APPLICATION-SPECIFIC ARCHITECTURE FRAMEWORK FOR HIGH-PERFORMANCE LOW-POWER EMBEDDED COMPUTING

by

Allen Chao-Hung Cheng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2006

Doctoral Committee:

Associate Professor Gary S. Tyson, Co-Chair
Professor Trevor N. Mudge, Co-Chair
Professor Marios C. Papaefthymiou
Associate Professor Dennis M. Sylvester
Assistant Professor Scott A. Mahlke

To My Beloved Wife,
Chia-Lin Chang

For her unfailing love, tender care, and steady support.

ACKNOWLEDGMENTS

Whenever acknowledgments are given, there is a danger of leaving someone out or forgetting something they have done. Many people contributed to the success of this dissertation thesis. While I would like to acknowledge individually each by name, I would inevitably leave out deserving colleagues, friends, and relatives. I apologize in advance for such omissions and convey my sincerest respect and admiration to all who contributed to the extraordinary experiences I have been fortunate to enjoy. I would like to give thanks to those who have been close to me over the years, first in familial relationships, then in professional ones.

First and foremost, I thank God – Father, Son and Spirit - for His unconditional love, guidance, sustenance, and strength. He has enabled me to do this work.

My family has been a tremendous source of love, encouragement and inspiration. Their support kept me going not only through this thesis work but through my entire life.

Chia-Lin Chang, my dear wife, who has also been my best friend and best supporter to me for the years I have spent in Michigan. Chia-Lin, I thank you for bringing me to the grace of God. I also thank you for faithfully staying besides me during the most difficult and confusing time of my life. Without you, this work can never be finished as fast and well. I love you so much!

Beyond family support, I am deeply indebted to my advisors Gary Tyson and Trevor Mudge, who have constantly supported me and to my work during these years.

Gary, your pointed questions and lucid explanations were invaluable help to me while working on my dissertation. You are the constant source of my inspiration, even when things do not seem so inspirable. Your optimism and ability to motivate me have kept me stay on the course during ups and downs are amazing! Gary, you are my advisor, my mentor, and even more, a great friend, who carried me through the dark times when I was in need.

Trevor, you are my best advocate both in and out of Michigan for me and for this work. You are my advisor who always stands by me and speaks for my best interests whenever I have a battle to fight. You are the source of the steadiness when everything seems so unsteady. You always seem to have the wisdom and experience to simplify situations and things that seem so complicated to me. Trevor, you are a true wonder! Your laid-back management style is legendary: you grant me all the freedom I need to do my work, and still manage to gently push me forward without causing unnecessary tension and stress. It was really a joy ride!

Thanks to Marios Papaefthymiou, Scott Mahlke, and Dennis Sylvester for serving on my doctoral committee.

Marios, I thank you for being open and sharing with me your wonderful experience and perspectives as a faculty in Michigan. You are on my thesis committee, although interestingly a lot of our interactions come from teaching 370, which I enjoyed very much. Your incredible sense of humor and jokes made supposed-to-be-taxing staff meetings much more like stress relievers.

Dennis, I thank you for contributing your hardcore EE knowledge and expertise that helped me decipher the cryptic standard cell libraries and fortify my hardware synthesis analyses.

Scott, I thank you for teaching me everything about compilers that I ever wanted to know. This valuable compiler knowledge will enable me to expand the horizon of this dissertation work to the next level.

Thanks to Karem Sakallah, Toby Torey, Elliot Solloway, Don Windsor, and Karen Langona for all the wonderful role modeling, mentoring and the joyful interaction that I am fortunate to receive while working as your GSI.

Last but not least, thanks to EECS and ACAL staff Karen, Dawn, Bertha, and Denise. You are such great resources who helped me navigate my graduate school career. You provided everything I needed for a smooth graduation.

As I cross the major milestone of my life, it is such a warm and pleasant thought to share this day that bears the fruits of the difficult battle I fought with all those who touched my life in memorable ways. I thank you all, fore-mentioned or not, for guiding me with your wisdom and knowledge; showering me with your love and tender care; inspiring me through your indelible influences and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

ABSTRACT

APPLICATION-SPECIFIC ARCHITECTURE FRAMEWORK FOR HIGH-PERFORMANCE LOW-POWER EMBEDDED COMPUTING

by

Allen Chao-Hung Cheng

Co-Chairs: Gary S. Tyson and Trevor N. Mudge

The design space of embedded systems is extremely large. Examples of embedded systems range from small form factor portable handheld devices such as smart phones, MP3 players, and personal digital assistants (PDA), to real-time control systems used in automobiles and the space shuttle. These embedded applications require a new architecture paradigm with strict requirements on power consumption, computing performance, competitive end-user price, and rapid time to market (TTM). Thus, when designing microprocessors for embedded systems, it is extremely important to consider energy efficiency, performance, production cost, and design turnaround time.

This dissertation introduces Framework-based Instruction-set Tuning Synthesis (FITS), an architectural and microarchitectural innovation that addresses all the above design constraints of embedded microprocessors. FITS reduces energy consumption by

running same applications with much smaller code size and improved locality. This is accomplished through tailoring the instruction set to the requirements of a targeted application. Smaller code size with better locality means it is possible to replace original instruction caches with smaller ones that can still yield better cache miss rates. Smaller instruction cache consumes less dynamic and leakage power. Lower cache miss rates yield less traffic from the processor to off-chip memories, which can improve both performance and power consumption.

Making new chips can be both an economical challenge and a time-consuming process. Fabricating a new chip can incur millions of dollars of non-recurring engineering (NRE) cost. The average turnaround time for fabricating a new chip ranges from several months to several years. FITS reduces the chip production cost and shortens the design turnaround time through the use of a general-purpose, functionally-rich underlying microarchitecture. Instead of having to fabricate a new chip every time there is a new application to target, we use a single microarchitecture platform that contains enough general-purpose functionality that can map to all applications of interest. The one-time high NRE cost and long turnaround design cycle can be effectively amortized through mass production over the period of product lifetime.

FITS improves the performance by introducing the Versatile Integrated Processing (VIP) unit and integrating a Zero-Overhead Loop Execution (ZOLE) unit into the microarchitecture. The VIP unit is a universal data-crunching engine that delivers superb data computing and data streaming performances. The ZOLE unit streamlines the program control flow by removing expensive loop control overhead from both nested and non-nested loops.

The application-specific instruction set tailoring is achieved by replacing the fixed instruction decoder of general-purpose embedded processors with a programmable decoder. The use of a programmable decoder allows designers to add new capabilities to microarchitecture without being restricted by the limited instruction space. The only constraint of adding new operations is due to chip area. The net effect is that the underlying microarchitecture may contain an extremely large set of operations that can never be mapped to any single instruction set architecture (ISA); yet, through the use of a programmable instruction decoder, FITS can choose the needed subset of operations being mapping to the premier instruction space for a given application. The instruction selection is determined at compile time. The definition of ISA is loaded to the programmable decoder at boot time. If necessary, the programmable decoder can be dynamically reconfigured with different set of ISA definitions at run time.

Through the use of programmable decoder, general-purpose microarchitecture equipped with VIP and ZOLE, FITS provides designers with a new genre of embedded microprocessors that can achieve application-specific processor performance and low energy consumption, while maintaining the fabrication advantages of a mass-produced single-chip solution that yields low production cost and fast time to market.

CHAPTER 1

INTRODUCTION

1.1 Motivation and Background

The design space of embedded systems is extremely large. Examples of embedded systems range from small form factor portable handheld devices such as smart phones, MP3 players, personal digital assistants (PDA), and digital cameras and camcorders, to real-time control systems used in automobiles and the space shuttle. These embedded applications require a new architecture paradigm with strict requirements in computing performance, energy efficiency, competitive end-user price, and rapid time to market (TTM). Thus, when designing microprocessors for embedded systems, it is extremely important to consider performance, power consumption, production cost, and design turnaround time. This trend necessitates a new platform of innovations leveraging novel architectural, and microarchitectural techniques.

An emerging popular strategy to meet the challenging cost, performance, and power demands is to move away from general-purpose designs to application-specific designs. An application-specific processor (ASP) is a processor designed for a particular

application or set of applications that share many common characteristics. Thus, an ASP design contains only those capabilities necessary to execute its targeted workload. The result is that ASPs can achieve levels of performance and efficiency that are unattainable in general-purpose processors. The performance gain of ASP is highly valued for modern embedded workloads, which are well known of their ever increasing processing requirements demanded by endless user computing needs.

With wide-spread use of Intellectual Property (IP) cores and advancement in electronic design automation (EDA) tools, customized instruction set synthesis has become a feasible option to make products stand out in competitive consumer electronics market today. Generally speaking, modern embedded microprocessors have many strict design constraints that touch many facets of application requirements, such as processing speed, energy efficiency, chip area, code size, production cost, and design turnaround time, etc. Designers of contemporary general-purpose machines with 32-bit instructions are struggling to achieve even the minimal satisfactory balance among these design requirements.

This dissertation proposes Framework-based Instruction-set Tuning Synthesis (FITS): an architectural and microarchitectural innovation that addresses all the above design constraints of embedded microprocessors.

1.2 Addressing Performance Issue

FITS improves the performance by integrating proposed Versatile Integrated Processing (VIP) unit and a Zero-Overhead Loop Execution (ZOLE) unit into the

microarchitecture. The ZOLE unit streamlines the program control flow by removing expensive loop control overhead from both nested and non-nested loops. The VIP unit is a universal data-crunching engine that delivers superb data computing and data streaming performances. The area cost of adding new operations using VIP is extremely low: for every additional VIP unit added, the number of additional operations available will increase exponentially. Furthermore, because VIP is synthesized in standard cells and chaining each extra VIP only costs few multiplexers, we can implement thousands of new specialized operations using less area than it would take to configure a single operation using programmable circuits like FPGAs, and would result in faster circuit speeds.

The key difference between our proposed VIP and other approaches of customized reconfigurable function units is in how the large potential instruction space of specialized chained operations is mapped to the instruction set. Other reconfigurable function units reserve a single opcode which specifies the data dependencies to whichever configuration is programmed into the customized accelerator, while the decoupled instruction set provided in our underlying FITS architecture allows all function permutations to be mapped into the instruction set architecture (ISA) without necessitating additional opcode space. A conventional ISA could not support the wide range of different functions that can be configured since the number of total instructions will quickly grow into the order of thousands just with 2 levels of chained function units and it will be well past that with 3 or more levels of chained functional units. Prior work configures the programmable circuit with the chained functions that the compiler or designers found useful, while the programmable nature of the FITS instruction decoder enables the microarchitecture to implement a fixed circuit capable of executing any of the

function permutations; we simply map the one or more permutations that the application requires to one of the instructions in the FITS ISA.

While it may seem that implementing the circuit to perform all permutations would require much greater area than that of a programmable circuit would require, that is not the case. Since each permutation differs in only the control signals going to the multiplexers at each level of the chained function unit design; it is not area limitations that prevent the design of these chained function units (since the area requirement for implementing any one circuit are only slightly smaller than implementing all permutations), but it is the tremendous increase in operations that can be specified in the microarchitecture and the corresponding increase required in the opcode for conventional ISA that is the true limitation. Customized reconfigurable function units provide one method of avoiding the limitation (programmable circuits), while VIP/FITS provides a more flexible method (programmable instruction decode).

Another aspect of performance improvement comes from the custom synthesized application-specific ISA tailored to the requirements of a given application. The application-specific instruction set tailoring is achieved by replacing the fixed instruction decoder of general-purpose embedded processors with a programmable decoder. The use of a programmable decoder allows designers to add new capabilities to microarchitecture without being restricted by the limited instruction space. The only other constraint of adding new operations is due to chip area, which has been addressed by the space-efficient VIP unit.

The net effect is that the underlying microarchitecture may contain an extremely large set of operations that can never be mapped to a small fixed width ISA. Yet, through

the use of a programmable instruction decoder, FITS can choose the needed subset of operations being mapping to the premier instruction space for a given application. The instruction selection is determined at compile time. The definition of ISA is loaded to the programmable decoder at boot time. The programmable decoder can be dynamically reconfigured with different set of ISA definitions at run time, if necessary.

One other major advantage of using the programmable decoder is the benefit of decoupling the microarchitectural enhancements from the ISA so that new instructions can be integrated into the underlying microarchitecture, as much as the chip area goal permits, without being restricted by limited opcode space nor being crippled with bigger instruction decoders. Designers are free to include additional functional capabilities to improve performance, even when those enhancements are useful for only a small percentage of applications since the inclusion of one operation does not require the elimination of another to fit in the instruction set encoding space.

1.3 Addressing Power Consumption Issue

Power consumption is now a leading design constraint in microprocessor designs, especially in low-end embedded system market [Mudge01]. In addition to costly heat removal expense, excessive power consumption in embedded devices also reduces the battery lifetime. As a result, the quality and reliability of an embedded device would be severely compromised by high power dissipation. With battery power density increasing only at a rate of approximately 5% per year, any significant extension of battery lifetime must come from a thorough improvement of energy efficiency for each power-hungry

component in a system. Among other system components, memory structures, such as caches, register files, TLBs, BTBs, etc., are by far the most predominant source of power dissipation on the processor. For instance, in Intel's StrongARM processor, caches consume more than 40% of total chip power with 27% being devoted to the instruction cache [Montanaro96]. This dissertation presents a novel ISA synthesis technique that could reduce significant instruction cache power loss.

FITS reduces energy consumption by running same applications with much smaller code size and improved locality as a result of half-width ISA. The philosophy of FITS is that high performance and high code density can co-exist if we can match the instruction set to the requirement of a targeted application. FITS improves code density by utilizing instructions that are only 16-bit instead of 32-bit that are commonly used in most conventional machines. Since the instruction width is reduced by half, the total code size can be reduced by half as long as what was originally done in a single 32-bit instruction can also be done in a single 16-bit instruction. To best utilize the half-sized instruction width, the instruction space is allocated to only those operations that are necessary and useful to the given application. The chapter of results and analyses shows that FITS can achieve a code size reduction that is close to 50% with better performance through application-specific customization.

Half-sized program with better locality means it is possible to replace original instruction caches with those that are only half big and still can yield better cache miss rates. Smaller instruction caches with better hit rates can save both dynamic and static power consumption. Better cache hit rates also means less traffic from the processor to off-chip memories, which can further improve both performance and power consumption.

1.4 Addressing Cost and Time to Market Issues

Making new chips can be both an economical challenge and a time-consuming process. Fabricating a new chip can incur millions of dollars of non-recurring engineering (NRE) cost. The average turnaround time for fabricating a new chip ranges from several months to several years. FITS reduces the chip production cost and shortens the design turnaround time through the use of a general-purpose, functionally-rich underlying microarchitecture. Rather than fabricating a new chip to map each new application, we choose a single general-purpose microarchitecture platform augmented with VIP and ZOLE units, so there are an extremely large set (i.e. in the order of thousands) of operations that can be selected to map the requirements of any application of interests. Because of the nature of this single general-purpose microarchitectural platform, FITS can reduce the chip production cost and shortens time to market by leveraging the fabrication advantages of a mass-produced, single-chip solution that amortizes the one-time high NRE cost and lengthy design turnaround time.

1.5 Thesis Summary

In this dissertation, we proposed an efficient and effective Framework-based Instruction-set Tuning Synthesis (FITS) platform for designing a new class of embedded microprocessors that can effectively address all important design constraints. FITS offers designers an enhanced general-purpose microarchitecture solution with configurable ISA synthesis to tailor the processor to match the requirements of a given application. FITS

delays instruction set synthesis until after chip fabrication. With a fixed microarchitecture, synthesis is performed by replacing the fixed instruction decoder with a programmable decoder. With a programmable decoder, designers can add new capabilities to microarchitecture without being restricted by the limited instruction space. The underlying datapath of a FITS processor would be similar to a general-purpose embedded processor such as ARM and is enhanced with VIP and ZOLE units that can result in thousands of extra specialized operations. The total available instructions are extremely large that may never be mapped to any single ISA. Through the use of a programmable instruction decoder, designers can map only a subset of this large set of instructions to the synthesized ISA. By only mapping those operations that a particular application needs to the synthesized instruction set, it is possible to encode all instructions in a short, 16-bit format while retaining all of the special purpose operations that can ever be found in any large instruction embedded processor.

The contributions of this dissertation research are threefold. First, we improve performance with reduced code size by synthesizing specialized 16-bit VIP and ZOLE instructions that can accelerate performance for full range of embedded applications, which would normally require 32-bit instructions. Secondly, we improve energy efficiency from a much dense, half-sized code of high locality, which requires smaller instruction caches and has less off-chip memory traffic. More energy can be conserved by deactivating those parts of the datapath that are not mapped to any execution of the synthesized ISA. Lastly, we reduce production cost and time to market by utilizing a single processor platform across a wide range of applications, while retaining the ability to optimize the ISA for the individual requirements of each application.

In summary, through the use of programmable decoder, general-purpose microarchitecture equipped with VIP and ZOLE, this dissertation research provides designers with a new genre of embedded microprocessors that can achieve application-specific processor performance and low energy consumption, while maintaining the fabrication advantages of a mass-produced single-chip solution that yields low production cost and fast time to market.

1.6 Thesis Organization

The remainder of this dissertation thesis is organized as follows: Chapter 2 analyzes and describes important characteristics of embedded workload. Chapter 3 presents the FITS design framework methodology and the architectural and microarchitectural innovations that support it. Chapter 4 explains experimental set-up procedures. Chapter 5 discusses the experimental results and provides detailed analyses for benefits of FITS. Chapter 6 discusses related work. Chapter 7 offers conclusions and future directions of this dissertation research.

CHAPTER 2

WORKLOAD ANALYSIS

This section presents important characteristics of embedded applications in terms of their requirements in opcode space, operand space, immediate space, and physical register space. A representative subset of the MiBench suite [Guthaus01] programs have been compiled into the ARM binary using the GCC tool chain [GCC04] and simulated using the SimpleScalar tool set [Austin02] to provide both static and dynamic profiling statistics. The results and analyses of each requirement are discussed in each of the sub-sections respectively.

2.1 Opcode Space Requirement

The opcode space in an ISA specifies the number of different instructions or functional capabilities a processor may perform. If an application need to perform many different instructions, a large number of instruction bits need to be allocated for opcodes.

Figure 2.1 shows the number of distinct opcodes that were executed throughout the life time of a program. The 100% bar on the left represents the total number of

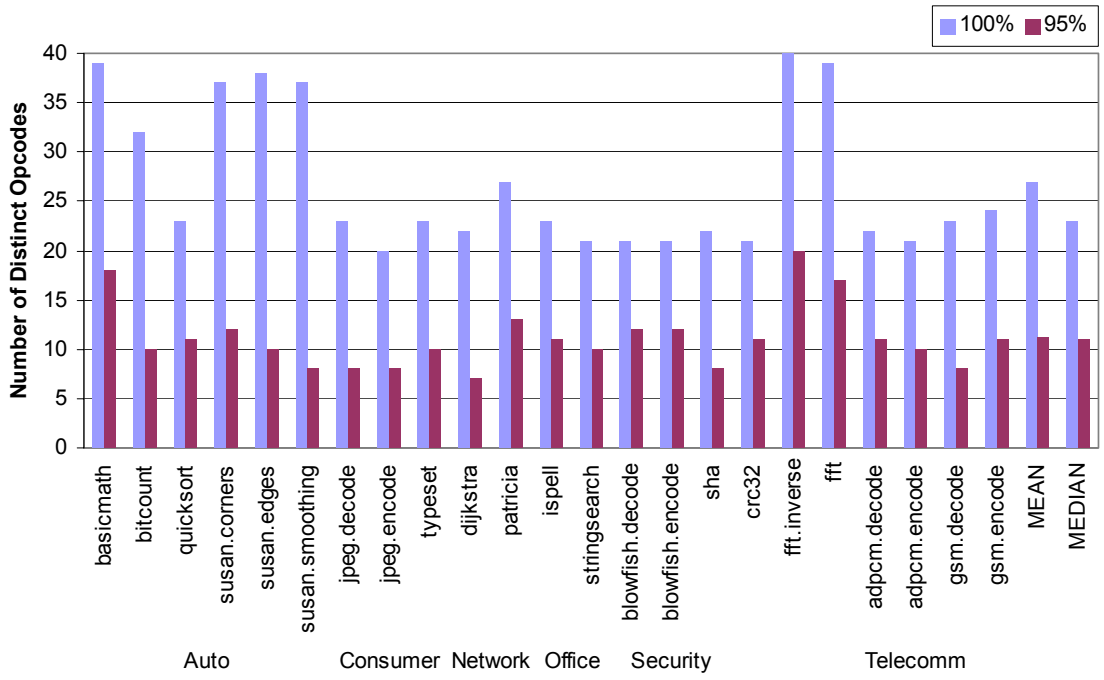

Figure 2.1: Utilization of Distinct Opcodes throughout Program Life Time

opcodes that a program has executed, so they account for 100% of total dynamic instruction frequency. Among these 23 MiBench programs, 16 of them (69.6%) utilize 27 or less opcodes; 7 of them (30.4%) utilize from 32 to 40 opcodes. The 95% bar on the right indicates the number of opcodes needed to account for 95% or above of total dynamic instruction frequency. Ignoring less than 5% of total dynamic instructions reduces the opcode requirement significantly: 20 out of 23 programs need at most 13 opcodes, while the highest opcode demand does not exceed 20.

