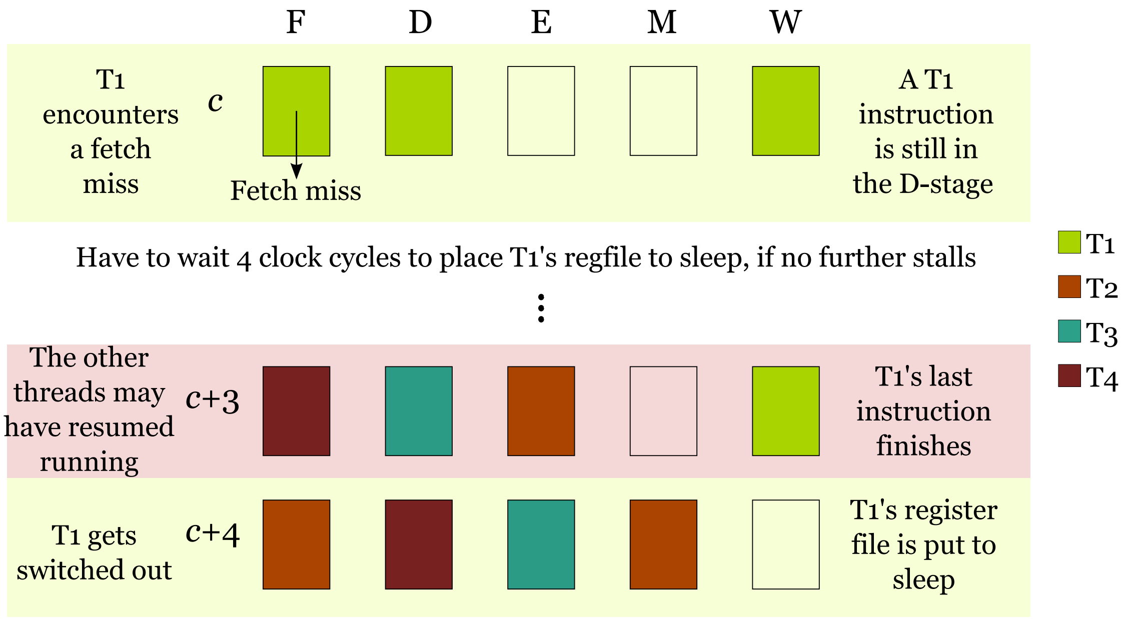

F
D
E
M
W
T1 encounters a fetch miss
c
Fetch miss
A T1 instruction is still in the D-stage
Have to wait 4 clock cycles to place T1's regfile to sleep, if no further stalls
The other threads may have resumed running
c+3
T1's last instruction finishes
T1 gets switched out
c+4
T1's register file is put to sleep
T1
T2
T3
T4

Figure 5.11 A pathological case during a fetch miss in an FGMT core. All the other threads are switched out and only instructions belonging to T_1 are in the pipeline.

Further, in the CGMT and FGMT cores, the latency of putting a register file from a low-leakage state to the active state can be overlapped completely, thereby not having to incur any performance overhead. However, for the SMT cores, performance degradation can happen when there are not enough ready threads in the core and keeping a thread blocked for the additional cycles inserts bubbles into the pipeline. However, the likelihood of this scenario can be reduced by increasing the number of threads that the SMT core is able to support.

Furthermore, the amount of savings achieved is also influenced by the number of ready threads available. One pathological case for the FGMT approach is shown in Figure 5.11 where threads T_{2-4} are all stalled and only instructions from thread T_1 are in the pipeline. Following a fetch miss in cycle c, T_1's register file may not be put to sleep till the all of T_1's instructions are drained. In the example depicted in the figure, we have to wait four additional cycles to do so. If this stall were resolved in 10 cycles (a typical L2 hit latency for a CPU clock frequency of 1 GHz), then the savings achieved by this technique are reduced by 40%.

5.6 Experimental Setup and Results

In this section, we describe the experimental setup used in this work to evaluate the effectiveness of the proposed techniques in multi-core processors.

5.6.1 Integer Register File Characterization

For the purpose of estimating leakage characteristics and the latency of a register file with intermediate-strength power gating, we consider a 32-entry 64-bit flip-flop based register file (without any error correction code bits) with two read ports and one write port in 45nm FreePDK technology [64]. The layout design of a D-flip-flop from the Nangate 45 nm open cell library [65] was extended to include the two read ports and one write port and a SPICE netlist with parasitics was extracted using Calibre from Mentor Graphics. The characterization of the register file was performed using spice simulations. The average access latency was 0.893 ns. The leakage states *sleep1* and *sleep2* reduce leakage by 36% and 52% respectively. Their wake-up latencies, for a clock of 1 GHz, were computed to be 3 cycles and 5 cycles. The breakeven periods were shorter than the wake-up latencies. These details outlined in Table 5.2.

Table 5.2 Register file leakage states

Leakage State	Normalized Leakage	Wakeup Latency (1 GHz Clock)
active	1	-
sleep1	0.64	3 cycles
sleep2	0.48	5 cycles

5.6.2 Processor Configurations and Workload Details

We used the M5 simulator [66] for modeling the multi-core processors featuring multithreaded CPU cores. The M5 simulator supports four different CPU models to provide simulation platforms for functional and detailed simulations. Among them, *O3CPU* models a detailed out-or-order processor core and the *InOrderCPU* models a detailed in-order processor core. The in-order code has some default support for both CGMT and SMT models. It was further extended to provide comprehensive support to model the multithreading approaches described in this chapter. We run all our detailed simulations for DEC Alpha ISA in syscall emulation mode.