The reason for the significant reduction in opcode requirement is because not all opcodes are demanded equally frequently. This unbalanced utilization of opcodes is illustrated in Figure 2.2. This figure shows the percentage of under-utilized opcodes as a fraction of total number of opcodes has been executed. An opcode is “under-utilized” if it account for less than 1% of total dynamic instruction frequency. As shown in the figure,

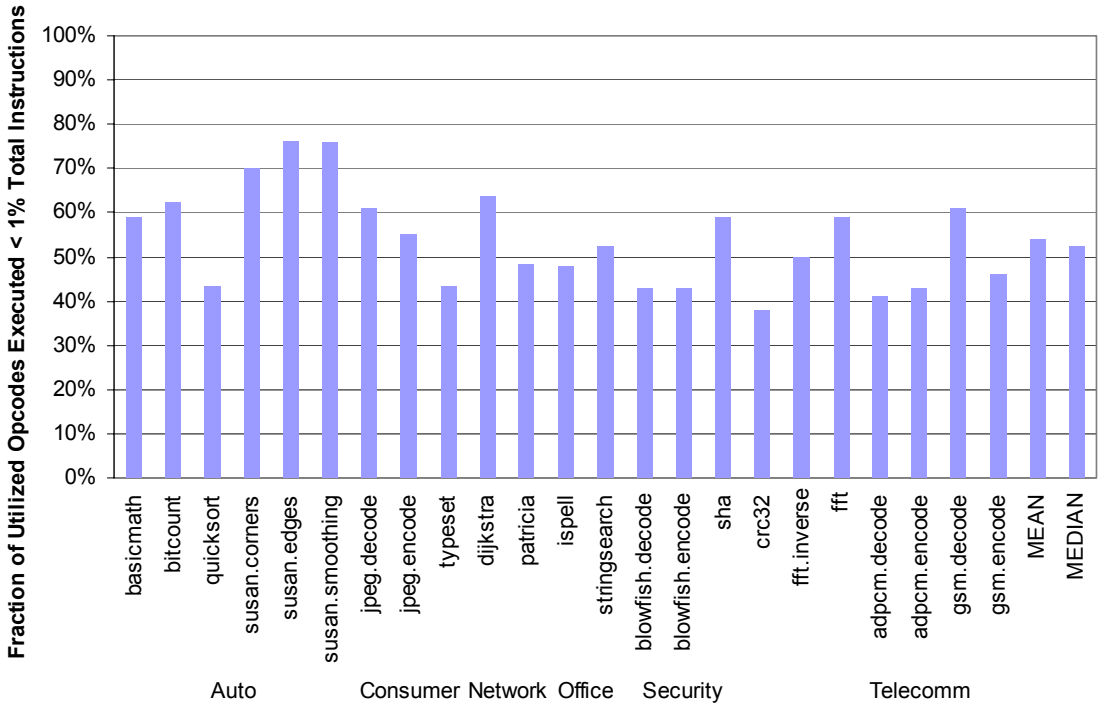

Figure 2.2: Percentage of Under-utilized Opcodes

many opcodes are rarely executed: on average, 55.6% of opcodes do not contribute to more than 1% of total dynamic instructions. Rather than mapping these infrequently executed opcodes onto the ISA space, we can emulate them in software to save the instruction space without affecting performance significantly. A software emulated instruction is often translated into one or more "real" machine instructions as specified in the ISA.

2.2 Operand Space Requirement

The number of explicit distinct register operands, or the register address mode, is another important parameter to be considered for designing a cost-effective ISA design. Three-address instructions, or instructions that have three register operand fields, prevail

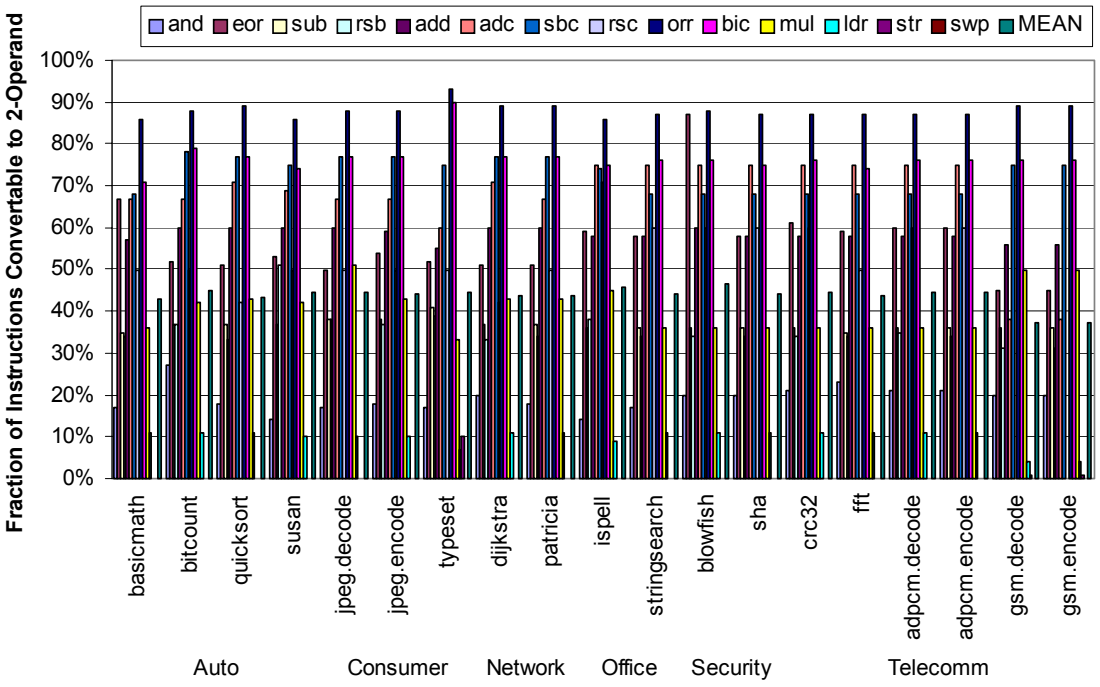

Figure 2.3: 2-Address Compatible Static Instruction Distribution

in many popular 32-bit ISA designs. Despite many advantages of having three explicit register operand fields, our preliminary studies showed that two register operands are often enough. This section shows the results of our static and dynamic examination on how often two-address instructions suffice in a program. Static statistics are important from a code size viewpoint, and dynamic statistics help us gauge power dissipation.

2.2.1 Static Profiling Operand Analysis

Figure 2.3 is a static address mode profiling results for compiled ARM binaries. It illustrates the fraction of all three-address integer instructions that can be satisfied with only two addresses. This is determined, at compile time, by calculating the fraction of total instructions in which the destination register operand is the same as one of the

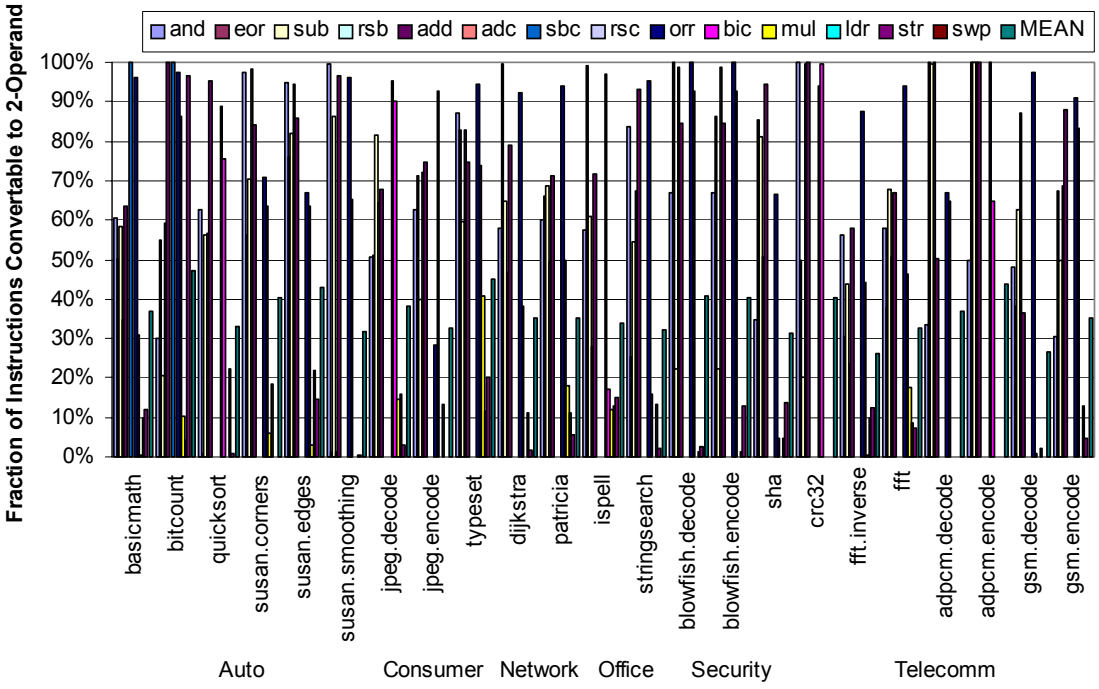

Figure 2.4: 2-Address Compatible Dynamic Instruction Distribution

source register operands. The results are sorted according to the instruction type as shown in the figure. Besides *load*, *store*, and *swap* instructions, the remaining 3-address integer instructions only need two operands 19% to 88% of the time. This result suggests the possibility of intermixing two-address instructions and three-address instructions within an application. This approach trades off the expressive power of an instruction for a compact instruction space.

2.2.2 Dynamic Profiling Operand Analysis

Figure 2.4 is a dynamic address mode profiling results from simulating the compiled ARM binaries on SimpleScalar. Similarly, it illustrates the fraction of all

dynamic three-address integer instructions that can be satisfied with two addresses. This is determined, at run time, by calculating the fraction of total instructions in which the destination register operand is the same as one of the source register operands. These dynamic profiling statistics strengthens the static profiling results by showing a wide applicability of replacing three addresses with two addresses during the actual program execution. The lower distribution are from instructions, which either (1) rarely get executed such as *add with carry*, *subtract with carry*, *reverse subtract with carry*, and *swap*; or (2) tend to use all three operands such as *load*, and *store*. The remaining 3-address instructions only require two register operands 59% to 87% of the time. The *multiply* instruction is rather unpredictable across the benchmark suite and ranges from 0% (e.g. *quicksort*) to 41% (e.g. *typeset*).

2.3 Immediate Space Requirement

ARM instructions use a lot of immediate operands. Clearly, if we are going to have a processor architecture that is similar to ARM, we will have to be capable of handling these immediate values efficiently. In this sub-section, we provide static and dynamic analyses of the space requirement for these immediate operands. To aid our analyses, we classify immediate operands into three categories: branch immediate operands, ALU immediate operands, and memory (load and store) immediate operands. Branch instructions contain immediate operands because ARM uses PC plus offset to calculate the branch target address. ALU instructions use immediate operands because ARM not only does regular arithmetic and logic operations, which often use immediate

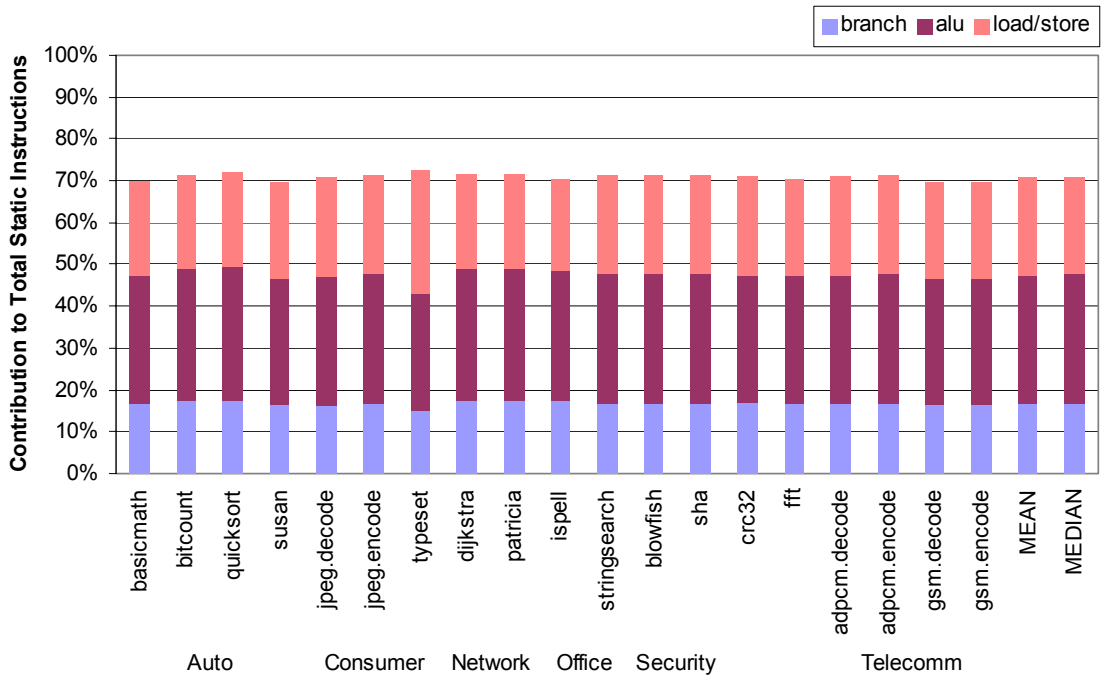

Figure 2.5: Static Distribution of Immediate-Instructions

operands to process data, but it may also do a shift operation followed by these data processing operations and the specification of the shift is an immediate. Memory instructions use immediate values because ARM uses base plus displacement to calculate the effective memory addresses.

2.3.1 Static Profiling Analysis of Immediate Operands

Immediate instructions are instructions that have immediate constants embedded in them. This section provides the static analysis of their characteristics in compiled ARM binaries. Figure 2.5 illustrates two important aspects of immediate operands' usage. First, it shows their uses spread across the entire benchmark suite: on average, 71% of all static instructions contain immediate values. Second, it shows a clear distribution for each

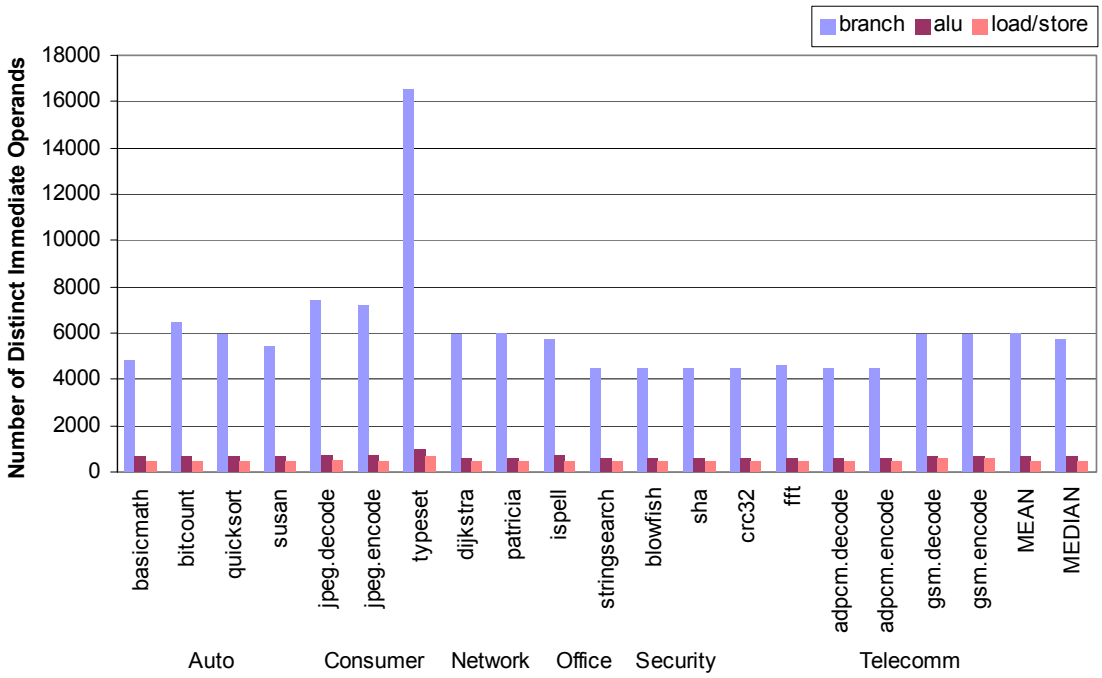

Figure 2.6: Static Utilization of Distinct Immediate Operands

type of immediate instructions within a program: on average, ALU immediate instructions constitute 30.7% of total program size. Memory immediate instructions constitute 23.5% of total program size, and branch immediate instructions constitute 16.8% of total program size.

Static utilization of these immediate operands may help us determine the size of immediate operand in instructions. Figure 2.6 shows the number of unique immediate constants utilized by each category respectively. Despite their small contribution to total code size, branch instructions use the largest number of unique immediate constants and range from 4427 to 16531 with an average at 6020 and a median at 5681. However, all programs except the *typset* use less than 8000 branch immediate operands. The numbers of distinct ALU and memory immediate operands utilized are much smaller than branch

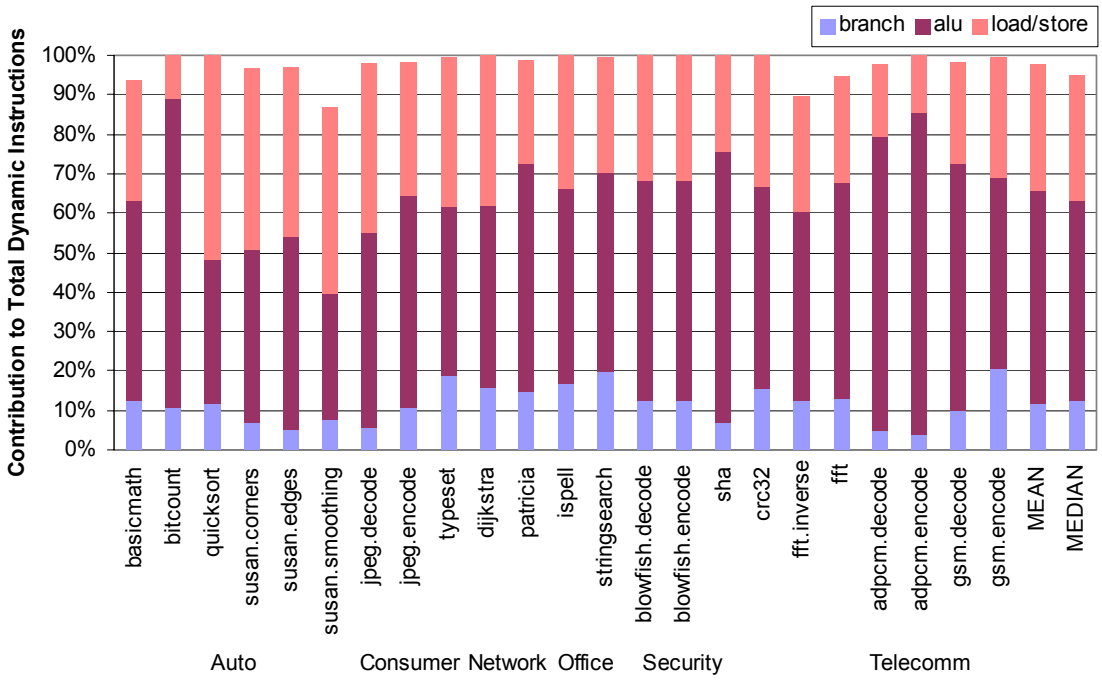

Figure 2.7: Dynamic Distribution of Immediate-Instructions

immediate operands. The ALU immediate operands range from 558 to 956 different values with an average at 648 and a median at 623; while the memory immediate operands range from 422 to 683 different values with an average at 464 and a median at 428.

2.3.2 Dynamic Profiling Analysis of Immediate Operands

One disadvantage of looking at static profiling analysis alone is that we may overshoot the requirement since not all immediate operands are dynamically executed equally frequently. In contrast to the static profiling approach, dynamic profiling allows us to identify the most frequently used immediate constants, and thus enabling us to pinpoint the greatest need of immediate constants and allocate the instructions bits

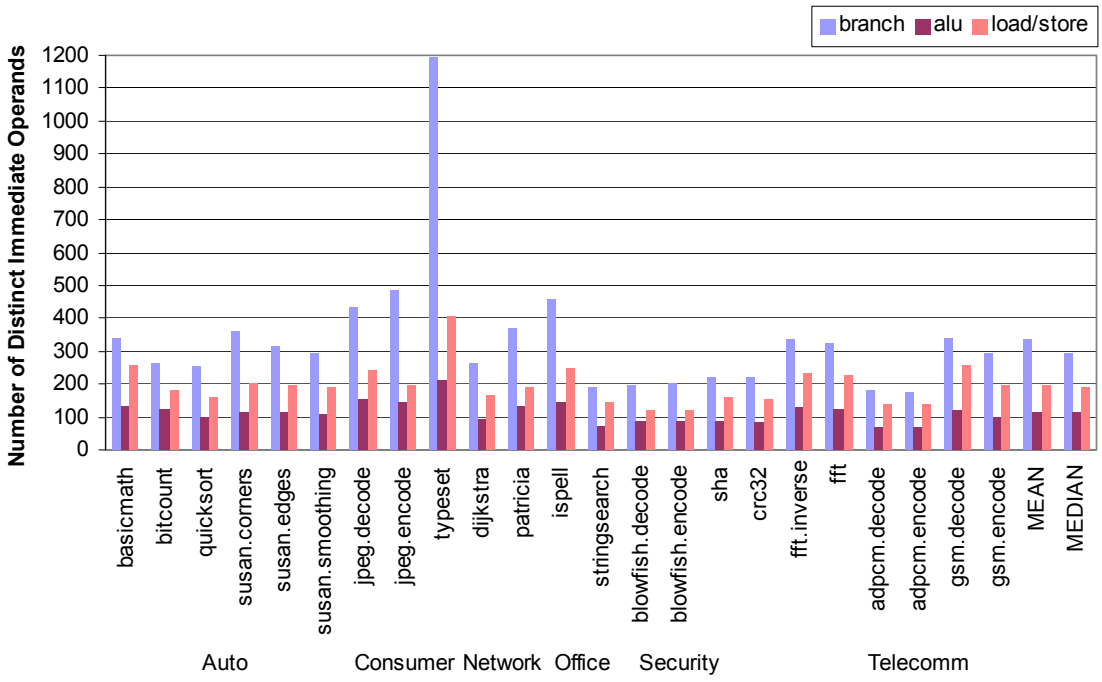

Figure 2.8: Dynamic Utilization of Distinct Immediate Operands

accordingly. Therefore, it is necessary to perform both static and dynamic profiling analyses to get a more balanced view of application execution needs.

Figure 2.7 shows the dynamic distribution of immediate instructions. It strengthens the trend illustrated by Figure 2.5: On average, 97.7% of all executed instructions are immediate instructions. Within this overwhelmingly large fraction, 53.9% are ALU immediate instructions; 32.2% are memory immediate instructions, and 11.7% are branch immediate instructions.

Figure 2.8 shows the number of unique immediate constants utilized by each category respectively. Despite the fact that they have the smallest share of total dynamic instruction counts, branch instructions again use the largest number of unique immediate constants, and range from 117 to 1193 with an average at 335 and a median at 295. The number of unique ALU and memory immediate operands dynamically utilized is again

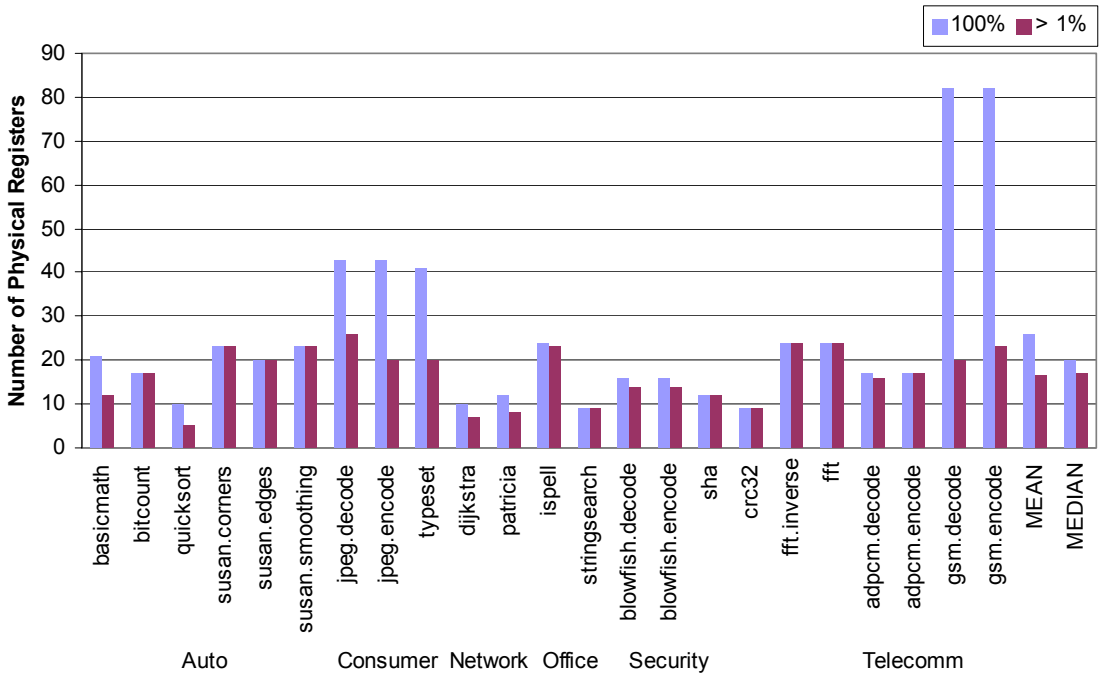

Figure 2.9: Utilization of Physical Registers

smaller than branch immediate operands. The ALU immediate operands range from 63 to 213 different values with both an average and a median at 113. The memory immediate operands range from 121 to 408 different values with an average at 197 and a median at 192.

2.4 Physical Register Space Requirement

The physical register requirement is the maximum number of physical registers necessary without causing memory spills during the register allocation phase of the compiler. This information is closely related to the register operand width in the physical instruction space. The smaller the number of physical registers used by a program, the better the performance of instruction set with short register operand width would have.

To collect the maximum number of physical registers that an application may need, we used the MIRV [MIRV01] compiler to compile and profile MiBench. The results are shown in Figure 2.9.

The profiling is done by examining this requirement at the procedure call level. Specifically, we looked at the register usage for each procedure and determined the maximum usage among all procedures for a given program. We did not look at the register spilling at procedure boundaries because they are less well-defined to be captured. This does not prevent us from obtaining a good overall estimate since most programs nowadays spend majority of their time in function calls one way or the other.

The left bar (100%) shows the maximum physical register requirement among all procedure calls: they range from 9 to 82 with an average at 26 and a median at 20. The right bar (>1%) shows the maximum physical register requirement among those procedures that each individually contributes to more than 1% of total dynamic instruction count; hence it is the number of physical registers a program needs the most: they range from 5 to 26 with both an average and a median at 17. We argue that in order to achieve a better resource utilization, procedures which contribute less than 1% of total executed instructions yet demands many registers (e.g. Main, Init…etc.) should be given less consideration. According to Figure 2.9, 3 programs need 8 or less physical registers; 7 programs need 9-16 physical registers; 13 programs need 17-26 physical registers.

CHAPTER 3

FRAMEWORK DESIGN

This chapter describes the FITS design approach and the framework that supports it. The basic philosophy of FITS is that high performance and high code density can both be achieved if we can match the instruction set to the requirements of a targeted application. FITS improves code density by using only 16-bit instructions instead of the conventional 32-bit instructions. Since the instruction width is reduced by half, the total code size can be reduced by half as long as what was originally done in a single 32-bit instruction can also be done in a single 16-bit instruction. In the chapter of experimental results, we will show that FITS indeed can achieve a code size reduction that is close to 50%. FITS does not trade off performance for code density. Through application-specific customization, FITS can achieve high performance using only 16-bit wide instructions. To best utilize the half-sized instruction width, the instruction space is allocated to only those operations that are necessary and useful to the given application. As a result, we can have a design that has best parts of both worlds: compact code density of 16-bit instructions with high performance of 32-bit instructions.

3.1 Methodology

FITS is an application-specific hardware software co-design approach that matches microarchitectural resources to performance needs of a given application, while improving code-density. FITS does application-specific customization at the instruction set level utilizing programmable decoders for instruction decode and register access. A FITS processor consists of a fairly large set of functional units, including standard ALU operations as well as a set of other useful instructions (e.g. Multiply/accumulate, zero-overhead looping instructions, etc.). Limitations on the functions provided are only due to chip area goals, not instruction set size limits. This can greatly increase the number of similar operations, such as saturating add, because the additional circuitry to add saturation to an add operation is minimal. Since instruction space encoding is decoupled from the underlying microarchitecture, it is possible to add many instructions that may only be useful to a small subset of applications. With a programmable decoder, FITS can tune an ISA to include only those operations necessary for a single application. Moreover, FITS is extremely flexible in terms of the range of underlying microarchitecture that it can work with: from general-purpose DSPs or embedded processors such as ARM to application-specific customized data-path. FITS provides the same level of customization as many ASPs, trading somewhat greater chip area requirements for eliminating the need to synthesize a new chip for each application.

To tune a FITS processor, a FITS aware compiler analyzes the instruction and register requirements of an application, before instruction selection and register allocation. We currently use profile information, but we are exploring new optimization

heuristics using static dataflow information to perform the code transformation. Once code generation is complete, the compiler can specify the register organization and instruction decoding to perform for the application. This configuration information is then downloaded to a non-volatile state in a FITS processor. At this point, the processor instruction set and register file organization is complete. If this application is later upgraded with increased functionality, FITS can re-configure the decoders to match the new requirements of the application. In general, FITS can transform any general-purpose machine into an application-specific processor platform with over-provisioned resources that can be dynamically configured to adept to the needs of different applications.

3.2 System Design Flow

The system design flow of FITS consists of five stages: profile, synthesize, and compile are done off-line; configure and execute are done on-line. As illustrated in Figure 3.1, the targeted application is first analyzed by the FITS profiler to extract its characteristics. The output of the profile stage is a list of extensive requirements analysis related to each element that makes up an instruction set, such as opcode field, operand field, immediate field, and register pressure.

After gathering the profiling information, FITS uses this information as a guideline to synthesize an appropriate instruction set that will satisfy the requirements of a given application. This is the stage where the instruction selection and encoding take place. Instructions are selected based on their referenced frequencies. When the instruction synthesis finishes, the definition of a complete ISA is formed. The FITS

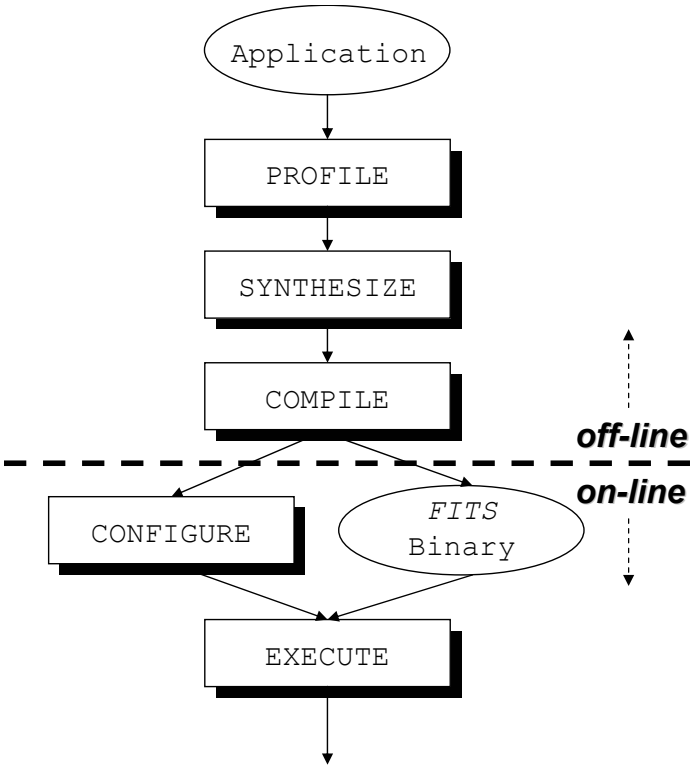

Figure 3.1: System Design Flow of FITS Framework

compiler would then take the instruction set definition to compile the given application into a 16-bit FITS binary. Any unused portions of datapath are turned off to save power consumption [Joseph03]. Up until this point, the instruction synthesis is completed and everything is performed off-line. During the chip initialization, the programmable decoder is configured using the instruction decoding and register organization specified by the compiler. The overhead of this one time configuration is trivially insignificant. Please refer to the chapter on programmable decoder for more details on its initialization. Once everything completes successfully, the compact FITS code is executed without performance degradation.

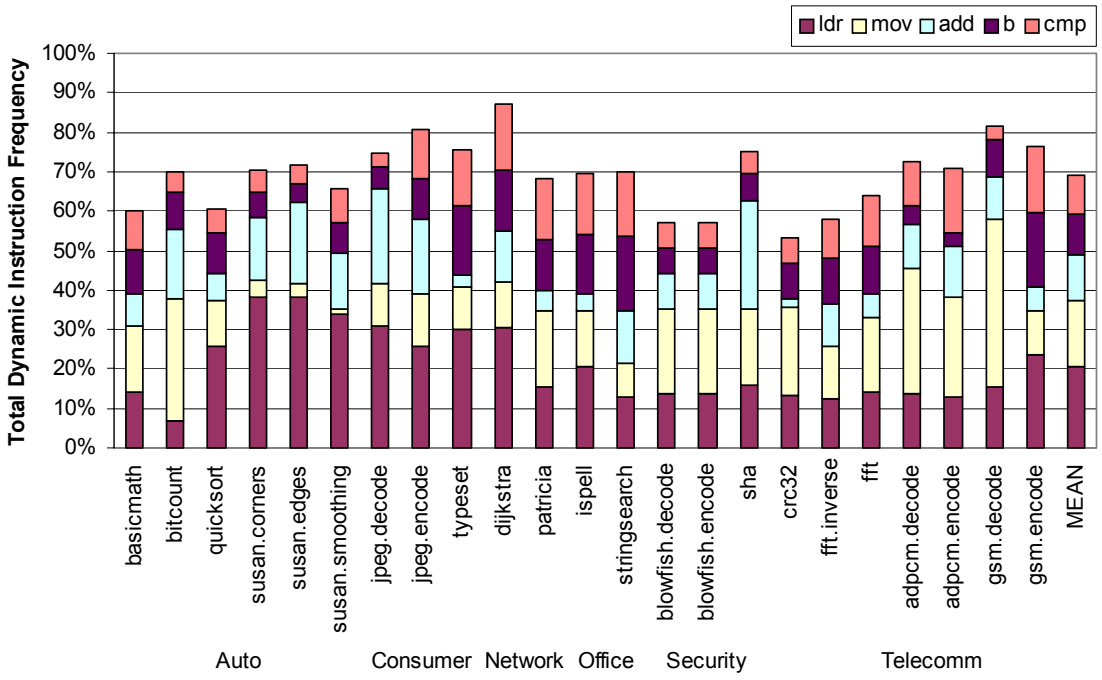

Figure 3.2: Synthesized Base Instruction Set Synthesis (BIS)

3.3 Instruction Set Synthesis Flow

This section describes how instruction synthesis is performed in FITS framework. Along with step-by-step description, we also demonstrated the synthesis process with real data from embedded benchmark programs. A representative subset of MiBench programs [Guthaus01] are compiled into ARM binary using GCC tool chain [GCC04]. We choose the ARM ISA as the target ISA to be studied, because it is popularly found in many embedded applications.

At instruction synthesis stage, the compiler must make tradeoffs in the instruction selection phase of optimization. This may include software emulation of rarely used instructions. In almost all cases, the instruction set mapping includes a Base Instruction

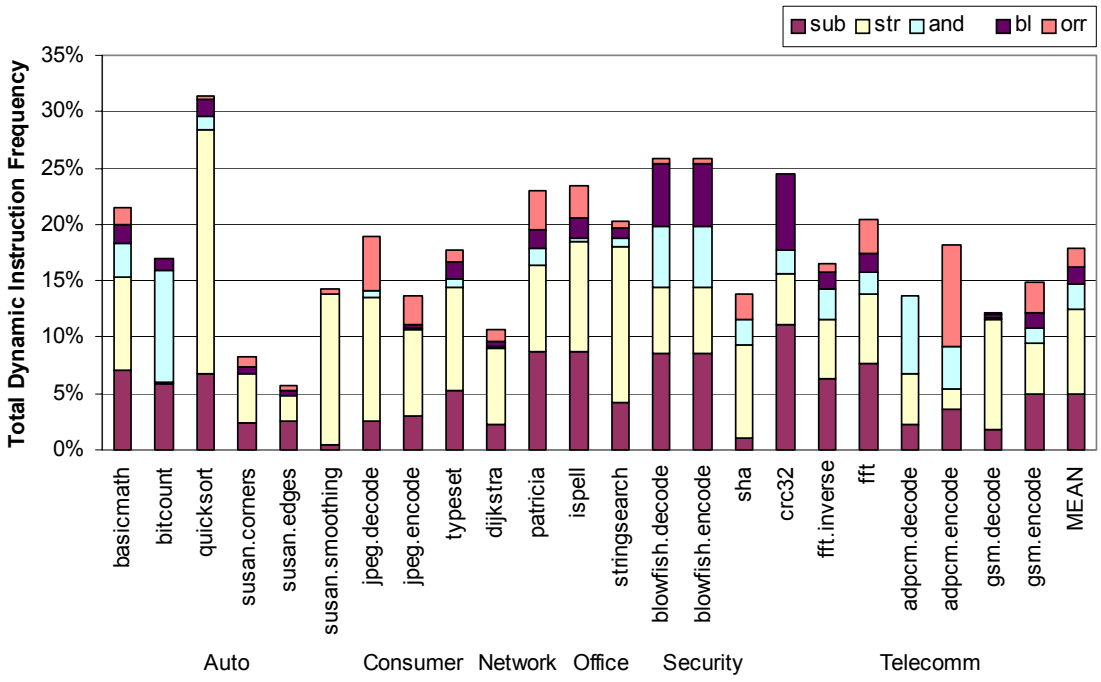

Figure 3.3: Synthesized Supplemental Instruction Set (SIS)

Set (BIS), a Supplemental Instruction Set (SIS), and an Application-specific Instruction Set (AIS).

3.3.1 Base Instruction Set (BIS)

A BIS includes instructions found across all applications. As illustrated in Figure 3.2, these may include instructions, such as load, move, add, branch, and compare, which are universally required by all applications. Together, these BIS instructions contribute to at least 53% of total dynamic instruction frequency as seen in *crc32*. In some applications like *dijkstra*, where application behavior can be easily captured in small dense loops with many repeated computations, the BIS can contribute as high as 87% of total dynamic

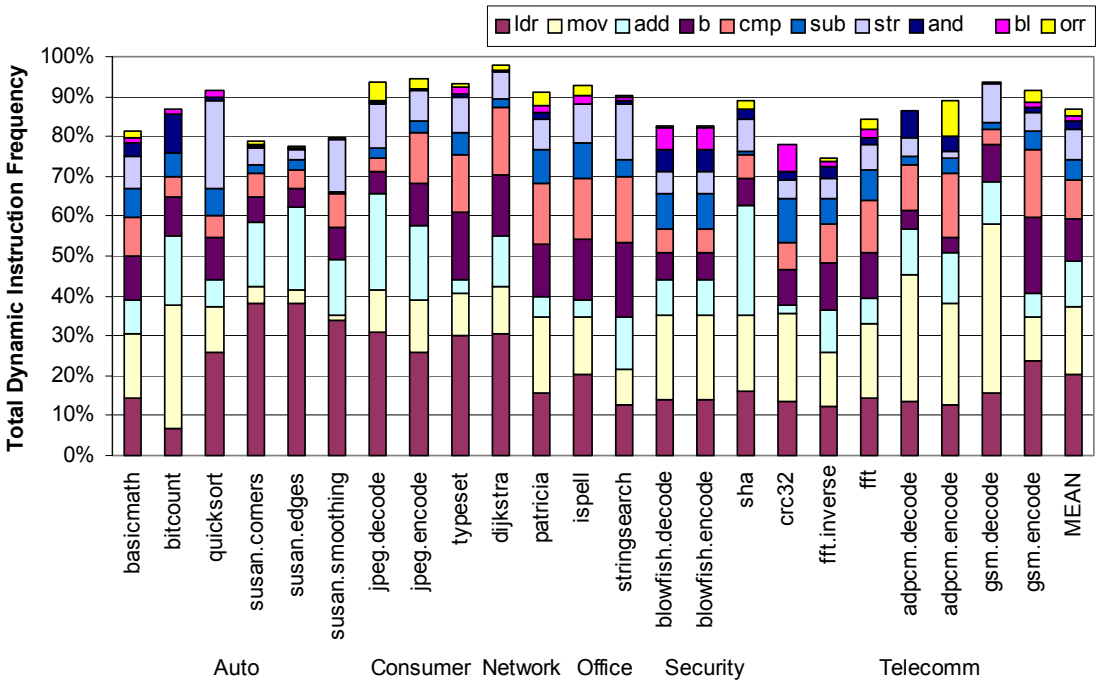

Figure 3.4: Synthesized Turing-complete Instruction Set (TIS)

instruction frequency. On average, 70% of total program time can be captured by BIS instructions.

3.3.2 Supplemental Instruction Set (SIS)

A SIS includes instructions required to make the instruction set Turing-complete [Church36][Turing36]. An instruction set is Turing-complete only if it can specify all behaviors of any arbitrary programs that can be envisioned, just as a Turing machine can describe any programs in the universe. As illustrated in Figure 3.3, a SIS may include instructions, such as subtract, store, and, branch and link, and or, which are required only to compose a Turing-complete instruction set. Together, these SIS instructions can

contribute to an average of 18% of total dynamic instruction frequency, and in the case of *quicksort*, as high as 31%.

3.3.3 Turing-complete Instruction Set (TIS)

It is essential for any FITS instruction sets to be Touring-complete, so correct program behaviors can be guaranteed when running a program that required some rarely executed instructions, which are not mapped to the target ISA. Therefore, after the BIS and SIS instructions have been identified and synthesized, the next step is to take the union of BIS and SIS to create the Turing-complete Instruction Set (TIS). As Figure 3.4 illustrates, the TIS contains instructions from both BIS and SIS. The total dynamic instruction frequency coverage of TIS for each application is the sum of those from BIS and SIS. On average, the TIS contribute to 87% of total dynamic program time. In the case of *dijkstra*, more than 97% of total dynamic instructions can be captured by TIS instructions.

The BIS and SIS together contain enough functionality to simulate any instructions not mapped for an application. BIS and SIS are generated differently and separately during the instruction selection phase. For clarity purpose, they are separated into two different instruction sets; even we include both them in all applications.

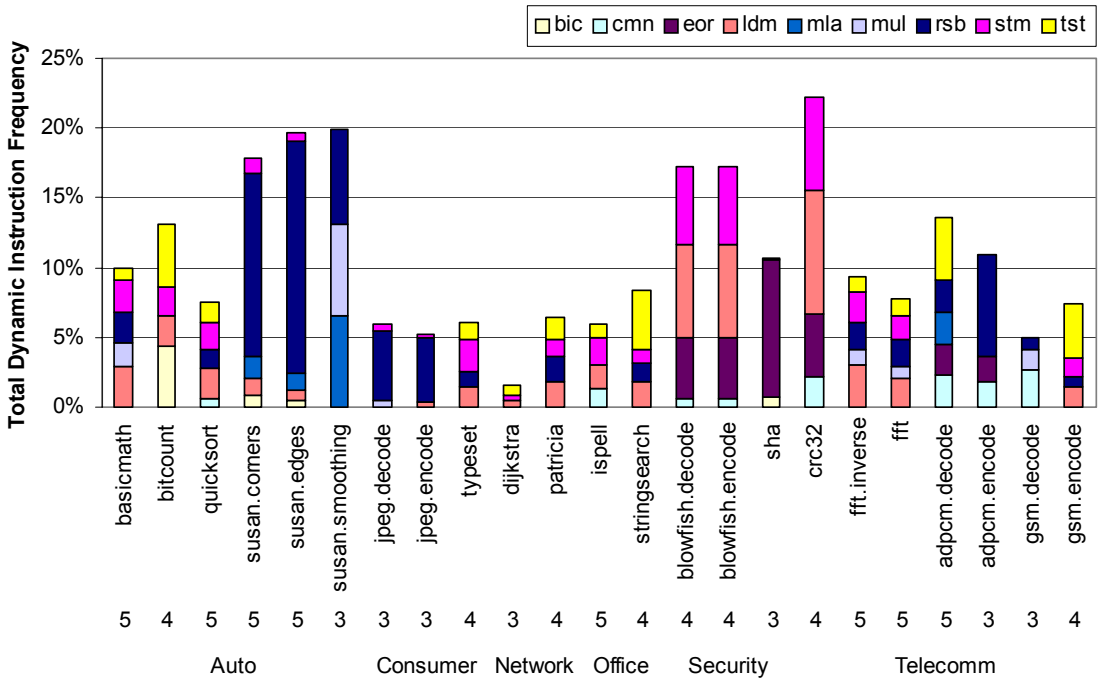

Figure 3.5: Synthesized Application-specific Instruction Set (AIS)

3.3.4 Application-specific Instruction Set (AIS)

In addition to the BIS and SIS instructions, FITS will include a set of application-specific instructions (taken from the set of functional units in the microarchitecture) necessary for the application to meet any performance goals. An application-specific instruction set (AIS) includes instructions that may not be required by all applications, and are included only to boost the performance of a particular application. Thus, the AIS from one application is probably going to be different from that of another application. The AIS is determined by evaluating the performance of existing TIS and the potential performance gain by adding any extra instructions. Since AIS is really application

independent, a FITS compiler only need to focus selecting the instructions that can maximize the performance of a given application.