Table 5.3 SPEC 2000 integer benchmarks

Name	Dynamic Instruction Count (Millions)
vpr	17.6
gap	44.8
vortex	88.3
twolf	91.9
eon	94.0
crafty	94.4
gcc	96.8
mcf	188.6
perlbmk	200.6
parser	267.8
gzip	601.9

- The dynamic instruction counts are for the small reduced (smred) input sets [67].
- Benchmark `bzip2` is not shown because it does not have an smred input set.

Table 5.3 enumerates the integer benchmarks from the SPEC 2000 benchmark suite and their dynamic instruction counts for the small reduced (smred) input sets [67]. The binaries are tru64 binaries (COFF version 3.11-10) built with optimization levels O2 and O3. Multiprogrammed workloads for each processor core is created by choosing the required number of benchmarks (all distinct) at random. Simulations are run till the first thread finishes execution.

We configure a number of multi-core processors comprising of in-order CPU cores by varying the number of cores, the number of hardware contexts that each core supports, and a number of L1 and L2 cache parameters. The processor parameters are tabulated in Table 5.4. The number of cores is either two, four or eight. The number of hardware threads that each core supports is scaled from two to eight in case of CGMT, from three to eight in case of FGMT, and from four to eight in case of SMT. We cap the number of threads per core at eight threads because the cost growth for supporting additional hardware threads is linear up to around eight threads but is superlinear after that [68].

The in-order cores have simple specifications. In case of CGMT and FGMT, the cores can process at most one instruction each cycle in each of its pipeline stages, while, in the case of SMT, the

Table 5.4 Multi-core processor parameters

Parameter	Multithreading Approach		
	CGMT	FGMT	SMT
Clock Speed	1 GHz		
Number of Cores	2, 4, and 8		
Number of Contexts	2-8	3-8	4-8
Pipeline Bandwidth (in insts/cycle)	1	1	2
Functional Units	1 int ALU 1 int Mult	1 int ALU 1 int Mult	2 int ALU 1 int Mult
Load/Store/Fetch Buffers	1 per thread		
Fetch Select Policy	Round-robin		

cores can process two. Therefore, we provide two integer ALUs to each core in case of SMT but only one integer ALU to the cores in case of CGMT and FGMT. The count of integer multipliers, however, is the same (one) for all the cores. The execution latencies of the integer ALU and integer multiplier are 1 cycle and 3 cycles, respectively. Furthermore, we model a fully pipelined integer multiplier so that integer multiply instructions that are not data dependent on each other may be issued every clock cycle. The clock frequency is uniform (1 GHz) across all the processor configurations. Each core has one load buffer, one store buffer, and one fetch buffer per thread. The policy to select a thread to fetch instructions from is set to round-robin.

The cache parameters are tabulated in Tables 5.5, 5.6, 5.7, 5.8, and 5.9. We set the cache parameters based on the specifications of the Niagara series of processors. The hit latencies for the

Table 5.5 Memory access latencies

Memory Unit	Access Latency
L1 D-cache	1 cycle
L1 I-cache	1 cycle
L2 cache (shared)	10 cycles
Physical Memory	30 cycles

Table 5.6 L1 D-cache and I-cache parameters

Size	2 cores	4 cores	8 cores
	64 KB	32 KB	16 KB
Set Associativity	2 TCs	3-4 TCs	5-8 TCs
	2	4	8
MSHRs	*as many as the number of TCs*		

TCs - Thread Contexts

private L1 caches (both I-cache and D-cache) and the shared L2 cache are set to 1 cycle and 10 cycles, respectively.

Table 5.7 L2 cache size (in MB)

Number of Threads	Number of Cores		
	2	4	8
2	2	3	4
3-4	3	4	6
5-8	4	6	8

Table 5.8 L2 cache set associativity

Number of Threads	Number of Cores		
	2	4	8
2	4	6	8
3-4	6	8	12
5-8	8	12	16

Table 5.9 L2 cache MSHR count

Number of Threads	Number of Cores		
	2	4	8
2	4	6	8
3-4	6	8	12
5-8	8	12	16

The physical memory access latency is set to 30 cycles. The cache line size for each cache is set to 64 bytes. As the number of cores are increased, the private L1 caches are reduced in size to have larger shared L2 caches. Therefore, while the L1 cache size is scaled down from 64 KB to 16 KB as the number of cores is increased from 2 to 8, the size of the L2 cache is scaled upward between 2 MB and 8 MB based on the number of cores and the number of thread contexts (Tables 5.6 and 5.7).

When the number of cores and the number of thread contexts increase, the set associativity for both L1 and L2 caches are increased to reduce the number of conflict misses (Tables 5.6 and 5.8). The number of Miss Status Handling Registers (MSHRs) in the L1 caches is also increased with the number of thread contexts to allow at most one outstanding L1 cache miss per thread (Table 5.6). The MSHR count for the L2 cache is dependent on both the number of cores and the number of thread contexts (Table 5.9). During the simulations, the caches are warmed up for the first 100,000 cycles.