As illustrated in Figure 3.5, each application has from 3 to 5 AIS instructions to boost its dynamic instruction frequency coverage. Different applications may require a different set of AIS instructions. Some applications rely more heavily on these additional AIS instructions to cover their dynamic execution requirement. For example, adding additional 4 AIS instructions can fulfill additional 23% of dynamic run time execution needs for *crc32*. Other applications, such as *dijkstra*, where most of its dynamic execution requirement can be met by its TIS instructions will only gain marginal benefits when adding AIS instructions. Adding 3 to 5 additional AIS instructions on top of existing 10 TIS instructions yields a total of 13 to 15 total instructions. For an instruction format that only has 4-bit opcode field, this will ensure there is at least one opcode entry reserved for configuring FITS programmable decoder.

3.3.5 Addressing Mode Synthesis

To improve the operand space utilization, FITS uses the two operand version of an instruction, say add, when almost all of the uses of the instruction can be done with two operands without requiring an additional move, provided there is a register space, and three operands otherwise. FITS can mix and match these two address modes, so that some instructions have two operands and some have three, as long as any two operand definition that has a three operand use is in the part of the register file that can be read by the three operand instructions. Since there is only one address mode for each instruction,

there is no need of extra opcode bit to indicate mode switch. Register allocation is also designed to trade off the register file size and encoding with register spill frequency.

3.3.6 Immediate Operand Synthesis

Since the space requirements for different categories of immediates demonstrate distinctive trends, as shown in the chapter of workload characterization, it makes sense to partition the immediate synthesis problem into three sub-categories and perform a category-based synthesis accordingly. FITS adopts an utilization-based technique to encode the immediate operand space. FITS identifies the most frequently accessed immediates and places them them in programmable, non-volatile memory storage, replacing the instruction immediate with an index into the immediate storage. This is similar to the dictionary compression method in [Lefurgy00] except: (1) FITS can dynamically reconfigure the total immediate field width and adjust widths of other instruction fields accordingly to best reflect the application's requirements, and (2) FITS targets the immedaite fields only rather than a whole instruction.

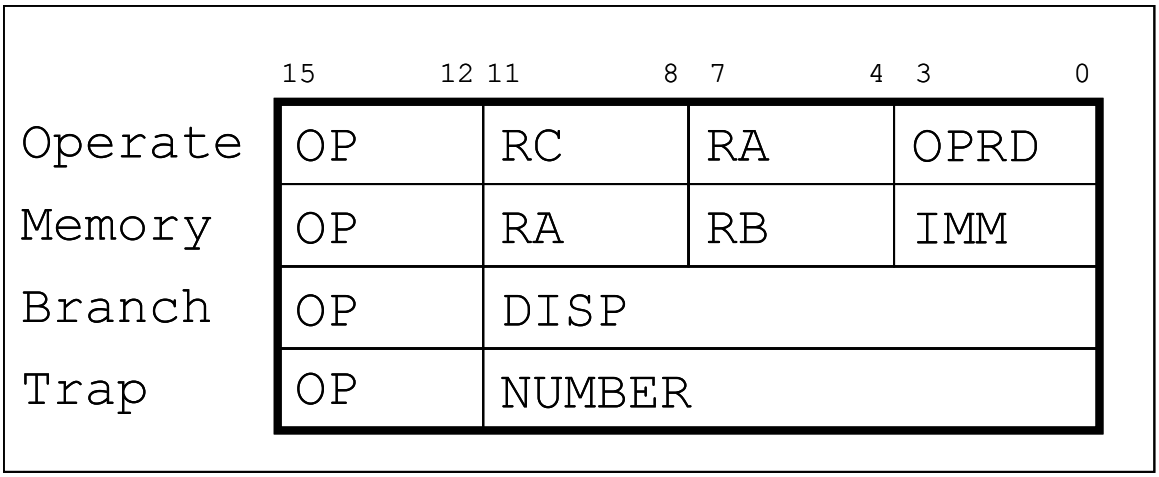

Figure 3.6: An Example FITS Instruction Formats for CRC32 of MiBench

3.4 Instruction Formats

FITS instructions are all 16 bits in various different instruction formats specifying 0, 1, 2, or 3 register fields. Generally speaking, all FITS ISAs have four basic instruction categories: operate, memory, branch, and trap. The details of the instruction format may vary, depending on the needs of the targeted application. For the illustration purpose, Figure 3.6 included example instruction formats, which was used for the CRC32 program from the MiBench Telecommunication benchmark group.

The Operate instructions are used for data processing such as arithmetic, compare, logical. They use a source register RA and a source operand OPRD, writing result register RC. For three-operand instructions, the OPRD field can be either a register specifier or an immediate value, depending on the addressing mode. For two-operand

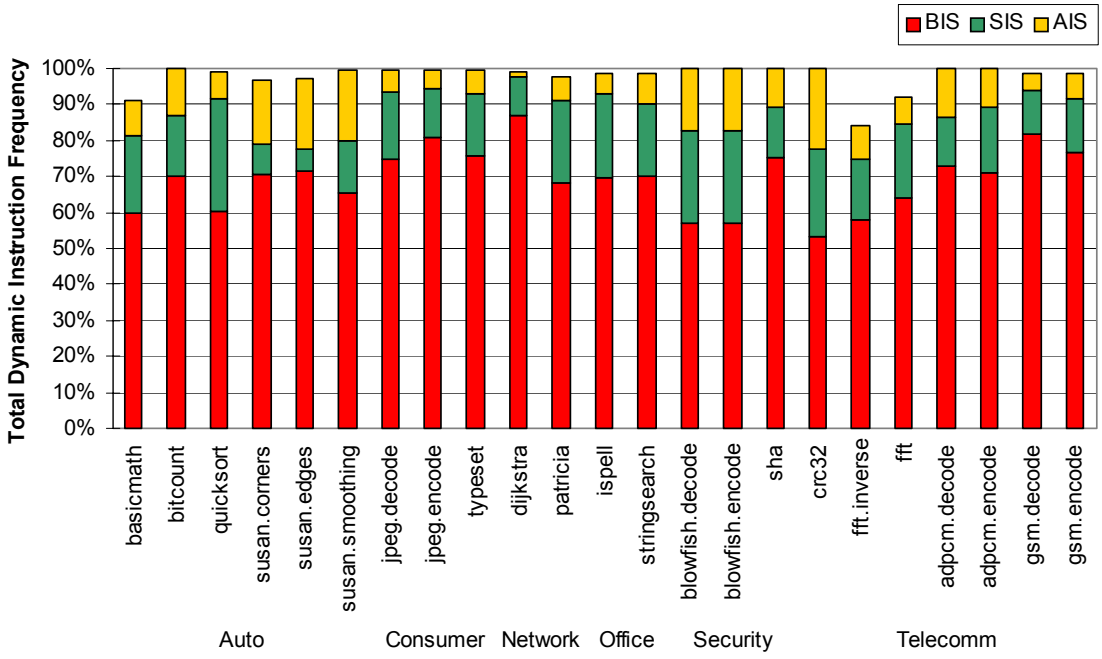

Figure 3.7: Synthesized Final Instruction Sets

instructions, the OPRD field can be combined with RA to specify an 8-bit zero-extended literal. The Memory instructions move data between register RA and memory, using RB plus a displacement indicated by the IMM field as the memory address. The Branch instructions change the program control flow to the target specified by the sum of 12-bit DISP offset and the PC. Subroutine calls put the return address in the register specified by the first four bits of DISP field. The Trap instructions perform interrupts, exceptions, task switching, and other complex operations that must be done atomically.

3.5 Synthesized Instruction Sets

For our experiments, we evaluated the effectiveness of FITS across a wide range of embedded applications contained in the MiBench benchmark suite [Guthaus01]. A

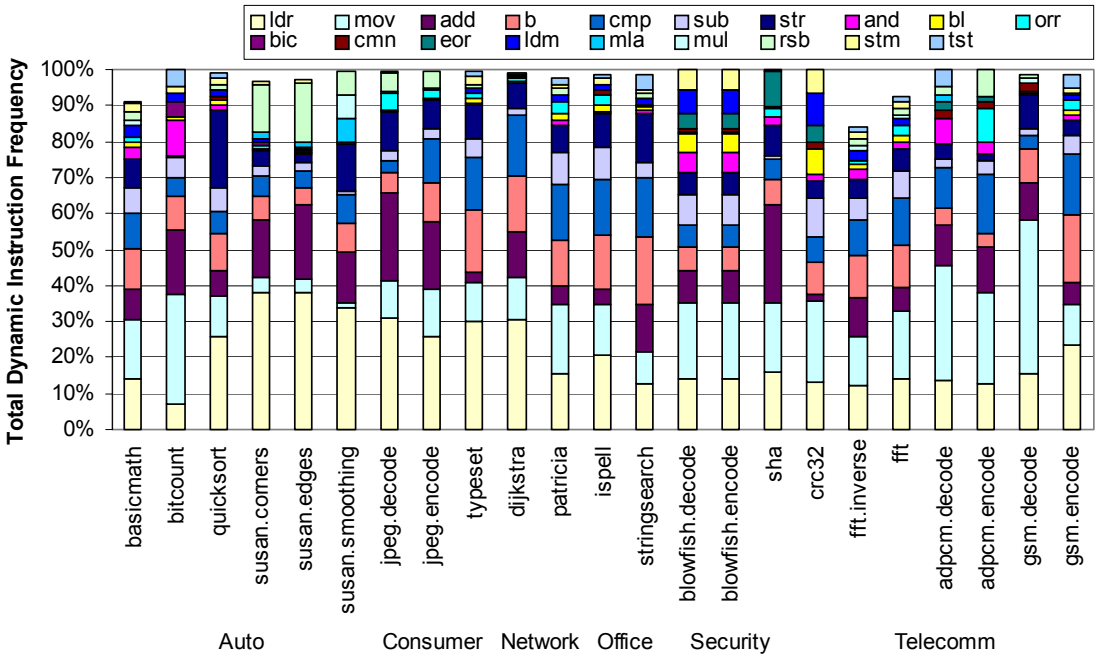

Figure 3.8: Synthesized Final Instruction Sets – Detailed Instruction Breakdown

representative subset of MiBench programs from each of its 6 application domains: automotive, consumer, network, office, security, and telecommunication, are compiled into ARM binary using GCC tool chain [GCC04]. We choose the ARM ISA as the target ISA to be studied, because it is popularly found in many embedded applications. We used the SimpleScalar toolset [Austin02] to examine the quality of the synthesized instruction set in terms of the dynamic execution needs for opcodes and immediates, which the synthesized ISA can capture.

3.5.1 Synthesized Instructions

Figure 3.7 shows synthesized final instruction sets for different applications. The synthesized final instruction set consists of three subsets of instructions: BIS, SIS, AIS as

described in previous section. Figure 3.7 illustrates the composition of the final instruction sets in terms of the distribution of BIS, SIS, and AIS. Figure 3.8 provides the detailed instruction breakdown of synthesized instruction sets.

BIS consists of five opcodes: *load*, *move*, *add*, *branch*, and *compare*. BIS accounts for the majority of the dynamic instruction execution needs: on average, 69.1% of total executed instructions are those from the BIS. SIS also consists of five instructions: *subtract*, *store*, *and*, *branch with link*, and *or*. SIS accounts for 17.9% of total executed instructions on average. The size of AIS is different from one application to another. The union of entire suite's AIS consists of: *bit clear*, *compare negative*, *exclusive or*, *load multiple*, *multiply accumulate*, *multiply*, *reverse subtract*, *store multiple*, and *test bits*. Depending on the individual execution characteristics, each application includes at most 3 to 5 of them and each AIS from different application accounts for 10.8% of total executed instructions on average. This AIS distribution will likely expand as we will modify the compiler to utilize more specialized instructions (such as a 0-cycle loop instruction), but since we limited ourselves to use the same code generation as the ARM, the ability to identify useful AIS instructions was also limited. Together, the contributions made by BIS, SIS, and AIS account for 97.8% of total executed instructions. One important observation to be made here is that instructions required by each benchmark are different, i.e., no fixed set of 16 instructions would be sufficient for all programs.

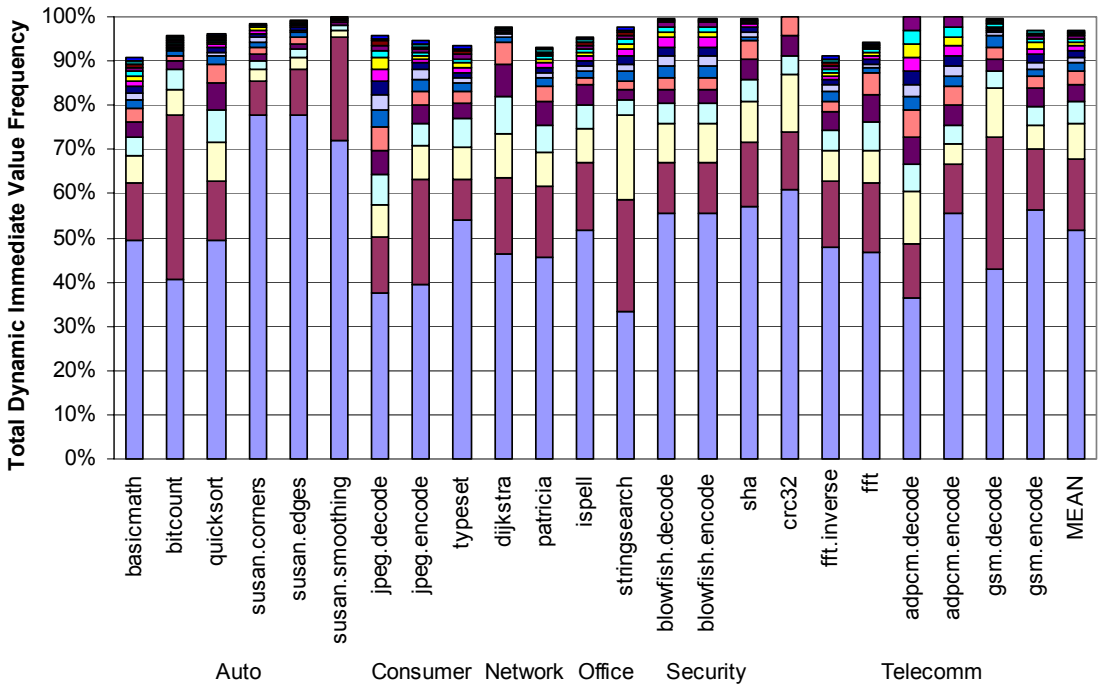

Figure 3.9: Synthesized Final ALU Immediate Operands

3.5.2 Synthesized Immediate Operands

Figure 3.9 shows the contribution of the top 16 synthesized ALU immediate operands to the total number of accessed frequencies of the entire ALU immediate operand space. It is satisfying to learn that with as few as only 16 unique immediate operands, the synthesized 4-bit immediate scheme can capture, on average, 96.9% of total number of references made to the ALU immediate operand space. It is interesting to observe that, on average, 51.8% of the contribution was made by the most frequently referenced immediate value: zero.

Figure 3.10 shows the results for synthesized memory immediate operands. On average, 87.4% of total references made to entire memory immediate operand space can

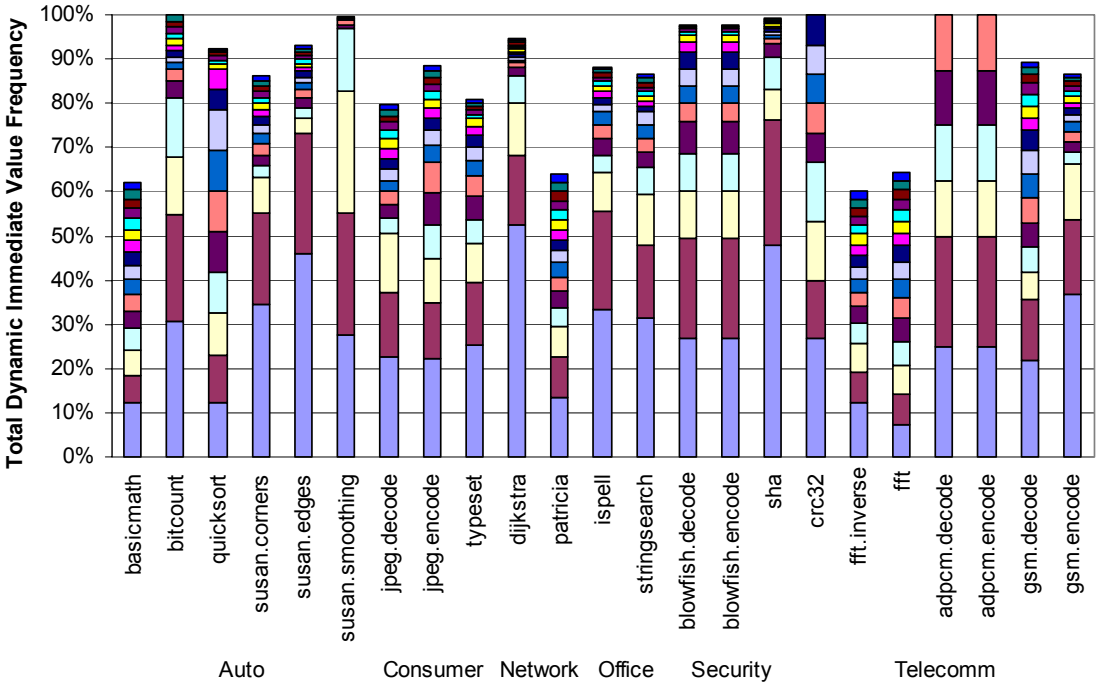

Figure 3.10: Synthesized Final MEM Immediate Operands

be captured by the top 16 synthesized immediate operands. The anomalies are *basicmath*, *patria*, *fft* and *fft.inverse*, which are due to excessive utilization of floating point memory immediate operands. Similar to the ALU immediate operand synthesis result where the single most frequently accessed immediate value is responsible for a significant portion of total immediate references: 26.9% of the total memory immediate operand references were attributed to the value zero.

3.6 FITS Microarchitectural Enhancement

The FITS framework mentioned thus far assumes standard general-purpose microarchitecture that can be seen in many embedded processors, such as ARM. In the case of using the same the microarchitecture that can perform the same functional

executions, the best FITS can do is to perform equally well like conventional processor designs with 32-bit instructions, but not better. It is certainly cost-effective to use half-sized instructions, which, in turn, yield roughly half-sized programs that can perform equally well compared to the 32-bit counterparts. Yet, to raise the performance to the next level, we must equip FITS framework with performance enhancers. The performance enhancers we extend FITS framework with can speed up both data processing and program control flow streamlining.

The FITS framework extension includes an enhanced microarchitecture that provides the additional rich capabilities and exceptional horse power to meet the ever increasing application requirements. These microarchitectural enhancements come from two special on-chip processing units: the Versatile Integrated Processing (VIP) unit, and the zero-overhead loop execution unit. The VIP unit is a universal data-crunching engine that delivers superb data computing and data streaming performances. The zero-overhead loop execution unit streamlines the program control flow by removing expensive loop control overhead from both nested and non-nested loops.

Finally, to provide an interface to utilize these special resources efficiently at ISA level, FITS again relies on the use of a programmable decoder that can dynamically map a needed operation into instruction set definition as necessary. The use of a programmable decoder allows adding new capabilities to microarchitecture without being restricted by limited instruction space, which is one of the most critical constraints existing in most multimedia processors.

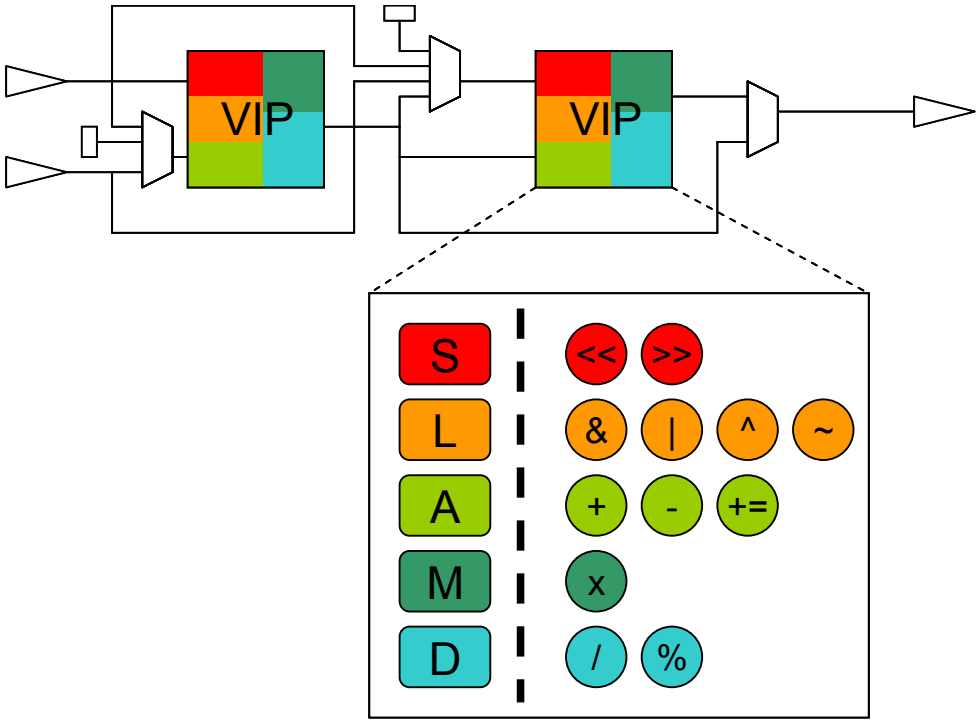

Figure 3.11: Versatile Integrated Processing (VIP) Unit

3.6.1 Versatile Integrated Processing (VIP) Unit

A Versatile Integrated Processing (VIP) unit can execute up to five basic types of computations: Shift (S), Logical (L), Arithmetic (A), Multiplication (M), and Division (D), or SLAMD, as illustrated in Figure 3.11. Each basic type includes one to four different operations. Using as few as three multiplexers enables all possible permutations of input operands to a VIP unit. Although one VIP unit is stacked in front of the other VIP unit, the two-to-one multiplexer at the end of the datapath can select either the result of the first VIP unit or that of the second VIP unit to output. This creates a non-binding cascaded processing paradigm, which can freely act as a single-level or a dual-level data processing engine as directed by the requirement of a program.

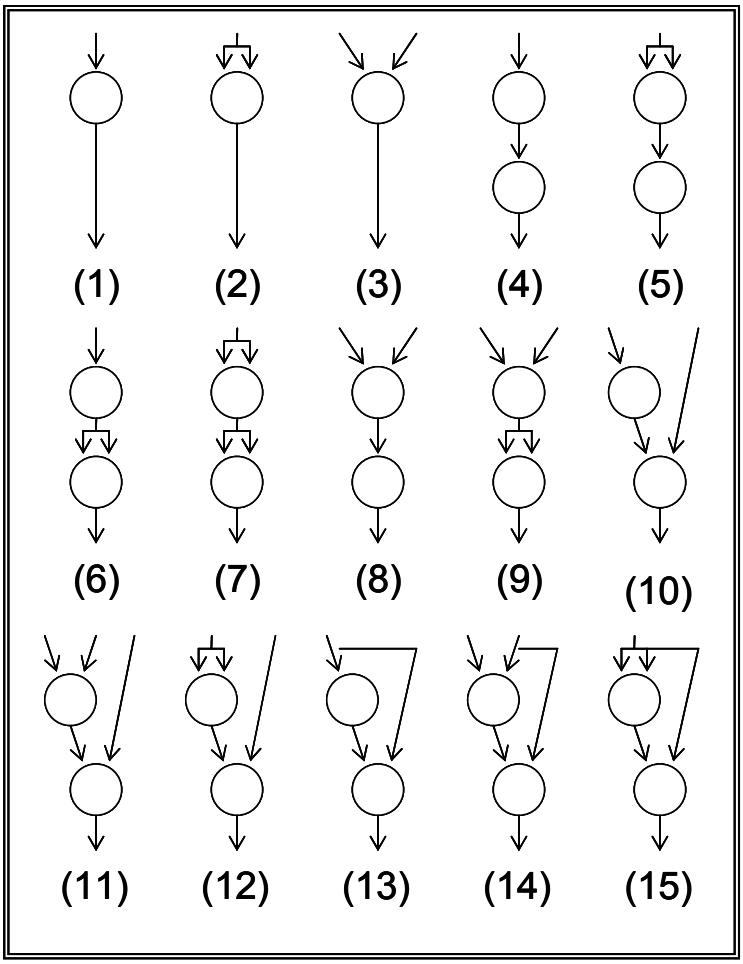

Figure 3.12: VIP Instruction Computing Patterns

The VIP unit has very flexible operand routing schemes, which can generate a rich set of computing patterns. As shown in Figure 3.12, there are as many as fifteen VIP instruction patterns, which a special data streaming operation can be synthesized to. A circle in each VIP instruction pattern represents one of the twelve basic SLAMD operations supported by a VIP unit. Synthesizing custom instructions to tailor a FITS processor to the requirement of an application is simple. For example, a logical operation followed by a shift computation, such as xor-right-shift (e.g. ^a >> 5), can be synthesized using VIP instruction pattern ten with XOR (^) operation executed by the first VIP and a

right shift (>>) executed by the second VIP. Multiply-accumulate (MAC) instruction, such as (c += a x b), is another popular common computation in most media and DSP applications. To synthesize a MAC instruction, the VIP instruction pattern eight can be used with the first VIP executing multiply (x) operation and the second VIP executing accumulate (+=) operation.

The VIP unit does not require extra read and write ports from the register file. The FITS register file supports two register reads per cycle, which is common in most modern RISC processors. Any pattern that can read three input operands (e.g. pattern eleven) will take the third input operand from the immediate field of an instruction. Since there is no extra cycle to wait for the input operands to arrive, provided that the VIP datapath is very well optimized and can be clocked at modest frequency of 250 MHz, all VIP instructions can be completed in one clock cycle. The VIP unit may be pipelined to achieve a higher clock frequency, if proven necessary. Yet, the discussion of this approach is beyond of the scope of this study.

Given that a VIP unit can execute up to a total of twelve different SLAMD operations, and there are three single-level computing patterns and twelve dual-level computing patterns for VIP instructions; a FITS processor can specify as many as (12 x 3) + (12 x 12 x 12), or 1764, distinct VIP instructions. This is a rich set of instructions, which can efficiently execute any single-level or dual-level computations that can be envisioned. To maintain the clock rate, 4 patterns that involve multiply and/or divide followed by another multiply and/or divide are not used. Since these two-level multiply/divide patterns, if ever used, can be easily replaced with a single multiply/divide, restrict them from being used will not affect the wide applicability of

VIP. To save chip area, it is possible to completely replace the regular ALU with a VIP unit since the operations implemented in the VIP unit is a superset of that of a regular ALU. Different applications exhibit different behaviors that may require different subset of VIP instructions to match. Any unused datapath within the VIP can be clock gated [Srinivasan05] to reduce the power consumption.

3.6.2 Zero-Overhead Loop Execution Unit

For many applications, a large percentage of the dynamic program execution time is spent in the innermost loops of a program [PattersonHennessy03]. These loop execution incur significant overhead due to the increment or decrement of the loop counter variables and the branches to initiate a new iteration.

Many software and hardware techniques have been proposed to improve loop execution time. Please see the chapter of related work for a more complete list of relevant literature. The zero-overhead loop support in FITS is similar to [Analog-ADSP21160] in that it is stack-based. Stack-based zero-overhead loop execution can support not only the innermost loop but also nested loops as well. Moreover, since it does not store the actual instructions within loops, there is no need for additional storage, which can result in extra power consumption and area overheads.

FITS microarchitecture supports three hardware stacks: top-of-loop address stack, end-of-loop address stack, and loop count stack. These three address stacks work in a synchronized manner for zero-overhead loop execution. When the FITS processor executes a zero-overhead loop instruction, LOOP, the program sequencer pushes the

address of the top-of-loop address, which is the address of the instruction following the LOOP instruction, on the top-of-loop address stack. The address of the last loop instruction is pushed on the end-of-loop address stack. The loop iteration count, which keeps track of the total number of iterations a loop must execute, is pushed on the loop count stack. The LOOP instruction specifies both the end-of-loop and the loop count. The end-of-loop address can be either a label for an absolute program memory address, or a PC-relative twos-complement address. The loop count is either an immediate unsigned value or can be the value of a register. The program sequencer decrements the loop count at the end of each loop iteration. The loop executes until the loop count reaches zero. The FITS stack-based zero-overhead looping mechanism removes most control overhead for executing loops. While this zero overhead loop instruction supports counter-based loops, i.e. the total number of loop iteration count is known at compiled time or can be easily computed at run time, this is sufficient for most media and DSP applications. In our experiments, we found that almost all program loops are or can be easily converted to counter-based loops. The benefits of this overhead removal are demonstrated later in the results chapter.

3.7 FITS Programmable Instruction Decoder

The most vital piece of innovation for FITS framework to work is the programmable instruction decoder. This section describes its detail mechanisms and implementation.

3.7.1 Hardwired Instruction Decoder

In almost all modern processors, datapath control lines for an instruction are hardwired. These hardwired control lines regulate the behavior (e.g. reads or writes) of different parts of the processor datapath, such as updating the PC and the register file, selecting ALU functions, accessing memories, and other processor state, etc.. A non-volatile read-only memory (ROM) is usually used to store control lines associated with each instruction. This conventional hardwired instruction decoder scheme is illustrated in Figure 3.13. When an instruction is decoded, its opcode is used to select the corresponding row of control line patterns to set the control on different parts of the processor datapath. The width of the ROM determines the number of control signals an instruction can regulate. The more complex underlying datapath is; the larger number of control signal lines there are, hence, the wider of each ROM entry has to be. The height or the number of rows of the ROM determines the number of distinct datapath control signal patterns the decoder can store. It is proportional to the size of the instruction opcodes. The larger the opcode is; the more operations a processor can support.

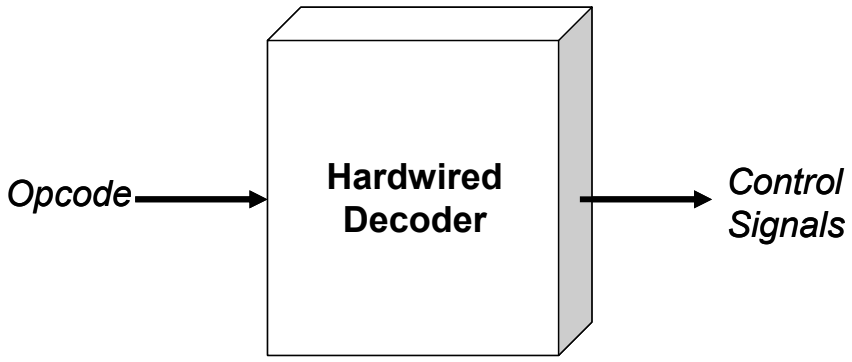

Figure 3.13: Conventional Hardwired Instruction Decoder

The reasons for using non-volatile memory, such as a ROM, instead of using volatile memory, such as RAM, to store control signal patterns is mainly because the fixed nature of conventional ISA designs. Because the control signals regulated by an opcode are never changed, it makes sense to store these control signals in memory that can retain its contents even when the power is turned off. Otherwise if a volatile memory is used, the control signals need to be loaded from somewhere (e.g. BIOS) into this volatile memory each time the machine is booted.

3.7.2 Programmable Instruction Decoder

Unlike the conventional hardwired fixed decoding scheme, the instruction decoding of a FITS processor is programmable. This section explains the architecture and mechanism of the FITS programmable decoder. A detailed evaluation of this programmability overhead is provided in the evaluation chapter.

One of the key components enabling FITS to adapt to different applications, with the same microarchitecture, lies on the use of its programmable instruction decoder. This

adaptability is in the form of having custom instruction set for each different application. The FITS instruction decoder consists of a standard *n*-to-2^n binary row decoder and a 2^n-entry programmable memory, where *n* is the opcode width and 2^n is the number of instructions specified by the ISA. For all the embedded applications we studied, 4-bit opcode is sufficiently large to meet the execution requirements, which makes the number of FITS instruction decoder entries equal to 16. The 16-entry FITS decoder is used to store the instruction control information, which ordinarily would be stored in the ROM of a conventional decoder. The width of a FITS decoder entry is same as that of a ROM entry, so each FITS decoder entry is wide enough to store all control line signals for one instruction. When the 4-bit opcode input is inserted, exactly one of the16 outputs is activated and the corresponding instruction control signals are fetched and being sent down the pipeline to set the datapath accordingly.

Either static random-access memory (SRAM) or a register flip-flop can be used to implement this programmable memory of FITS decoder. Although the dynamic random-access memory (DRAM) has higher density that uses as little as one transistor per bit, a periodic refresh operation is required to keep its memory contents from disappearing. Thus, the DRAM is not considered when selecting the memory cells to implement FITS decoder.

By making the instruction decoder programmable, designers can freely select a subset of pre-defined microarchitecture functions that are best suitable for the targeted application. These selected functions are mapped to the ISA of a FITS processor by loading their corresponding instruction control signals into the programmable decoder.

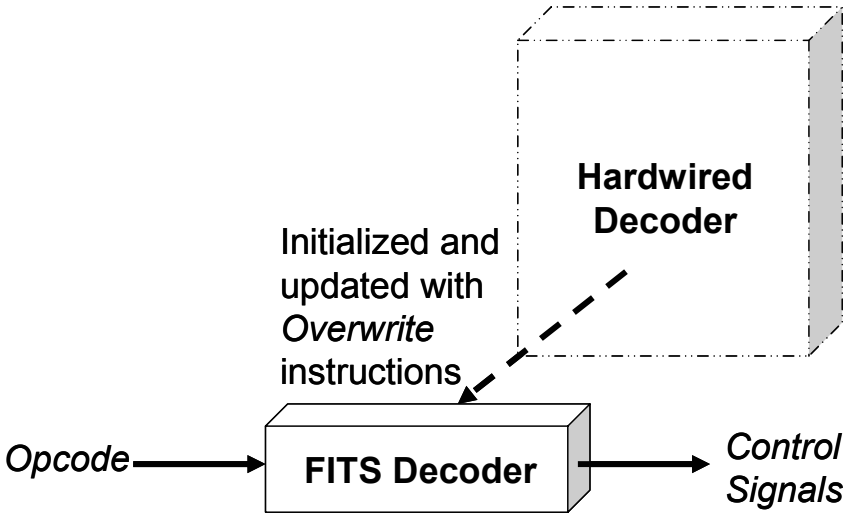

Figure 3.14: FITS Programmable Instruction Decoder

Specifically, a custom instruction set is defined and synthesized by the compiler for the targeted application.

At boot time, the programmable decoder is initialized with the synthesized instruction set. We introduced a *OVERWRITE* instruction and included it in all synthesized ISA to perform this decoder initialization. As Figure 3.14 illustrates, the *OVERWRITE* instruction updates the FITS decoder with the instruction control information stored in the original ROM decoder. The *OVERWRITE* instruction takes two operands: *DestinationEntry* and *SourceEntry*. The *DestinationEntry* operand specifies a FITS decoder entry, to which the instruction control information of a selected opcode is going to be written. The *SourceEntry* operand specifies a ROM decoder entry, from which the instruction control information of a selected opcode is going to be used to load the FITS decoder entry. The decoder initialization overhead is small. The reading of the ROM decoder and writing to the FITS decoder can be done in a single cycle. With 4-bit

opcode that specifies up to 16 instructions, we pay a one-time 16 cycles of start-up cost to load the FITS programmable decoder.

Depending on if we need to do any dynamic ISA reconfiguration or not, the fixed-wired ROM instruction decoder can be handled differently to reduce its power consumption. The need of dynamic ISA reconfiguration can be determined at compile time, when the target application is profiled and analyzed and the ISA synthesis is to be performed. If dynamic ISA reconfiguration is not necessary, then the ROM decoder is only accessed during the chip initialization, after which there is no need to access the ROM decoder again and it can be completely turned off to save power consumption. After initialization, all instruction decoding is handled directly by the FITS decoder in the same way that a conventional ROM decoder handles it.

If the ISA will need to be dynamically reconfigured at run time, we want the contents of the ROM decoder to be accessible during the ISA reconfiguration and we can leave the ROM decoder to be inactive or turned-off otherwise to maximize our power savings. Our preliminary studies show that the default synthesized ISA is effective enough to satisfy the majority of the program execution requirement. Thus, the ROM decoder is mainly accessed during the chip initialization. After initialization, the rarely accessed ROM decoder stays mostly inactive or powered-off and consumes nominal leakage power, which is the price we pay for the dynamic instruction re-configurability. More aggressive dynamic power management techniques, such as dynamic voltage scaling or dynamic frequency scaling [Magklis03], can be easily applied to reduce the power consumption of the ROM decoder further, but the discussion of which is beyond the scope of this dissertation and is left as our future research.

CHAPTER 4

EXPERIMENTAL METHODOLOGIES

4.1 Power Modeling

Power dissipation is becoming a critical concern for semiconductor industry. If current design trends continue, a typical microprocessor will consume 50 times more power than that can be supported by cost-effective packaging techniques by 2016 [Allen02]. Clearly, power has become one of the most serious design constraints in today's process generations. Design engineers who are serious about providing power-efficient cutting edge technology to their customers see the value in handling power problem early in the design flow. By understanding a design's power requirements at every phase of the design cycle, engineers will be able to produce high-performance, power sensitive products without impacting cost or time to market. To help illustrate how FITS addresses this issue, this section describes the power metrics and modeling tool that were used to measure the power dissipation results presented in the experiment chapter.

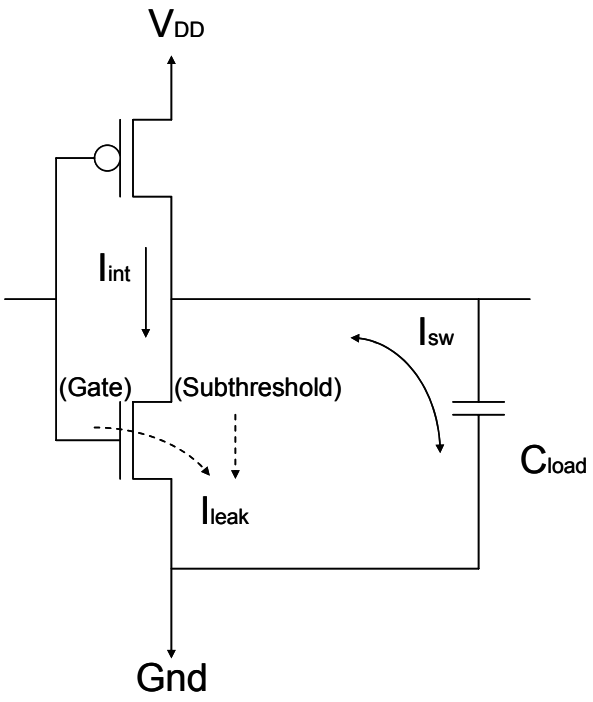

Figure 4.1: Power Dissipation in CMOS Circuits

4.1.1 Power Components for CMOS Circuits

There are three components of power consumption in complementary metal-oxide semiconductor (CMOS) logic circuits: switching (capacitive) power; internal (short-circuit) power; and leakage power. Figure 4.1 depicts how electric current flows in a CMOS transistor: I_{sw} , I_{int} , and I_{leak} are currents associated with switching power, internal power, and leakage power respectively.

Switching or capacitive power, which typically represents 60 to 80 percent of power consumption, is the power dissipated when a load capacitance is charged or discharged, i.e. 0 to 1 or 1 to 0 transitions of the nets in the design.

Internal or short circuit power, typically 20 to 40 percent of power consumption, is the power consumed within a cell. This includes power loss due to short-circuit (V_{DD} to ground) current as well as all power dissipated due to switching of internal nets. The sum of the switching power and internal power is together referred as dynamic power.

Leakage or static power, on the other hand, is the power dissipated due to sub-threshold leakage and the current flow through the reverse-biased p-n junction between diffusion and substrate. The leakage power is also referred as static power.

4.1.2 Power Equations for CMOS Circuits

In computer architecture research community, the overall power consumption of a CMOS logic is often modeled as the sum of dynamic power and static power consumption [Mudge01] as equation (4.1) shows:

$$P = ACV^2 f + \tau AVI_{\text{int}} f + VI_{leak} \tag{4.1}$$

The first term models the switching power caused by the charging and discharging of the capacitive load on the output of each logic gate: A is the fraction of gates actively switching; C is the total capacitance load of all gates; V is the supply voltage; f is the system operating frequency.

The second term measures the internal power. The I_{int} is the short-circuit current, which flows between the supply voltage and ground when the output of a CMOS logic gate switches. The τ is the coefficient that captures the momentary loss of internal power

during a gate switch. Sometimes, the short circuit power can be ignored. The reason is its relatively small contribution to the dynamic power and can thus be absorbed by the dynamic power, if necessary.

The third term measures I_{leak} , the static power lost due to leakage current. Regardless whether a gate is actively switching or not, as long as it is not turned off, it will consume the leakage power.

Dynamic power is activity based because it is directly related to the toggling frequency and operating duration of the gates in the circuit. The leakage power, on the other hand, is unaffected by activity since it is governed only by the number of gates and their threshold voltages. The only time that leakage can be reduced to zero is when the gates are turned off. Leakage power accounts for the majority of power dissipated when the circuit is inactive. Thus, this is an important metric to measure and optimize for portable battery-powered applications.

In addition to the dynamic power and static power, the peak power is also relevant because exceeding an upper power limit imposed by a system will lead to circuit damage. Reduction in peak power may also help reduce the *di/dt* noise, an inductive effect caused by sharp changes in power consumption which can result in circuit malfunction.

These fore-mentioned power characteristics imply that: given the same *C* and *V*, smaller logic block that completes a task faster could save both dynamic and static powers and possibly the peak power. As it will be shown in the experiment chapter, this is exactly how FITS achieves power savings for its programmable decoder, VIP function units, instruction cache, which all translate into significant power savings for the target

applications. . A FITS processor can achieve significant dynamic and static power savings by executing powerful and half-sized instructions. Powerful VIP instructions reduce dynamic program run time, thus reducing dynamic power. Half-sized instructions require smaller functional block and datapath (e.g. instruction cache), thus reducing static power.

4.1.3 Power Modeling Tools

It is very difficult to model power consumption of a system at the architectural level. A natural solution is to build a power estimator into the cycle simulators. However as [Kim01] pointed out, cycle simulators intentionally omit considerable implementation detail to speed up simulation speed. Therefore, the challenge is to select the necessary details that must be put back in to produce accurate power figures.