5.6.3 Results

The instructions per cycle (IPC) counts for the workloads on the different multi-core processor configurations are plotted in Figures 5.12, 5.13, and 5.14. The IPC for CGMT cores ranges between 0.41 and 0.52, while for FGMT cores the IPC is between 0.61 and 0.68. This marked difference is primarily due to the fact that the FGMT approach is very effective in hiding stalls due to both long latency events (for e.g., cache misses) and short latency events (branch resolution, data dependency resolution, etc.). However, CGMT switches threads to hide stalls only due to long latency events. Moreover, the thread switch penalty in CGMT cores may be more than one cycle, whereas, in FGMT cores, this penalty is exactly one cycle as long as there are ready threads available.

The IPC counts for the SMT cores are in the range of 0.91 and 1.22 because the pipeline width for the SMT cores is double of that of the CGMT and FGMT cores. For the same number of threads, the IPC reduces as the number of cores are increased because the L1 cache sizes become smaller. It can also be observed that while the IPC counts for the CGMT and FGMT cores tend to saturate as the number of thread contexts is increased to eight, the IPC for SMT cores increase more linearly indicating that it could support more threads before its performance levels out.

The leakage savings in the integer register files in CGMT cores is shown in Figure 5.15. As expected, the savings in the CGMT processors scale very well with the number of thread contexts. For 2 thread contexts, the savings are in the range of 0.9% to 2.9%, while, for 8 thread contexts, the savings are between 22% and 42%. This is because, in a CGMT approach, only a single thread context is active at a time in the entire pipeline till it experiences a long latency stall. Therefore,

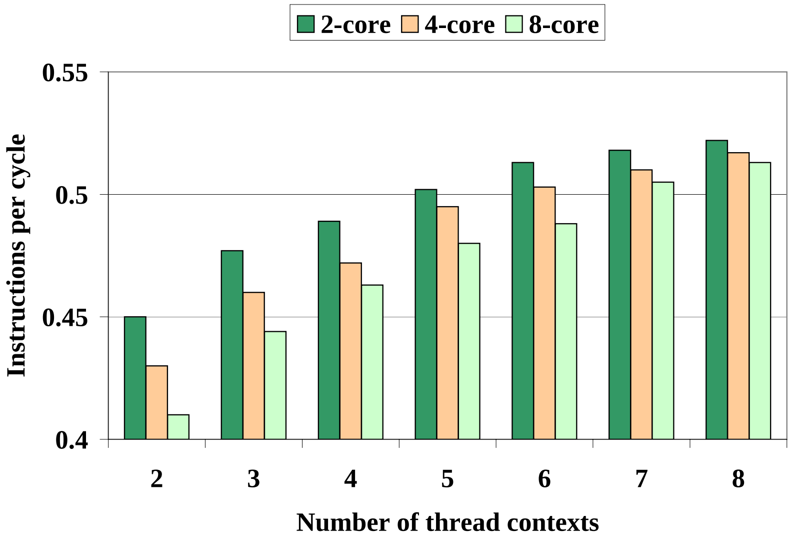


Figure 5.12 Average instructions per cycle (IPC) count for CGMT approach

the register files for the rest of the thread contexts, irrespective of whether they are ready or stalled, may be put to sleep.

In contrast to this, in the FGMT and SMT approaches, instructions belonging to different thread contexts are processed by different pipeline stages. Therefore, the savings do not scale with the number of thread contexts but instead are proportional only to the fraction of the time that is spent by the threads waiting on memory stalls. For FGMT cores, the leakage savings range from 0.8% to 2.02% for 3 thread contexts and 3.09% - 7.8% for eight thread contexts (Figure 5.16).

The total latencies of L1 D-cache read misses, L1 I-cache read misses, and L2 read misses (normalized over the total number of CPU cycles) averaged over the number of threads are plotted in Figures 5.17, 5.18, and 5.19.

For SMT cores, the leakage savings range from 1.02% to 2.23% for 4 thread contexts and 2.97% - 7.27% for eight thread contexts (Figures 5.20). The degradation in performance due to the proposed technique in SMT cores was calculated by counting the number of cycles when an instruction could not be processed by a pipeline stage because the register file was not awake. For SMT cores, the degradation was 0.023% in case of a 8-core processor with each core executing 8 threads.

5.7 Discussion

With continued scaling in modern VLSI circuit fabrication technology, leakage power and heat dissipation limits have driven not only circuit designers to devise newer techniques to reduce power dissipation in circuits but also the CPU designers to change the paradigm of designing processors to improve performance. Hardware multithreading and multi-core processors are outcomes of this new design paradigm. The work presented in this chapter leverages existing circuit level techniques to reduce leakage in such processors. In this work, we synchronize the sleep of a register file private to a thread with the unavailability of that thread and the wake-up with the readiness of that thread to run. This is because the integer register file is accessed very frequently by integer applications.