In this dissertation, we used a modified version of the "sim-panalyzer" [Panalyzer04] to run power modeling simulation for our experiments. "sim-panalyzer" is an infrastructure for microarchitectural power simulation at the architectural level. It is built on top of SimpleScalar-ARM simulator [Austin02]. "sim-panalyzer" measures power consumption by tying cycle accurate behavior to activity at the gate level for obtaining the dynamic power and to estimate the number of gates that the microarchitecture requires for obtaining the static power. Specifically, "sim-panalyzer" computes the power dissipation with the switching capacitance multiplied by the number of microarchitectural accesses. It uses the logic simulator to collect the number of gate switching in each internal node of the targeted circuit on the fly, and the capacitance

extractor to estimate the switching capacitance of each node. The chip-wide power dissipation breakdown given by the simulator is consistent with that of an actual fabricated StrongARM design [Montanaro96].

4.2 Benchmarking Workloads

This section describes the procedures that we used to perform the workload benchmarking. The applications we studied are from the MiBench benchmark suite [Guthaus01] and MediaBench benchmark suite [Lee97].

4.2.1 Profiling Procedures

Profiling allows a system designer to learn where a program spent its time and which functions called which other functions while a program was executing. This information can show a designer which pieces of a program are bottlenecks, and might be candidates for optimization to make a program execute faster. For this study, we use the GNU profiler, gprof [GPROF], to analyze the target workload. The gprof profiler works by changing how every function in a program is compiled so that when a function is called, it will stash away some information about where it was called from. From this information, the gprof profiler can figure out what function called it, and can count how many times this function was called. This change is made by the compiler when your program is compiled with the `-pg' option, which causes every function to call a profiling library routine responsible for constructing an in-memory call graph table to record a

function's parent and its parent's parent. This is done by examining the stack frame to find both the address of the child, and the return address in the original parent. Since the profiler uses information collected during the actual execution of a program, it is often used on programs that are too large or too complex to analyze by reading the source code. There are two forms of profiling output for program analysis. The flat profile shows how much time a program spent in each function, and how many times that function was called. The flat profile states concisely which functions burn most of the machine cycles. The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places to eliminate excessive function calls that use a lot of time. In this dissertation, we used both profiling outputs as general guidelines to apply FITS optimizations.

4.2.2 Instruction Cache Evaluation Procedures

To do an analysis of power consumption and performance evaluation on real embedded workload, four different processor configurations were simulated with "sim-panalyzer." A representative subset of the MiBench suite [Guthaus01] is compiled into the ARM binary using the GCC tool chain [GCC04]. To clearly demonstrate the effectiveness of FITS in reducing instruction cache power dissipation, we restrict the experiment to only allow a single controlled variable: instruction cache size. There are two different instruction cache sizes: 16 Kb or 8 Kb. For simplicity, simulations of the original ARM code with a 16 Kb and an 8 Kb instruction cache are abbreviated as

	ARM16	ARM8	FITS16	FITS8
Fetch queue	8	8	8	8
Branch Predictor	Not-taken	Not-taken	Not-taken	Not-taken
Fetch & Decode Width	2	2	2	2
Issue Width	2	2	2	2
Issue Order	In-Order	In-Order	In-Order	In-Order
Function Units	1 Int ALU, 1 FP Mult, 1 FP ALU	1 Int ALU, 1 FP Mult, 1 FP ALU	1 Int ALU, 1 FP Mult, 1 FP ALU	1 Int ALU, 1 FP Mult, 1 FP ALU
Instruction L1 Cache (cache size, associativity, block size)	16 Kb, 32-Way, 32-Byte	8 Kb, 32-Way, 32-Byte	16 Kb, 32-Way, 32-Byte	8 Kb, 32-Way, 32-Byte
Data L1 Cache (cache size, associativity, block size)	16 Kb, 32-Way, 32-Byte	16 Kb, 32-Way, 32-Byte	16 Kb, 32-Way, 32-Byte	16 Kb, 32-Way, 32-Byte
L2 Cache	None	None	None	None
Memory (bus width, first block latency)	4-Byte, 64 Cycles	4-Byte, 64 Cycles	4-Byte, 64 Cycles	4-Byte, 64 Cycles
Frequency	250 MHz	250 MHz	250 MHz	250 MHz
Power Supply	1.8 V	1.8 V	1.8 V	1.8 V

Table 4.1: Processor Configuration for Evaluating I-Cache Performance

ARM16 and ARM8 respectively; likewise, simulations of the FITS-optimized code with a 16 Kb and an 8 Kb instruction cache is abbreviated as FITS16 and FITS8 respectively. The rest of the microarchitecture remained the same and were modeled after Intel's SA-1100 StrongARM embedded microprocessors [Intel-SA1100] as shown in the following table.

We ran full simulation on all compatible benchmarks to their completions without skipping any instructions. Up to approximately 1 billion dynamic instructions were simulated for all benchmarks. Due to compatibility issues between the MiBench and the simulator, *basicmath* and *gsm.encode* are dropped from the power dissipation study and *gsm.decode* was thus renamed to *gsm* accordingly.

4.2.3 Programmable Decoder Evaluation Procedures

To perform a realistic and accurate cost evaluation on programmable decoder overhead, memories used for both the FITS programmable decoder and the regular ROM decoder were synthesized using the Artisan Memory Generator [Artisan-MemoryGenerator] under worst-case process conditions. The technology used is TSMC's six-layer metal 0.18 μm CMOS process [TSMC-18]. The SRAM memory synthesized for the FITS programmable decoder is a high-speed and synchronous SRAM, which has a single read port and a single write port [TSMC-SRAM]. The SRAM's storage array is composed of six-transistor cells with fully static memory circuitry. The ROM memories synthesized for the both the FITS and regular decoders are high-speed and synchronous diffusion ROMs that have a single read port and a single write port [TSMC-ROM]. The diffusion ROM's storage array is composed of diffusion-programmable single-transistor cells with fully static memory circuitry. Both SRAM and ROM operate at a voltage of 1.8V ± 10% and a junction temperature range of -40°C to +125°C.

Many of the characteristics of a memory cell are depend on its y-mux type, which defines the aspect ratio of the memory layout. When the y-mux type is changed from one

to the other, it will change many major characteristics, such as access time, area, and power consumption, of the memory. Consequently, the width of y-mux circuit for SRAM and ROM is fixed to be 4 and 8 respectively to ensure aspect ratio of the memory layout remain the same. These y-mux widths were selected because their corresponding aspect ratios yielded the best performance for each memory type.

4.2.4 VIP and ZOLE Evaluation Procedures

To show true performance advantages of VIP and ZOLE units, we evaluate both units on realistic embedded multimedia applications. We select the MediaBench benchmark suite [Lee97] as our evaluation benchmark. Four processor configurations were simulated on a modified SimpleScalar [Austin02] tool chain. Modifications to SimpleScalar were made, so it can execute VIP instructions and ZOLE instructions.

Because the development of automated control flow and data flow analyses is in progress, all VIP and ZOLE optimizations were hand coded into the source code for this study. The modified programs were then compiled using the modified GCC tool chain [GCC04]. Using the default data sets, we ran full simulation on all benchmarks to their completions without skipping any instructions. Hundreds of million to billion dynamic instructions were simulated for all benchmark programs. Due to compatibility issues between the some of the programs and the compiler tool chain, four programs: *pgp*, *rasta*, *ghostscript*, and *mesa* were dropped from this study.

CHAPTER 5

RESULTS AND ANALYSES

Experimental results are discussed in this chapter. We first presented a comprehensive evaluation on costs and benefits of FITS programmable decoder in area, access latency, and power consumption. Following the decoder analysis is the discussion on the effectiveness of FITS framework, in application level, using the following metrics: instruction mapping rate, code size saving, power reduction, and performance measurement. Metrics are presented in a progressively order so any cause-effect relationships could be clearly established and results could be easily assimilated.

5.1 FITS Programmable Decoder Evaluation

To understand the cost of incorporating programmable instruction decoding to a FITS processor, we compared the area, access latency, and power consumption of both fixed and programmable decoders. In all figures, three lines were plotted: ROM represents the data points of a regular fixed instruction decoder; FITS represents the data points of the FITS programmable instruction decoder; Overhead represents the overhead

associated with the FITS programmable decoder. Overhead is expressed in percentage difference, which is computed with the following formula:

$$Overhead = \frac{FITS - ROM}{ROM} \times 100\ \% \tag{5.1}$$

A positive overhead indicates additional costs for using FITS decoder; a negative overhead indicates achievable savings through using the FITS decoder.

5.1.1 The Size of a Decoder

The first step of evaluation was to determine the size of a decoder being examined. The size of an instruction decoder is determined by the product of the number of entries stored (row dimension) and the width of each entry (column dimension). Every decoder entry stores the datapath control information of one instruction. The size of the row determines the number of distinct instructions that a decoder can accommodate. The size of the column determines the number of control lines an instruction can regulate.

To show how the FITS decoder scales with a wide range of number of instructions, we plotted data points against the number of instructions ranging from 64 to 4096. This is same as looking at how well the programmable decoder can scale for opcode field width ranging from six bits to twelve bits. This range was chosen because many popular ISAs today have their top-level opcode field, the second-level opcode field, or the combination of opcodes from both levels, fall into this range. For example, MIPS

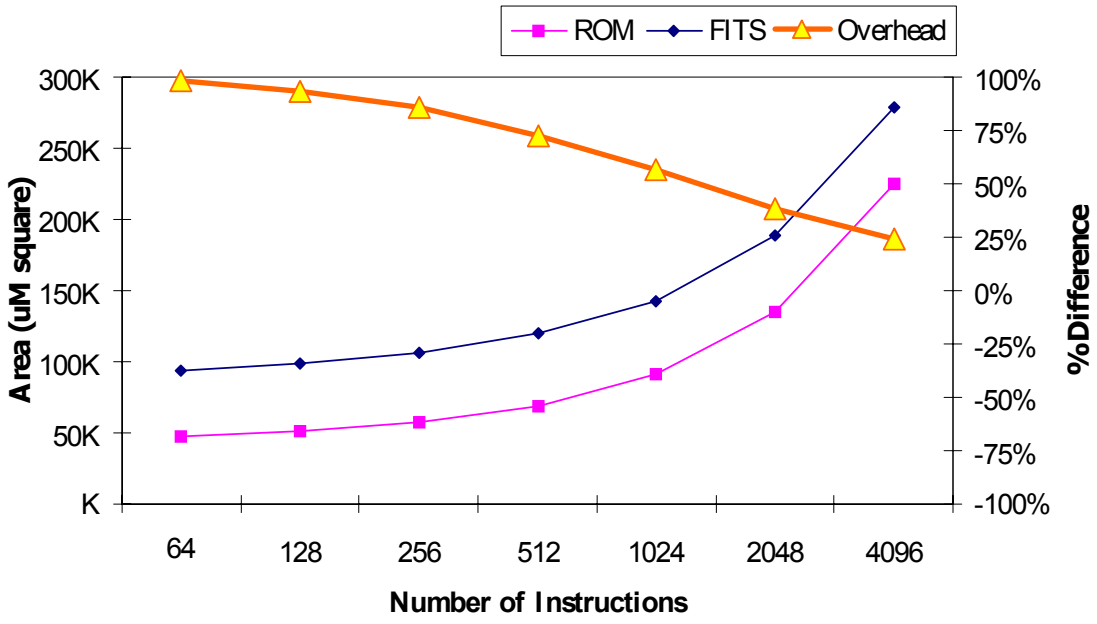

Figure 5.1: Area Comparison between Fixed and FITS Decoders

has 6-bit top-level opcodes and 6-bit second-level opcodes [MIPS-MIPS32]; ARM has 4-bit top-level opcodes preceded by a 4-bit conditional execution field, and 4-bit second-level opcodes [ARM-ARM]; Alpha has 6-bit top-level opcodes and 11-bit second-level opcodes [Compaq-Alpha].

Both the FITS decoder and an ordinary ROM decoder have entry size starting at 32 bits for 64 instructions. The entry size will increase by 1 bit every time we double the number of instructions available on chip. To put this dimension into perspectives, there are approximately 120 control lines in the integer datapath of Intel's Pentium 4 processor [PattersonHennessy05]. For its floating-point datapath, the number of control lines ranges between 275 to over 400 – the latter number for including the SSE2 instructions [Intel-IA32]. This is undoubtedly overkill for any embedded processor design.

5.1.2 Footprint Area Analysis

Figure 5.1 shows the footprint area, in micron (μm) squared, of the fixed decoder and the programmable FITS decoder. The footprint area shown includes the core area, power ring and pin spacing areas. The area of the FITS decoder is computed by adding the area of the 16-entry SRAM and the area of the ROM used for initialization. A 16x32 SRAM (for 64 instructions) is less than 47K micron squared and a 16x38 SRAM (for 4096 instructions) is less than 54K micro squared when implemented using the TSMC's 0.18 μm process. The additional area associated with programmable decoder is very small compared to the total chip area, which generally ranges from tens to hundreds of millimeter (mm) squared under the same process technology. Moreover, this area overhead is scaling down as the number of instructions supported increases. As shown in the figure, the overhead starts out to be 98% for 64 instructions and it drops down to only 23% for 4096 instructions. This is because while the number of entries in the ROM decoder needs to increase along with the increasing number of instructions, the number entries in the SRAM can be kept the same.

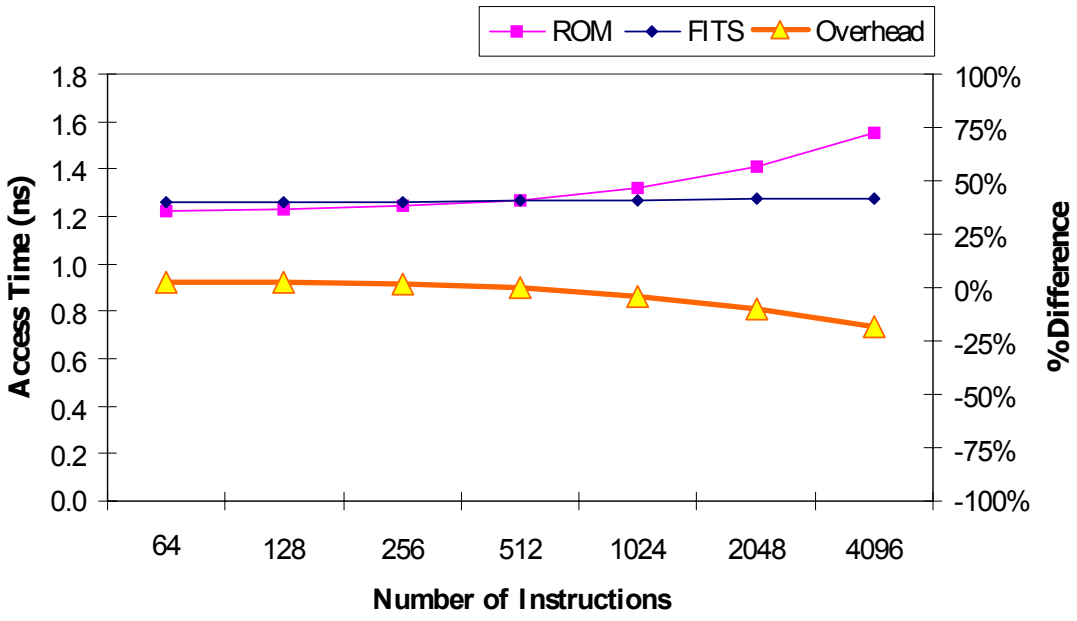

Figure 5.2: Access Time Comparison between Fixed and FITS Decoders

5.1.3 Access Time Analysis

Figure 5.2 shows the access time, in nanoseconds (ns), of the fixed decoder and the programmable FITS decoder. Access time is defined as the slowest possible input-to-output transition for accessing a critical path. The access time overhead for using the FITS programmable decoder is small: the worst case has less than 3% overhead when the ROM decoder is small (64 entries). Moreover, this access time overhead decreases down to less than 1.4% when the number of instructions reaches 256, after which accessing the FITS programmable decoder become faster than accessing the ROM decoder: 0.4% faster for 512 instructions; 4% faster for 1024 instructions; 10% faster for 2048 instructions, and 20% faster for 4096 words. Most important of all, with the processor clock frequency

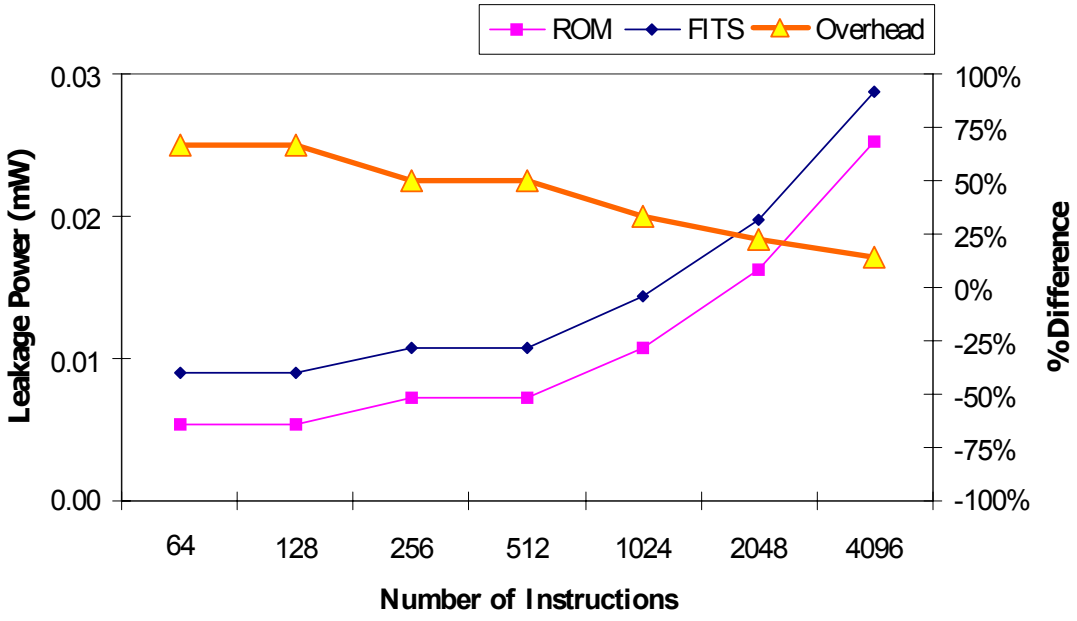

Figure 5.3: Leakage Power Comparison between Fixed and FITS Decoders

targeted at 250 MHz, all read and write accesses to the FITS programmable decoder can be easily finished within one cycle.

5.1.4 Power Consumption Analysis

The leakage power, dynamic power, and total power consumption, in milliwatts (mw), of the fixed decoder and the FITS programmable decoder are shown in Figure 5.3 and Figure 5.4 respectively. The dynamic power is calculated by multiplying the dynamic AC current by the operating voltage. The dynamic AC current assumes 50% read and write operations, where all addresses and 50% of input and output pins switch. Likewise, the leakage power is the product of the operating voltage and the standby leakage current, which assumes inactive memory cells with all input and output pins being held stable. The total power consumption is the sum of dynamic and leakage power consumed. Both

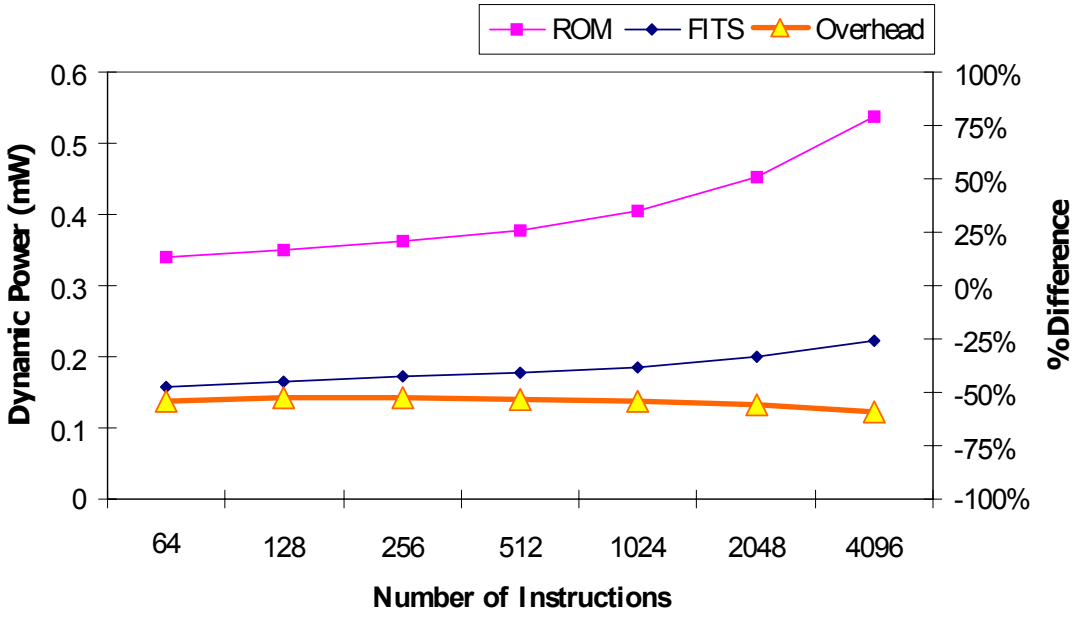

Figure 5.4: Dynamic Power Comparison between Fixed and FITS Decoders

dynamic and static power consumptions of FITS programmable decoder presented here includes the leakage power consumption of the back-up ROM decoder. Through clever dynamic power/energy management techniques [Magklis03][Huang00], it is possible to clock gate the rarely accessed back-up ROM decoder to achieve further power savings by reducing its leakage power. Yet, the discussion of applying these techniques is beyond the scope of this thesis.

Similar to the area and access time analyses shown in previous sections, the FITS programmable decoder shows a nice scaling effect in leakage power consumption. As shown in Figure 5.3, the leakage power overhead starts with 66% and gradually decreases down to 14% as the number of instructions supported increases from 64 to 4096.