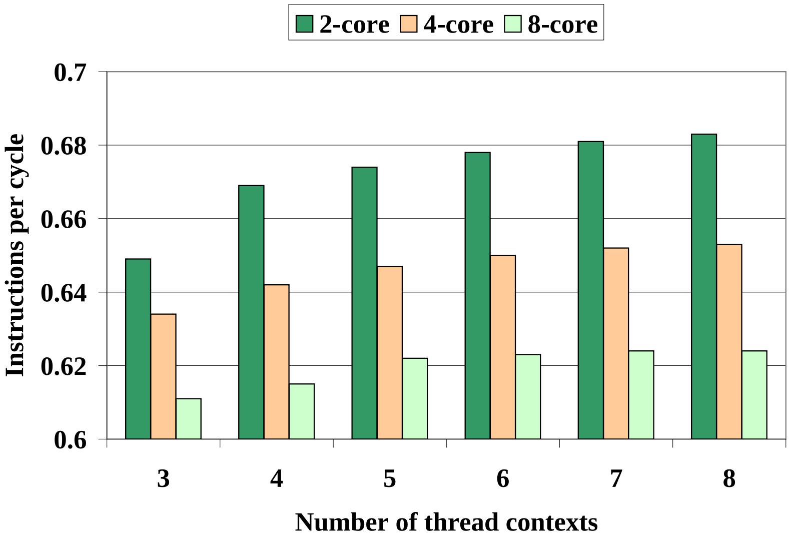


Figure 5.13 Average instructions per cycle (IPC) count for FGMT approach

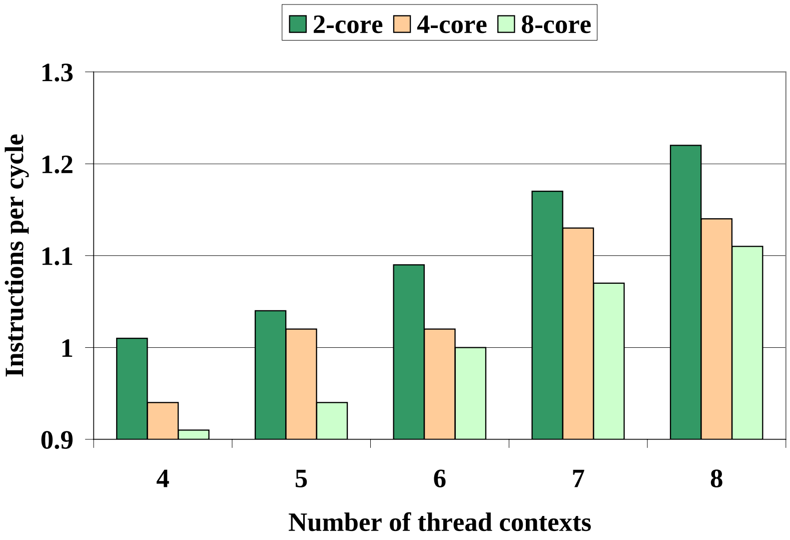


Figure 5.14 Average instructions per cycle (IPC) count for SMT approach

In the future, we would like to extend this work to in-order cores with floating point register files. Floating point applications use both integer and floating point register files and, therefore, the patterns of accesses to these units may provide more opportunities of power gating the register files. Therefore, the fundamental approach that is at the core of the techniques proposed in this work will result in conservative savings if directly applied in the case of floating point register files. Also, using a partitioned register file for each thread could have further advantages. Instead of waking up the entire register file at the same time, only the partitions that needs to be accessed can be woken up more urgently compared to the rest of the partitions.

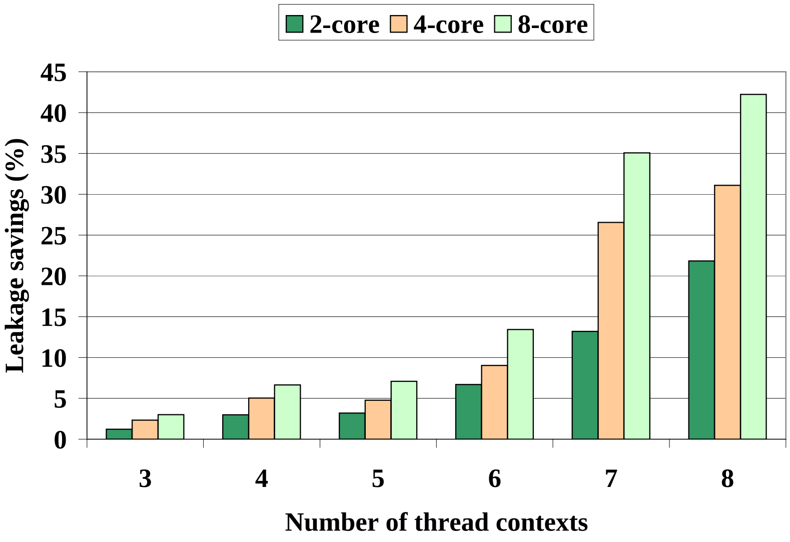


Figure 5.15 Average IRF leakage energy savings for CGMT cores

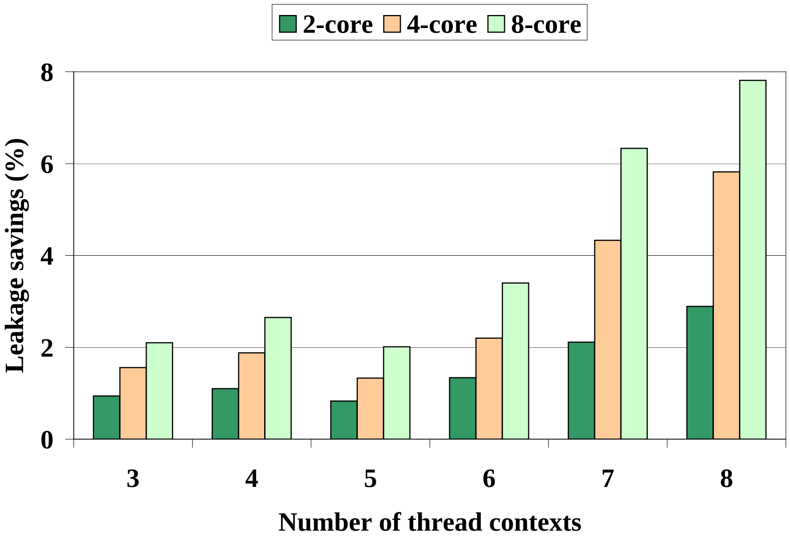


Figure 5.16 Average IRF leakage energy savings for FGMT cores

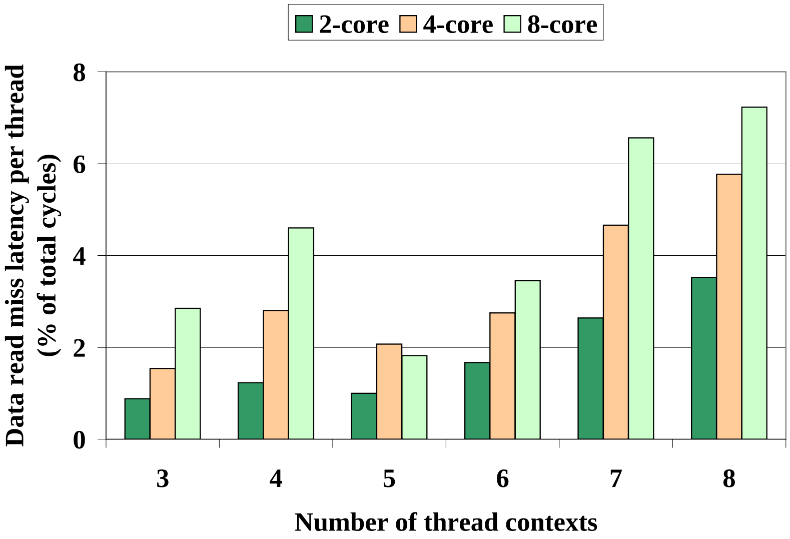


Figure 5.17 Data read miss latency per thread for FGMT cores

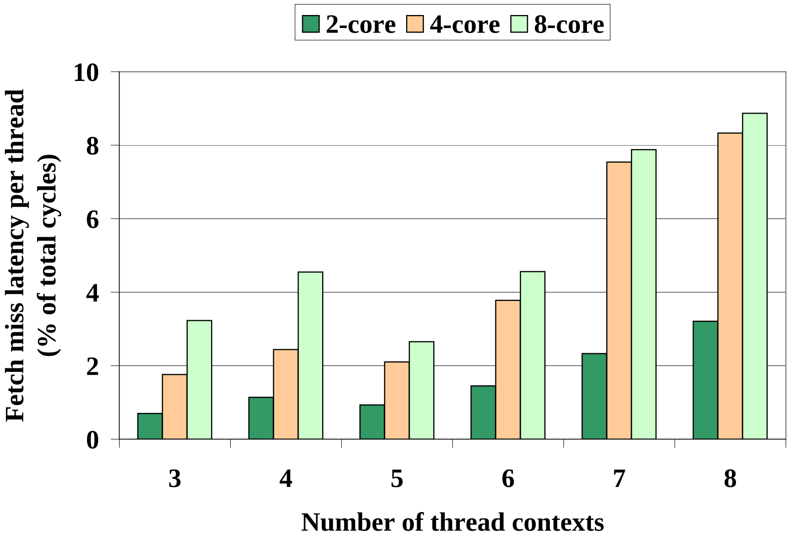


Figure 5.18 Instruction fetch miss latency per thread for FGMT cores

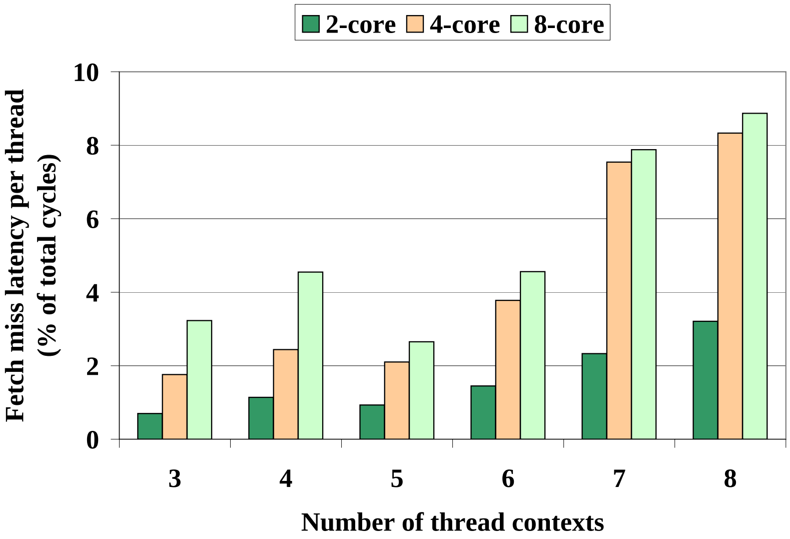


Figure 5.19 L2 read miss latency per thread for FGMT cores

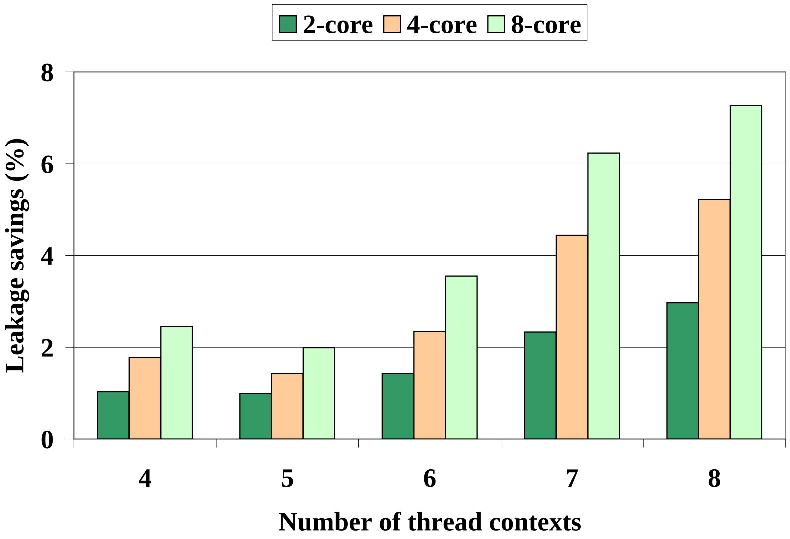


Figure 5.20 Average IRF leakage energy savings for SMT cores

CHAPTER 6

CONCLUSIONS AND FUTURE DIRECTIONS

Energy and power considerations in the world of computing, be that in embedded processors, personal computers, or server farms, is the newest dimension that has challenged computer architects and circuit designers to think of newer design paradigms for such systems. The research efforts reported in this dissertation represent a solid contribution to addressing the issue of reducing leakage energy in both embedded and general-purpose microprocessors. Here we list some future directions to extend the works presented in this dissertation:

- *Investigating more aggressive compiler techniques*: In the works described in Chapter 3 and Chapter 4, the algorithms proposed to identify idle program regions where functional units may be turned off target finding large subgraphs that are enclosed within functions and loops. This is because the breakeven periods of the functional units were two orders of magnitude higher than the processor clock frequency. However, more recently, newer technologies and more sophisticated circuit design techniques have brought down the breakeven periods to 2X-10X of the microprocessor clock period. This makes it important to investivate more aggressive technique at the compiler level to be able to power gate units in a more fine-grained manner.

- *Designing partitioned functional units and register files*: In this work, power gating of the units has been considered at the module level. Either the entire unit is active or is put to sleep for leakage reduction. However, an unit may be designed as partitioned blocks so that one or more of those partitions may be put to sleep when they are idle. This will entail some challenges at the circuit level because (i) partitioned designs usually have more area overhead than non-partitioned designs, and (ii) powering on a circuit block induces noise in the power rails of their neighboring blocks.