Unlike all the analyses shown above where positive overheads were observed, the dynamic power consumption of the FITS programmable decoder is less than that of a regular fixed decoder as indicated by negative overhead line. As depicted in Figure 5.4, a

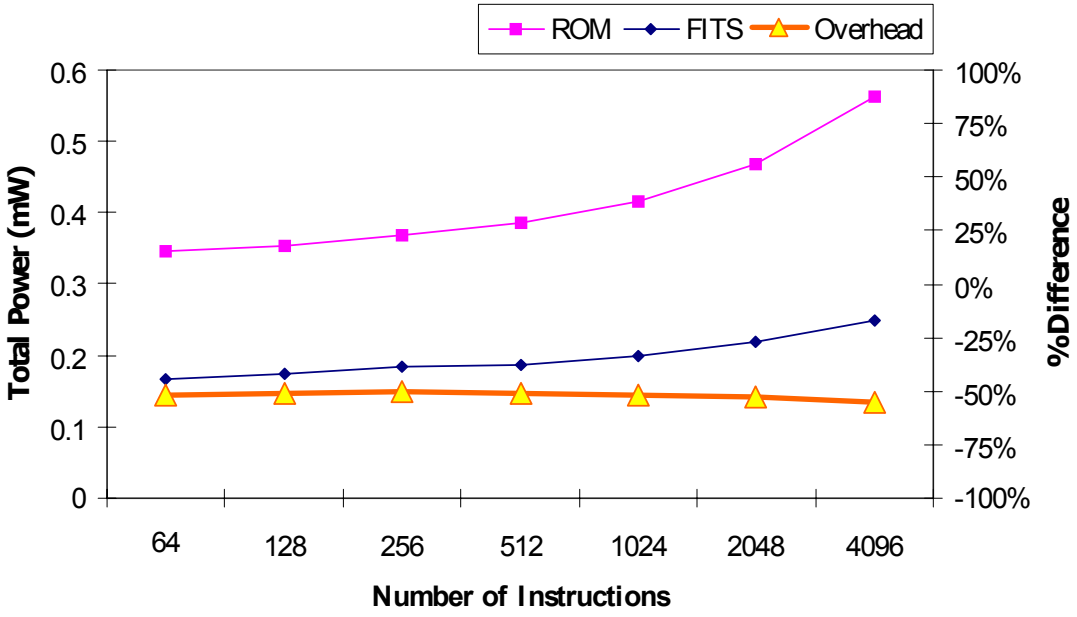

Figure 5.5: Total Power Comparison between Fixed and FITS Decoders

54% to 59% of dynamic power savings can be achieved by FITS decoder as the number of instructions supported increases from 64 to 4096. These power savings are due to the fact that FITS decoder mostly accesses the small 16-entry SRAM during program execution; whereas a regular fixed decoder needs to access a much bigger sized ROM that consumes more power to operate.

The observation to be made here is the scale of leakage power is an order of magnitude smaller than that of dynamic power. This means the positive overheads of leakage power shown in Figure 5.3 will not negligible effect for the total power consumption, which is computed by adding leakage power with dynamic power. Figure 5.5 depicts the total power consumption. As expected, the total power consumption of the FITS programmable decoder is dictated by the dominant dynamic power consumption. As indicated by negative overhead lines in Figure 5.5: 52% to 56% of total power savings

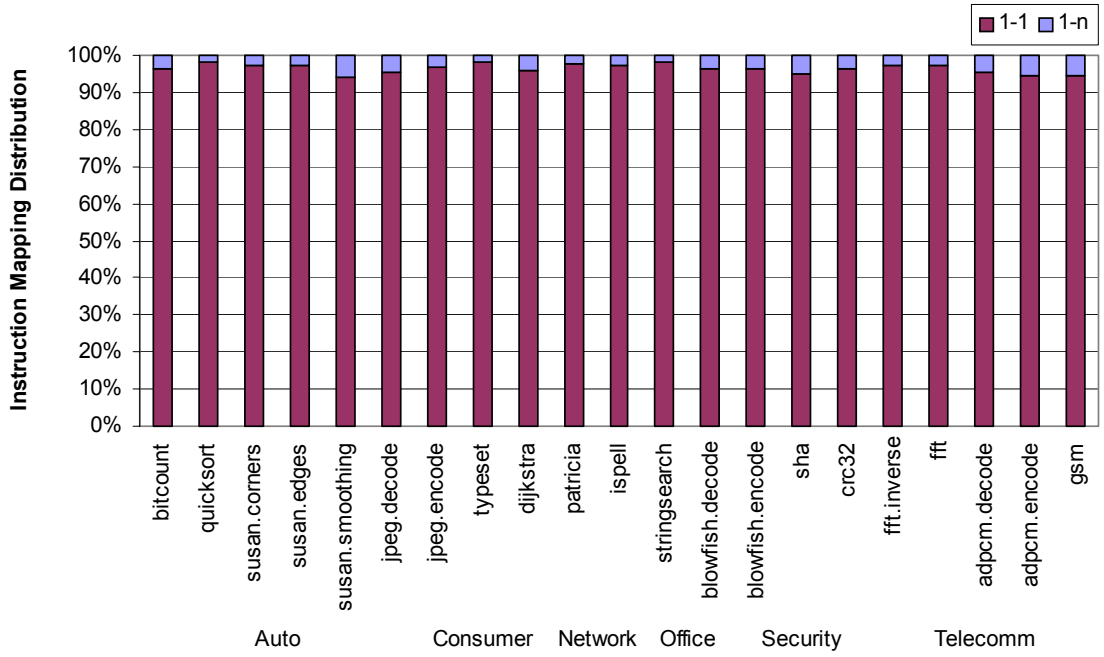

Figure 5.6: Static ARM-to-FITS Instruction Mapping

can be achieved by FITS decoder as the number of instructions supported increases from 64 to 4096.

5.2 Instruction Mapping Coverage

In order for FITS to demonstrate any noticeable power and code size benefits, enough one-to-one translations must be made from the native 32-bit ARM instructions to the optimized 16-bit FITS instructions. This section demonstrates the reality of FITS with its promisingly high one-to-one correspondence to ARM: a 96% average of static mapping and a 98% average dynamic mapping, as illustrated by Figure 5.6 and Figure 5.7 respectively.

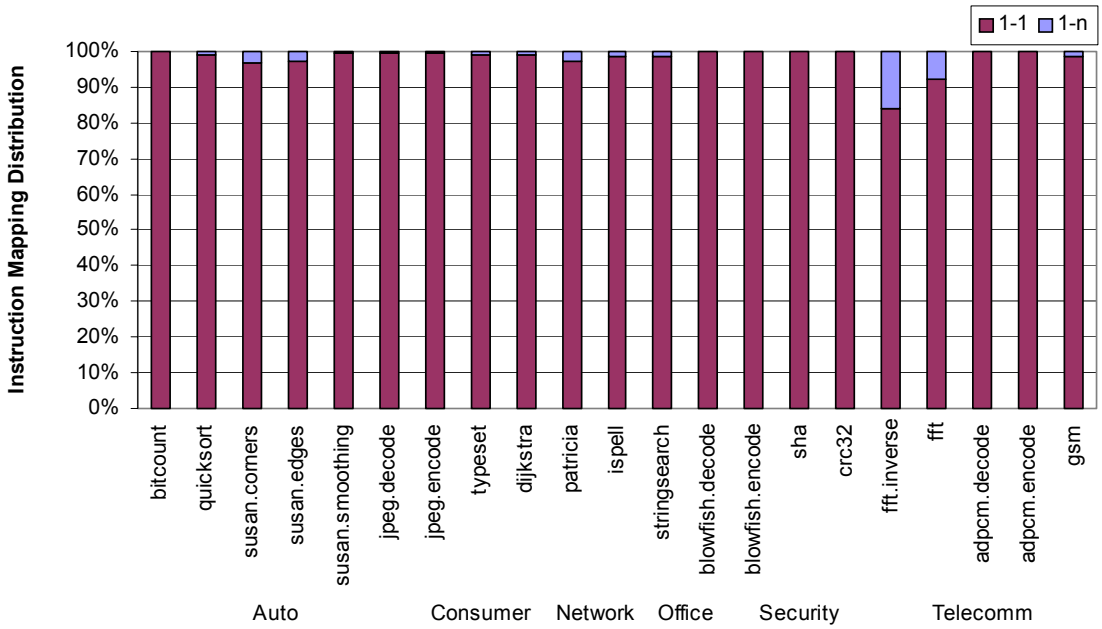

Figure 5.7: Dynamic ARM-to-FITS Instruction Mapping

Higher static mapping gives us smaller code size and fewer cache misses. Higher dynamic mapping means greater power reduction and faster execution. The mapping is determined to be one-to-one if there was a FITS instruction that could achieve the same result as an ARM instruction. Otherwise, a one-to-*n* mapping, where $n > 1$, is determined when we had to translate this ARM instruction into multiple FITS instructions. In theory, *n* could be any number ranging from 2 to 4; however, in practice, $n = 2$ is almost always the case. The lower dynamic one-to-one mapping of *fft.inverse* and *fft* is due to a larger fraction floating-point code executed which are currently not accounted for by the FITS.

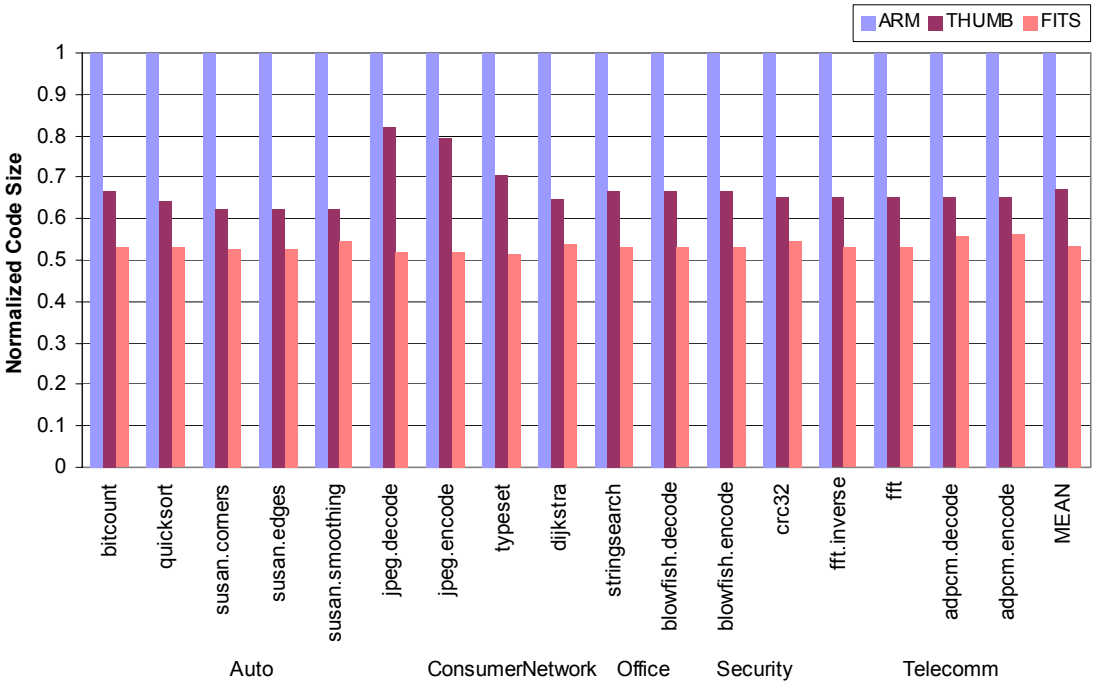

Figure 5.8: Code Size Comparison between ARM, THUMB, and FITS

5.3 Code Size Benefits

Figure 5.8 compares the program code density achieved by different code generations, namely, ARM, THUMB, and FITS. The FITS bars represent the program code size after the ARM-to-FITS translation. The ARM and THUMB bars represent the program code size compiled in pure 32-bit ARM and 16-bit THUMB respectively. ARM-THUMB intermixing result was omitted since FITS is a pure 16-bit instruction synthesis technique and ARM-THUMB intermixing does not yield better code density than that of THUMB alone. We normalized everything with respect to ARM in order to show the code size savings that THUMB and FITS each achieves in terms of percentages. On average, THUMB reduced approximately 33% of ARM code across the entire benchmark

suite. For *jpeg* where there are many expensive image compression and decompression routines, THUMB can only reduce code size by approximately 20%. On the other hand, FITS was able to reduce the ARM code by almost an half: on average, 47% of total ARM segment could be eliminated. The reason for THUMB not being able to achieve as much code size savings as FITS does is because THUMB is not able to utilize its 16-bit instruction fields as efficiently due to its general-purpose nature. Thus, for an application that has several performance critical regions, such as *jpeg*, many 32-bit ARM instructions would still need to remain in the program to handle the expensive processing.

Like most general-purpose ISAs, THUMB supports a wide range of instructions in order to be able to specify lots of applications. However, this general-purpose capability requires more opcode space and makes the other instruction fields, such as register and immediate operands, smaller. When the register operand width is reduced, the processor can specify less architect registers and thus increasing the register pressure. Higher register pressure causes more spilling and thus increasing the number of memory references in the program. This is the reason why THUMB is not able to achieve the level of code size savings that FITS gives.

This code size saving achieved by FITS does not come at expense of performance lost as illustrated by the performance results later. This is mainly due to the following two reasons. First and foremost, FITS aggressively optimized and adopted the utilization-driven synthesis heuristic, which makes it very effective in determining the target instructions for synthesis without any noticeable performance lost. Second, the resultant half-sized FITS code effectively makes the L1 instruction cache almost twice as large as before. Thus, the FITS execution core was able to take advantage of higher spatial

locality exhibited to largely raise the cache hit rate, and so does the overall execution speed.

5.4 Power Dissipation Benefits

The best way to reduce overall chip power dissipation is to attack each of the microarchitectural components using power. In this study, we focus on attacking instruction cache power consumption. We start by showing the breakdown of instruction cache power for each of the four processors under simulation. Next, we present the power reduction that FITS is able to achieve in each of the component powers: switching, internal, leakage, and peak powers. The reduction of each component power is then translated into the total instruction cache power reduction. Finally, the instruction cache power savings is mapped into the corresponding overall chip-wide power saving.

As mentioned in the chapter of evaluation methodology, we model only dynamic and static power dissipation. The dynamic power was further broken down into switching power and internal power to better facilitate monitoring power reduction by FITS. The switching power is the power consumed by the output driver and its output load capacitance of the instruction cache microarchitecture. The internal power is the dynamic power of the instruction cache microarchitecture itself. Therefore, the switching power is sensitive to the power consumed by the amount of output data during each cache access, or switch. On the other hand, the internal power is sensitive to the overall power consumed by the entire cache logic block when it is on; hence it is highly dependent upon the total size of the cache.

The static power, or leakage power, is sensitive to the power lost due to leakage current of each gate of cache logic block; thus it is also dependent upon the total size of the cache. The peak power depends both on the microarchitectural configuration of a cache, such as block size and total cache size, as well as the characteristics of the instruction address stream from each individual cache access.

Energy savings in both instruction cache and system chip could be directly inferred from the corresponding power reduction; hence they are not explicitly shown here. The validity of this energy saving inference comes from the fact that all the execution cores run at a fixed operating frequency and the difference between their simulation times was not significant. Since energy is the product of power and time, without too much difference in the time component, the ratio of energy saving would roughly have identical distribution as the ratio of power saving.

There are many results to be presented in this power consumption study. To facilitate easy assimilations, we show only average results collected from the entire benchmark suite to simplify the information presented here for better digestion. We move all individual detailed results to the appendices section at the back of this dissertation for interested readers to pursue.

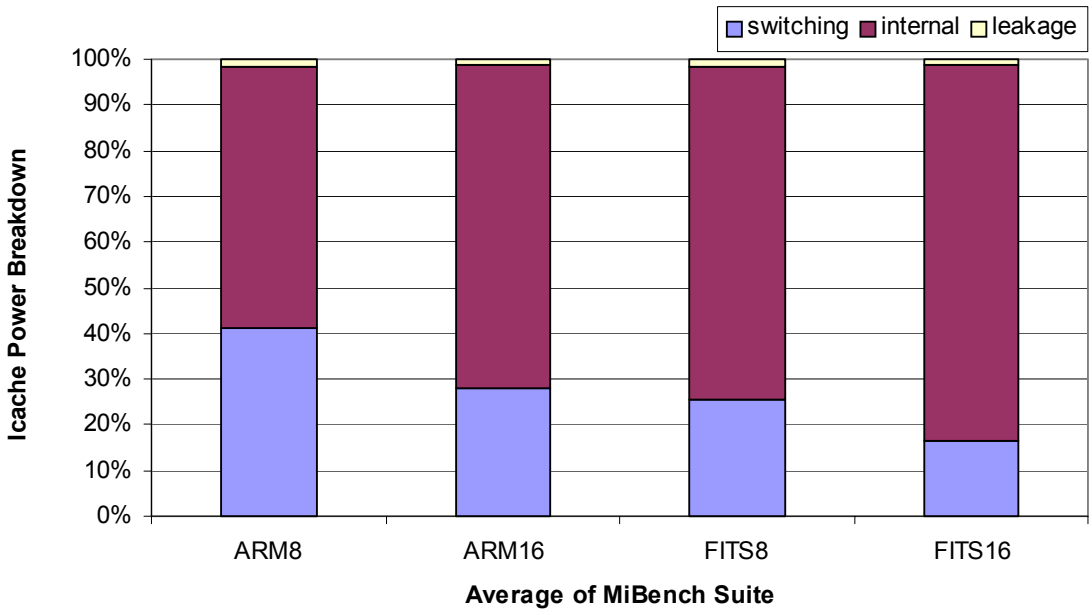

Figure 5.9: Instruction Cache Power Breakdown for ARM and FITS

5.4.1 Instruction Cache Power Breakdown

From the instruction cache power breakdowns shown from Figure 5.9, the following power usage trends are noticed. First, the total instruction cache power is dominated by the dynamic power, i.e. the switching power plus the internal power. This is expected since SA-1100 is a relatively low-end embedded microprocessor built with less aggressive fabrication technologies (e.g. 0.35µm), we would not encounter the same level of leakage current problem found on current state-of-the-art high-end designs fabricated with deeper sub-micron technology.

Second, as the size of the instruction cache increases, the percentage of switching power goes down; the percentage of internal power goes up; the percentage of leakage power remains approximately the same. The reason is larger cache consists of more gates

and thus more internal and leakage power. In addition, given the same cache block size and associativity, larger cache would yield better hit rate, which means less gate switches and the switching power is reduced.

Third, with the instruction cache size being equal FITS uses lower percentage of switching power, higher percentage of internal power, and approximately the same percentage of leakage. The leakage power percentage stays unchanged because the numbers of gates in caches of same size are equal. The reduction of switching power percentage is due to the increased cache hit rate of FITS-sized code. Since the cache size is the same, the increase of internal power percentage is due to the normalization effect after accounting for the reduction of switching power percentage.

Last, if we compare the switching power percentage between ARM8 to ARM16 and ARM8 to FITS8, we will find that applying FITS transformation reduces more switching percentage than simply doubling the size of cache. Considering this with the fact that the FITS reduction comes solely from the increased cache hit rate as opposed to the joint effect of increased internal power seen in ARM16, it implies that FITS can reduce switching power more effectively than doubling the size of the cache. This speculation is confirmed by the instruction cache power saving analysis that follows.

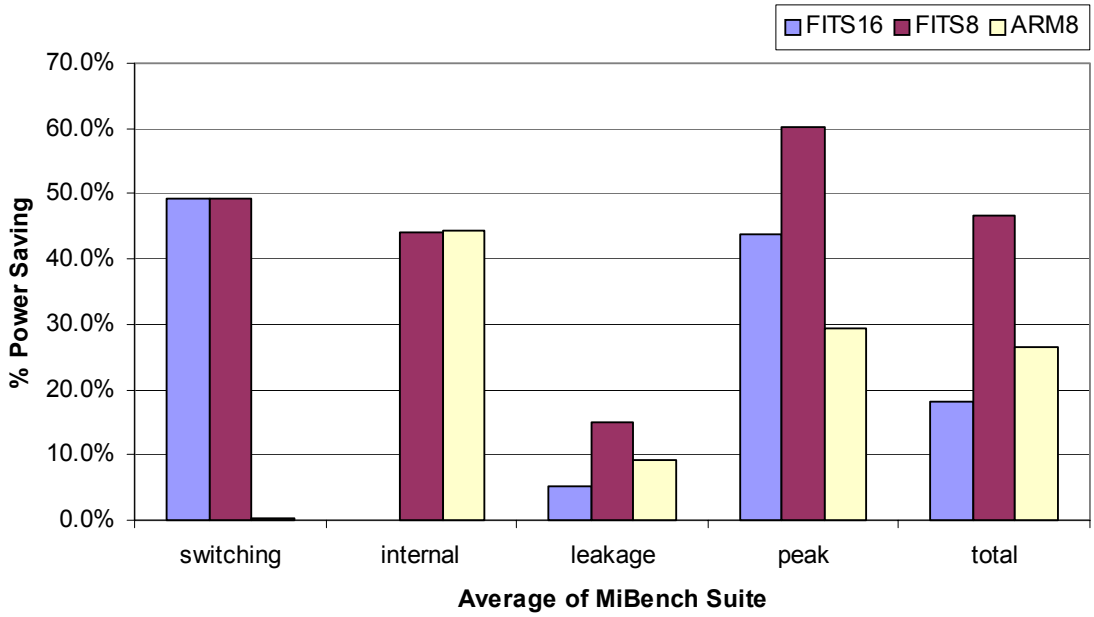

Figure 5.10: Instruction Cache Power Savings of FITS

5.4.2 Instruction Cache Power Saving

To see how FITS optimizes the power usage of an instruction cache, it is best to look at the power reduction in each power component broken down as shown in Figure 5.10. We compare the power saving from a 16 Kb and an 8 Kb FITS caches (FITS16 and FITS8) with the default 16 Kb ARM cache in the SA-1100 core. The 8 Kb ARM cache (ARM8) is included to show that simply reducing the size of ARM cache is not going to help us much and we may have to pay more performance penalty than we can bear.

As speculated in the section of power breakdown analysis, FITS-sized codes benefit greatly from switching power reduction. This is the power saving that clearly distinguishes a FITS-optimized cache from a normal ARM cache. Both FITS16 and FIT8 save approximately 50% cache switching power while ARM8 saves virtually none. The

switching power saving of FITS is a result of better cache hit rate due to better spatial locality that FITS-sized codes exhibit. On the other hand, ARM8 consumed as much overall switching power as the baseline 16 Kb cache indicates the overall gate switching frequencies of the two caches are essentially the same.