- *Modeling full system power*: In this work power modeling has been done only for the specific components that have been targeted to reduce leakage power in. Going forward, it would be important to model entire systems and analyze the power consumption and energy efficiency in the context of operating systems that manage such hardware systems. This includes modeling off-chip caches, if any, graphics processing units (GPUs), interconnection networks, physical memory, secondary storage devices, and other peripheral devices.

- *Supplementing ACPI for power management*: Advanced Configuration and Power Interface (ACPI) [69] specification establishes industry common standard interfaces for operating system (OS)-directed power management of entire systems. Currently, the ACPI specifies active and standby power management of the microprocessor as a single unified unit. However, the techniques presented in this disseration address reducing standyby power in the subcomponents of the microprocessor even when the core itself is active. An exciting future work would be to extend the current standard to incorporate the leakage power reduction techniques presented in this dissertation.

REFERENCES

[1] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient Embedded Computing. *Computer*, 41(7):27–32, 2008.

[2] T. R. Halfhill. MIPS Threads the Needle. *Microprocessor Report*, 20(2):1–8, 2006.

[3] E. Grochowski and M. Annavaram. Energy per Instruction Trends in Intel Microprocessors. *Technology@Intel Magazine*, pages 1–8, 2006.

[4] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19:23–29, 1999.

[5] F. J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, 1999.

[6] E. Grochowski et al. Best of Both Latency and Throughput. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 236–243, 2004.

[7] A. Sodan et al. Parallelism via Multithreaded and Multicore CPUs. *IEEE Computer*, 43(3):24–32, 2010.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Implementation of an 8-core, 64-thread, Power-Efficient SPARC Server on a Chip. *IEEE JSSC*, 43(1):6–20, 2008.

[10] L. Seiler et al. Larrabee: A Many-Core X86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[11] MIPS. MIPS32 1004KTM CPU Family Software Users Manual. *http://www.mips.com*, 2009.

[12] J. Rabaey. *Low Power Design Essentials*. Springer, 2009.

[13] R.K. Krishnamurthy, S.K. Mathew, M.A. Anders, S.K. Hsu, H. Kaul, and S. Borkar. High-Performance and Low-Voltage Challenges for Sub-45nm Microprocessor Circuits. *Intl. Conf. ASIC*, pages 283–286, 2005.

[14] S. Rusu et al. A 65nm Dual-Core Multithreaded Xeon Processor with 16MB L3 Cache. pages 17–25, 2007.

[15] S. Li et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480. ACM, 2009.

[16] D. Duarte, Y.F. Tsai, N. Vijaykrishnan, and M.J. Irwin. Evaluating Run-Time Techniques for Leakage Power Reduction. *Proc. 7th ASP-DAC*, pages 31–38, 2002.

[17] M. Johnson, D Somasekhar, and K. Roy. Models and Algorithms for Bounds on leakage in CMOS Circuits. *IEEE TCAD Integrated Circuits and Systems*, pages 714–725, 1999.

[18] S. Narendra, V. De, D. Antoniadis, A. Chandrakasan, and S. Borkar. Scaling of stack effect and its application for leakage reduction. In *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 195–200, New York, NY, USA, 2001. ACM.

[19] Y. Ye, S. Borkar, and V. De. A New Technique for Standby Leakage Reduction in High-Performance Circuits. *Digest of Technical Papers, Symp. on VLSI Circuits*, pages 40–41, 1998.

[20] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. *IEEE International Conference on Electronics, Circuits and Systems*, pages 167–173, 1998.

[21] J.T. Kao and A.P. Chandrakasan. Dual-Threshold Voltage Techniques for Low-Power Digital Circuits. *IEEE J. of Solid-State Circuits*, 35:1009–1018, 2000.

[22] S. Mutoh, T. Douskei, Y. Matsuya, T. Aoki, S Shigematsu, and J. Yamada. 1-V Power Supply High-Speed Digital Circuit Technology with Multi-Threshold Voltage CMOS. *IEEE J. of Solid-State Circuits*, pages 847–854, 1995.

[23] Z. Hu et al. Microarchitectural Techniques for Power Gating of Execution Units. *International Symposium on Low Power Electronics and Design*, pages 32–37, 2004.

[24] W. Buchholz. *Planning a Computer System: Project Stretch*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1962.

[25] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS '64 (Fall, part II): Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, pages 33–40, New York, NY, USA, 1965. ACM.

[26] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors, 1st ed.* McGraw-Hill Science/Engineering/Math, 2004.

[27] T. Ungerer, B. Robič, and J Šilc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[28] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *International Symposium on Computer Architecture*, pages 240–251, 2001.

[29] K. Flautner et al. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *International Symposium on Computer Architecture*, pages 148–157, 2002.

[30] L. Clark, S. Demmons, N. Deutscher, and F. Ricci. Standby Power Management for a 0.18μm Microprocessors. *International Symposium on Low Power Electronics and Design*, pages 7–12, 2002.

[31] S. Dropsho, V. Kursun, D.H. Albonesi, S. Dwarkadas, and E.G. Friedman. Managing Static Leakage Energy in Microprocessor Functional Units. *International Symposium on Microarchitecture*, pages 321–332, 2002.

[32] W. Zhang et al. Compiler Suppport for Reducing Leakage Energy Consumption. *Design Automation and Test in Europe*, pages 1146–1147, 2003.

[33] S. Rele et al. Optimizing Static Power Dissipation by Functional Units Superscalar Processors. *International Conference on Compiler Construction*, pages 261–274, 2002.

[34] Y. You, C. Lee, and J.K. Lee. Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors. *ACM Transactions on Design Automation of Electronic Systems*, pages 147–164, 2006.

[35] N. Seki et al. A Fine-Grain Dynamic Sleep Control Scheme in MIPS R3000. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 612–617, 2008.

[36] N. Komoda et al. Compiler Directed Fine Grain Power Gating for Leakage Reduction in Microprocessor Functional Units. *Workshop on Optimizations for DSP and Embedded Systems*, pages 42–51, 2009.

[37] S. Roy, N. Ranganathan, and S. Katkoori. A Framework for Power Gating Functional Units in Embedded Microprocessors. *IEEE Transactions on VLSI Systems*, 17:1640–1649, 2009.

[38] S. Rusu et al. Power Reduction Techniques for an 8-core Xeon Processor. In *ESSCIRC, 2009. ESSCIRC '09. Proceedings of*, pages 340–343, 2009.

[39] R. Kumar and G. Hinton. A Family of 45nm IA Processors. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 58–59, 2009.

[40] T. Saito et al. Design of Superscalar Processor with Multi-Bank Register File. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 3507–3510, 2005.

[41] A. Agarwal, R. Kaushik, and R.K. Krishnamurthy. A Leakage-Tolerant Low-Leakage Register File with Conditional Sleep Transistor. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 241–244, 2004.

[42] J. Lingling et al. Reduce Register Files Leakage Through Discharging Cells. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 114–119, 2006.

[43] H. O. Kim et al. Supply Switching with Ground Collapse for Low-Leakage Register Files in 65-nm CMOS. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(3):505–509, 2010.

[44] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE Annual Workshop on Workload Characterization*, pages 3–14, 2001.

[45] M. Leeser and X. Wang. Variable Precision Floating Point Division and Square Root. *Workshop HPEC*, pages 47–48, 2004.

[46] W. Zhao and Yu. Cao. Predictive Technology Model for Nano-CMOS Design Exploration. *ACM JETC*, 3:1–17, 2007.

[47] D. Burger and T. Austin. The Simplescalar Tool Set, version 2.0. Technical report, Tech. Rep. TR-97-1342, University of Wisconsin-Madison, 1997.

[48] S. Roy, N. Ranganathan, and S. Katkoori. Exploration of Compiler Optimization Techniques for Enhancing Power Gating. *International Symposium on Circuits and Systems*, pages 1004–1007, 2009.

[49] M.N. Wegman and F.K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, pages 231–236, 1991.

[50] L Rolaz. An Implementation of Lazy Code Motion for Machine SUIF. Technical report, Swiss Federal Institute of Technology, 2003.

[51] P. Briggs and T.J. Harvey. Multiplication by Integer Constants. Technical report, Rice University, 1994.

[52] K.D. Cooper, L.T. Simpson, and C.A. Vick. Operator Strength Reduction. *ACM Transactions on Programming Languages and Systems*, pages 603–625, 2001.

[53] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and Wu Ye. Influence of Compiler Optimizations on System Power. *IEEE Transactions on VLSI Systems*, 9:801–804, 2001.

[54] GNU Project. GCC, the GNU Compiler Collection. *http://gcc.gnu.org/*.

[55] D. Burger and T. Austin. The Simplescalar Tool Set, version 2.0. Technical report, TR-97-1342, University of Wisconsin-Madison, 1997.

[56] C Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *International Symposium on Microarchitecture*, page 330, 1997.

[57] R. Morgan. Building and Optimizing Compiler. *Digital Press*, 1998.

[58] T. Granlund and P. Montgomery. Division by Invariant Integers using Multiplication. *Proc. ACM SIGPLAN, Conf. on PLDI*, pages 61–72, 1994.

[59] GNU Project. GNU Binutils. *http://www.gnu.org/software/binutils/*.

[60] GNU Project. GNU C Library. *http://www.gnu.org/software/libc/*.

[61] GNU Project. GNU Compiler Collection (GCC) Internals. *http://gcc.gnu.org/onlinedocs/gccint/*.

[62] D. Novillo. GCC Internals. *http://www.airs.com/dnovillo/*, 2007.

[63] H. Singh et al. Enhanced Leakage Reduction Techniques using Intermediate Strength Power Gating. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(11):1215–1224, 2007.

[64] J.E. Stine et. al. FreePDK v2.0: Transitioning VLSI Education Towards Nanometer Variation-Aware Designs. In *Microelectronic Systems Education, 2009. MSE '09. IEEE International Conference on*, pages 100–103, 2009.

[65] Nangate. Nangate 45nm Open Cell Library. *http://www.nangate.com/openlibrary*, 2008.

[66] N. L. Binkert et al. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[67] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2002.

[68] J. Burns and J. L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Trans. Parallel Distrib. Syst.*, 13(2):142–155, 2002.

[69] Phoenix Technologies Ltd. Toshiba Corp. Hewlett-Packard Corp., Intel Corp. Microsoft Corp. Advanced Configuration and Power Interface Specification, Revision 4.0a. *http://www.acpi.info/*, 2010.

LIST OF PUBLICATIONS

- S. Roy, S. Katkoori, N. Ranganathan. A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors. *International Conference on VLSI Design*, January 2007, Page(s): 215 - 220.

- S. Roy, N. Ranganathan, S. Katkoori. Exploration of Compiler Optimization Techniques for Enhancing Power Gating. *International Symposium on Circuits and Systems*, May 2009, Page(s): 1004-1007.

- S. Roy, N. Ranganathan, S. Katkoori. Compiler Directed Power Gating in Embedded Microprocessors. *International Conference on Computer Design*, October 2009, Page(s): 35-40.

- S. Roy, N. Ranganathan, S. Katkoori. A Framework for Power-Gating Functional Units in Embedded Microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 17, Issue 11, November 2009, Page(s): 1640-1649.

- S. Roy, N. Ranganathan, S. Katkoori. Impact of Compiler Optimization Techniques on Power Gating for Leakage Reduction. *IEEE Transactions on Computers*, *Revised manuscript under review*, 2010.

- S. Roy, N. Ranganathan, S. Katkoori. State-Retentive Power Gating of Register Files in Multi-core Processors. *IEEE Transactions on Computers*, *Under review*, 2010.

ABOUT THE AUTHOR

Soumyaroop Roy received his Bachelor of Engineering degree in Electronics and Communication Engineering in 2001 from Birla Institute of Technology, Mesra, Ranchi, India and his Master of Science degree in Computer Engineering in 2006 from University of South Florida (USF), Tampa, FL. He is currently pursuing his Doctoral degree in Computer Science and Engineering at USF and has accepted a Senior Design Engineering position in the Architecture Performance Modeling group at AMD, Austin, TX. His research interests are in architecture and compiler methodologies for low-power design of microprocessors, architecture level performance and power modeling, and low-power VLSI design. From 2001 to 2004, he was a Software Engineer with the NCVHDL group at Cadence Design Systems India in Noida. He has taught several courses at the Computer Science and Engineering Department at USF, including Operating Systems, Data Structures, Foundations of Engineering, and Logic Design, and has served as a teaching assistant in numerous other courses. He received a Provost's Commendation for Outstanding Teaching by a Graduate Teaching Assistant at USF in 2010 and the 2010 Sypris Best Teaching Assistant Award at the Department of Computer Science and Engineering, USF. He is also a recipient of the 2009 IEEE Computer Society Richard E. Merwin scholarship. He is a student member of the IEEE and the IEEE Computer Society.