For the internal and leakage powers, the two half-sized caches, FITS8 and ARM8, demonstrate nontrivial savings in most applications. This is because both internal and leakage powers are directly proportional to the number of gates given the same operational period. For the leakage power; however, exceptions occur for some applications where FITS8 or even FITS16 shows greater savings than ARM8. This is because the saving of smaller amount of logic gates in ARM8 were compromised or even wiped out by its longer operational period due to larger cache miss rates. This effect was hidden in the internal power results because internal power contributes to more than half of the total cache power in all four different cache schemes (see the cache power breakdown); therefore, the power loss due to longer operational period were simply absorbed.

The peak power consumption, depends on both switching frequency and amount of logic gates; therefore, we can observe savings from all three cache schemes: on average 46% for FITS16, 63% for FITS8, and 31% for ARM8. Since peak power is sensitive to factors that affect both the dynamic and the static powers, greater peak power saving of FITS16 and FITS8 indicate that FITS is a well balanced low power technique for instruction cache.

This claim is supported by the overall instruction cache power consumption results, which combine all the component savings above. As the figure shows, FITS8

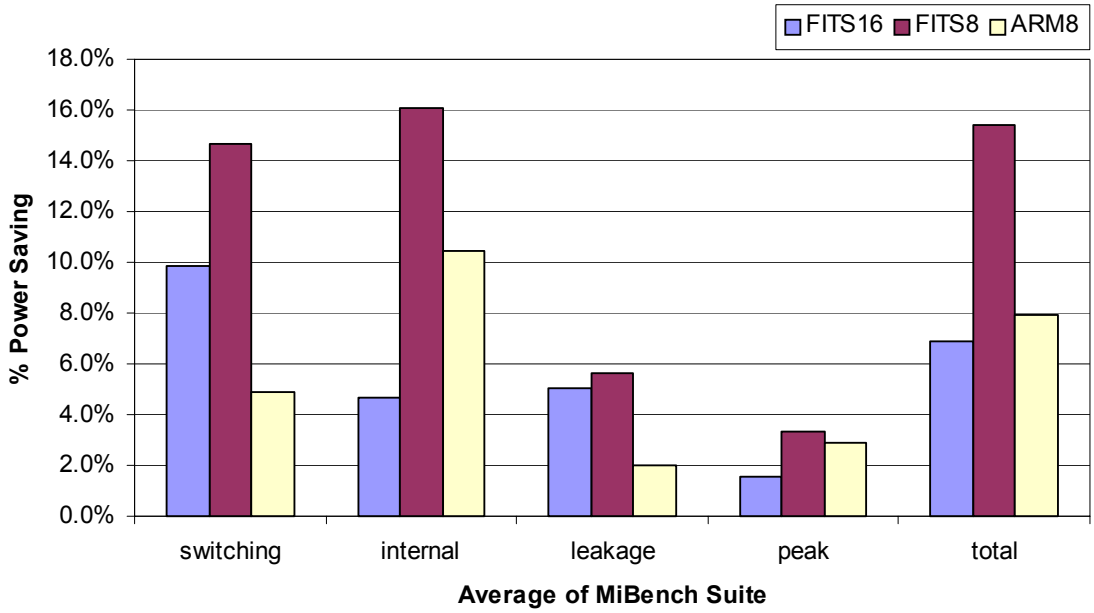

Figure 5.11: Chip-wide Power Savings of FITS

gives the highest 47% average total instruction cache power saving followed by ARM8 and FITS16 with each saves 27% and 18% respectively.

To see how effective does FITS reduce the total chip power, Figure 5.11 illustrates how these instruction cache power savings would be translated into the total chip power savings. It provides the corresponding chip-wide power savings for instruction cache power savings presented in Figure 5.10. As shown in Figure 5.11, FITS16 and FITS8 save approximately on average 10% and 15% chip-wide switching power respectively while ARM8 saves 5%. For the chip-wide internal power savings, FITS16 and FITS8 save approximately on average 5% and 16% respectively and ARM8 saves 10%. For leakage power savings, both FITS16 and FITS8 save approximately on average 5% to 6% while ARM8 saves 2%. For peak power savings, on average, FITS16 saves more than 1.5%; FITS8 saves more than 3%; ARM8 saves over 2%. At last, for the

overall chip-wide total power saving, on average, FITS8 saves 15%; FITS16 saves7%; ARM8 saves 8%.

As shown by the individual detailed results in the appendix, FITS has demonstrated exceptional power savings on *patricia*. *Patricia* uses special tree data structure to represent routing tables for IP traffic in network applications. This special tree data structure, or a *patricia* tree, is used in place of a full tree with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the tree to reduce traversal time at the expense of code complexity. The increased code complexity has translated into more cache misses for normal ARM code, which resulted in increased power dissipation. On the other hand, improved program spatial locality seen on FITS caches neutralized the increased code complexity effect of using a *patricia* tree: cache miss rates were significantly improved for caches loaded with FITS code. Other applications, such as *typeset* and *stringsearch*, have demonstrated similar power savings for the same reason of much improved cache miss rates. *Typeset* is a front-end typesetting tool for HTML. It captures the processing required to typeset an HTML document, and is a major core component of a web browser. *Stringsearch* searches for given words in phrases using a case insensitive comparison algorithm. Both applications took the advantages of much increased spatial locality rendered by dense FITS codes. Please refer to following section for more discussion on cache miss rate results.

5.5 Performance Benefits

To demonstrate that FITS does not save power at the expense of performance; we include the following performance results. Performance is measured in both instruction cache miss rates and instructions per cycle (IPC) rates. The cache miss rate analysis helps to explain why simply reducing the cache size of the default ARM cache does not reduce much power. The IPC analysis gives an idea of overall FITS performance compared to the ARM. Both results showed that FITS saves power without compromising performance. Looking this section together with the power results from the last section, we observe that reducing the regular sized cache to 8 Kb not only hurts performance as measured by high miss rates and low IPC, it also just shifts power use. On the other hand, 8 Kb caches for FITS have no more misses than 16 Kb for ARM.

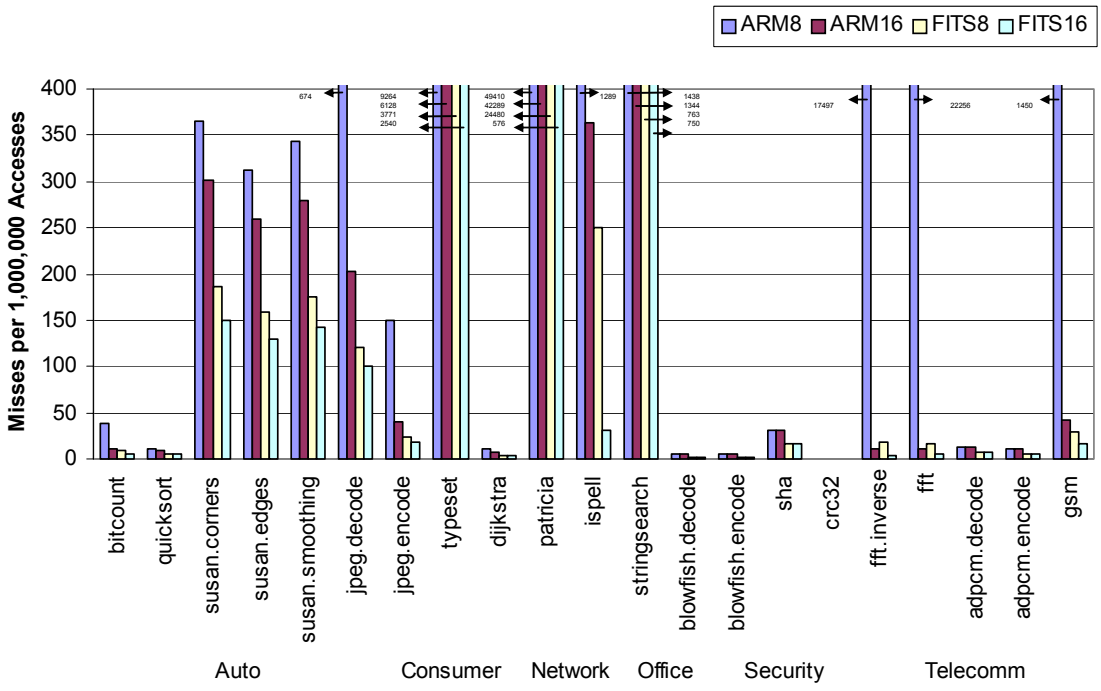

Figure 5.12: Instruction Cache Miss Rate

5.5.1 Cache Miss Rate Evaluation

Figure 5.12 shows the instruction cache miss rates for all four processor configurations. The miss rate was measured as misses per one million cache accesses since most of the benchmarks are easily cacheable due to their small code size footprint. Values that are too large to be displayed are marked with their real miss rate numbers on the side. FITS surpassed ARM with greatly improved cache performance: the half-sized FITS8 caches have smaller miss rates than the normal full-sized ARM16 caches. This is due to the better spatial locality exhibited by FITS-sized code. Since the instructions are half the size, the cache lines can be viewed as being twice the size (this operates much like a next line prefetch on cache miss since twice the number of instructions are brought

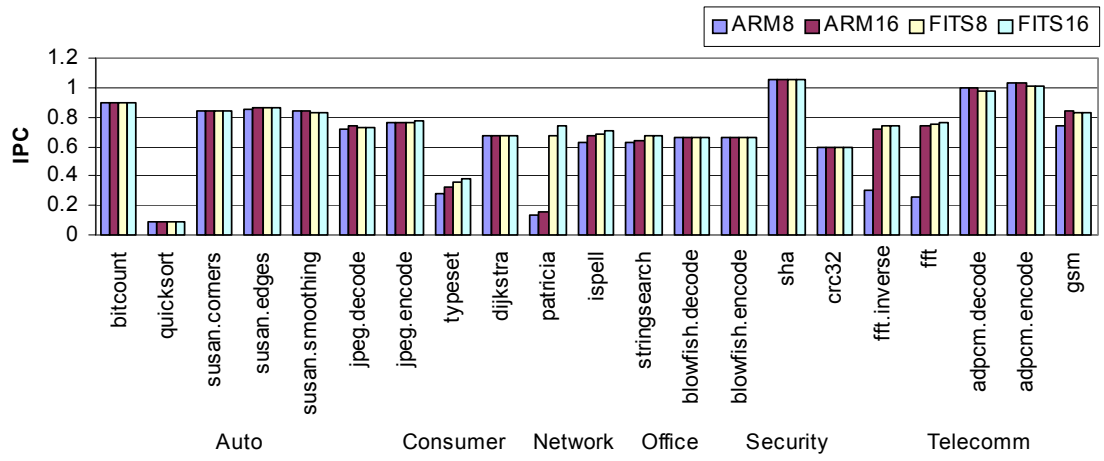

Figure 5.13: Instruction per Cycle (IPC) Rate

into the cache (i.e. fewer compulsory misses and for displaced lines, fewer conflict misses to restore the instructions). Moreover, since embedded applications are typically stream-based, most branches in MiBench are easily predictable. Therefore, this instruction "packing" effect makes FITS caches seem virtually twice as large as their true physical size.

5.5.2 Instruction per Cycle (IPC) Rate Evaluation

Figure 5.13 shows the IPC performance measures for all four processor configurations. Since the SA-1100 simulated core is a dual-issue, in-order machine, the highest IPC possible is 2. Overall, the IPC for all four configurations are satisfactory. This is the result of the easy predictability and cacheability of MiBench programs. As expected, the IPC performance of FITS codes is comparable to that of native ARM codes. It is interesting to observe that an 8 Kb FITS cache could achieve roughly the same IPC as a 16 Kb ARM cache with only few minor variations. We expect FITS to be

performance neutral, but consistently find a small improvement, and in some applications, a large improvement (e.g. *patricia*). This is due to increased instruction cache locality exhibited from packed FITS code.

5.5.3 VIP and ZOLE Speedup Evaluation

The performance is evaluated in terms of total dynamic execution time. The baseline processor is a single-issue, in-order five-stage MIPS pipeline. Any popular processor could have been used, but the MIPS is a common basis of research within the academia community. The other three processor configurations are the baseline augmented with VIP processing units; the baseline augmented with ZOLE unit; and the baseline augmented with both VIP and ZOLE units. This is to mimic real life scenarios where there can be different chip area constraints specified by customers. If there are enough spare transistors left, we may as well include both VIP and ZOLE units; or otherwise, we must make clever tradeoffs based on careful analyses and thorough evaluations.

Figure 5.14 depicts the performance speedup benefits for each FITS optimization. For VIP optimization alone, the performance improvement was as much as 39% (1.39x speedup) as demonstrated by mpeg2decode and a 12% average performance improvement was achieved throughout the entire MediaBench suite. For LOOP optimization alone, the performance improvement was as much as 64% (1.64x speedup) as demonstrated by epic and a 19% average performance improvement was achieved throughout the entire benchmark suite. For VIP and LOOP optimizations combined, the

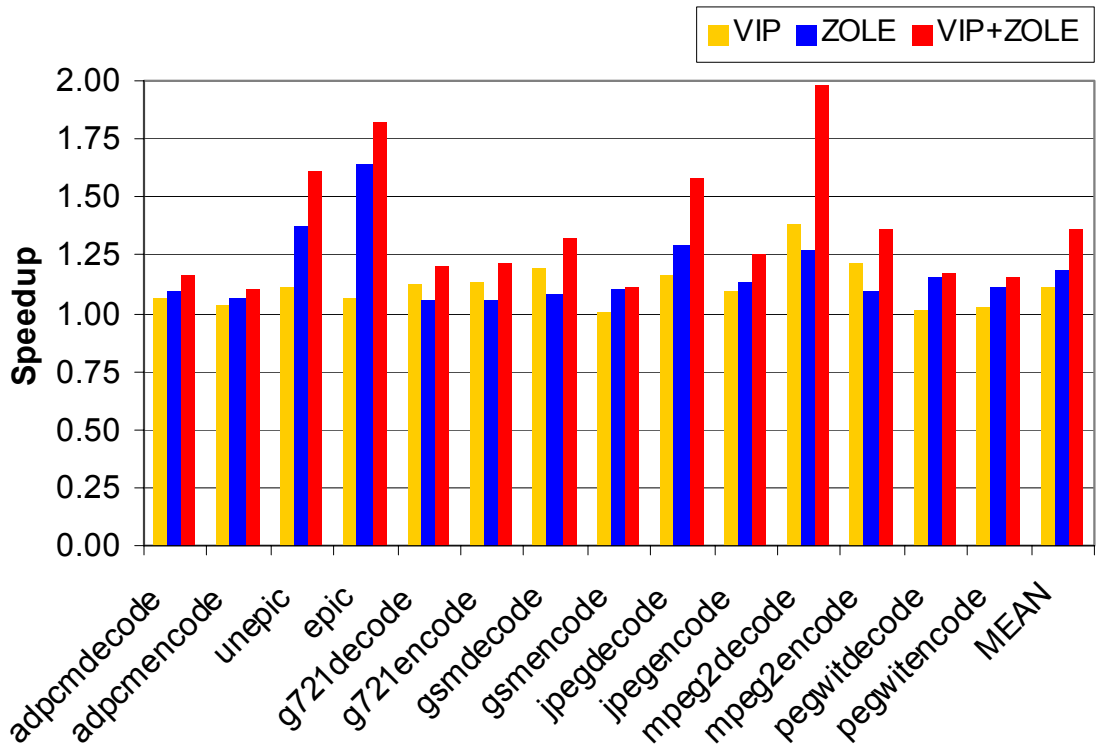

Figure 5.14: Performance Speedup Achieved by VIP and ZOLE

performance improvement was as much as 99% (1.99x speedup) as demonstrated by mpeg2decode and a 37% average performance improvement was achieved benchmark wide. All optimizations were applied to subroutines that account for more than 1% of total run time. Once a FITS aware compiler is fully developed, optimization can be automated to exploit more opportunities to achieve even greater speedups.

This speedup result demonstrates significant performance improvements of FITS. Yet, there is still much potential remained to be explored. Specifically, all speedup presented above were obtained from original algorithm without modification. These algorithms were written by programmers without the knowledge of FITS's powerful and versatile microarchitectural support. It is clear that if programmers are aware of the FITS's VIP and ZOLE supports, further speedup gains can be easily achieved with more

aggressive optimization at the algorithm level. Instead of trying to approximate a multiply-divide operation with a long series of adding, subtracting, and shifting operations plus conditional tests, programmers can write multiply followed by divide directly, knowing that it will be mapped into one of the VIP instructions. Besides potential significant speedup, this algorithm level optimization can also further reduce the code size, which can lead to on-chip and off-chip memory power reduction and better cache performance.

CHAPTER 6

RELATED WORK

This chapter describes prior and concurrent research related to FITS. These relevant researches are presented under different categories based on the nature of the work. Along with detailed description and careful analysis, what is also provided are insights explaining why FITS is advantageous compared to these work.

6.1 Microprogramming

One of the early approaches to reduce code size was microprogramming [Wilkes53]. A microprogram, or microcode, is a program that is made up of macro instructions which consist of several microinstructions. Each CISC-like macro instruction fetched from memory caused a sequence of microinstructions to be fetched and executed. Each microinstruction defines the set of datapath control signals that must be asserted in a given state. Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction. Microcode saves time by allowing to fetch fewer instructions from the main memory. This work differs from microcode in several ways,

including that specific instructions within the programmable decoder can be individually referenced and that the instructions in the programmable decoder can be changed for each executable binary as well as within each executable binary on the fly.

6.2 Code Compression

Embedded applications must execute under constraints of limited memory and low energy consumption. Instruction caches have been recognized as a major source of energy consumption in embedded systems. In [Kadri03], Kadri et al. observed that energy consumption and program execution time are very sensitive to the level one instruction cache size. One way to address this issue is to compress the code within a program, which can decrease the number of cache misses due to a smaller footprint of instructions being accessed. IBM's CodePack technique [IBM-CodePack][Orpaz02] included in its PowerPC processors [IBM-PowerPC] used Huffman tables to compress cache blocks. Xie et al. [Xie01] proposed a code compression algorithm based on arithmetic coding in combination with a pre-calculated Markov model. These code compression schemes compress all instructions in the program. Thus, the decompression overhead occurs at every instruction fetch. Benini et al. [Benini99] and Lekatsas et al. [Lekatsas00] proposed selective instruction compression. They proposed dictionary-based code compression algorithm to compress frequently appearing instructions. Only the most commonly executed instructions are compressed, while other instructions of the code are left uncompressed. These code compression approaches have the disadvantage of complicating instruction fetch and decode logic since instructions can differ in size.

Instruction reuse is another popular approach to reduce code size. Procedural abstraction [Debray00] is a compiler optimization, which identifies common code sequences and abstracts them into procedures. The original sites of each code sequence are replaced with function calls. A hardware extension of this technique is to use echo instructions [Lau 03]. An echo instruction indicates where the abstracted code sequence is located and the number of instructions to be executed. Unlike conventional procedure calls, echo instructions do not call returns at the end of the abstracted sequence. Another advantage of this approach is that abstracted sequences can overlap to further facilitate code reuse. The main disadvantage of both procedural abstraction and echo instruction is that the overhead of executing calls and returns for each abstracted code sequence will usually slow down the program execution. Spatial locality may also be reduced, which may decrease cache performance.

More recently, Hines et al. proposed instruction packing technique to reduce code size. [Hines05] Instruction packing removes instruction fetch cost by placing frequently occurring instructions into special registers, just as frequently accessed data are kept in registers through compiler register allocation. The advantage is that code size is reduced without use of large dictionaries. The difference between instruction packing technique and FITS is at the instruction decode. FITS utilizes a programmable instruction decoder to achieve application-specific customization effect by allowing a subset of instructions implemented in the microarchitecture to be mapped to the ISA for each different application. The advantage to make decoder programmable is that all instructions are half-sized (16-bit long) and native: there is no need to decompress or unpack an

instruction before its corresponding control signal can be fetched from the decoder and passed down to pipeline.

6.3 SIMD Architecture

Single-instruction multiple-data (SIMD) architecture is a popular technique to exploit parallelism in performance demanding applications. The ability to process multiple data in one instruction makes SIMD architecture attractive for programs that have large vectors and matrices. Recent works such as IMAP [Kyo05], Imagine Stream [Ahn04], and Synchroscalar [Oliver04] have demonstrated the potential of applying SIMD in high-performance embedded workloads. Nevertheless, it is very difficult to keep all generously allocated processing elements occupied all the time when a program, or regions of a program, lack of "embarrassingly parallel" ILP. The leakage power consumed by idle processing elements can add up to significant energy waste. Moreover, extending any regular microprocessors with SIMD not only requires adding new hardware support but also a new programming model. Adding new programming model requires a complete software tool chain to support, which is usually not economically feasible for many vendors. These drawbacks limit SIMD from being widely deployed to many real world applications.

6.4 Zero-Overhead Loop Execution

For many applications, a large percentage of the dynamic program execution time is spent in small program loops [PattersonHennessy03]. These loop execution incur significant overhead due to the updates of loop counters and the branches to initiate a new iteration. Many software code transformation techniques have been proposed to improve loop execution time. For instance, both software loop unrolling [Davidson96] and software pipelining [Intel-IA64] are popular techniques to decrease loop overhead. Nevertheless, these approaches suffer from drastic increase in code size. Space increasing transformations, such as loop unrolling or software pipelining, are often unacceptable for many embedded and DSP applications that have strict code size and power requirements.

On hardware side, it is popular to use branch prediction to reduce branch misprediction penalty and superscalar or VLIW execution to allow other instructions to execute in parallel with the loop overhead instructions. However, the use of complex hardware to reduce branch overhead can often cause more power consumption, which is also not acceptable in embedded and DSP processors that have strict power envelope.

Branch folding is another architectural technique to reduce branch misprediction penalty. It is used in AT&T CRISP microprocessor [Ditzel87] and in IBM PowerPC architecture [IBM-PowerPC]. By performing early decoding of branch instructions in the fetch stage and looking ahead on conditional branches to resolve them early, these resolved or “folded out” branches do not need to be issued to the execution pipeline; thus achieving the effect of a zero-cycle branch execution. Implementing this technique

requires (1) early branch decoding logic at front end; (2) condition code dependency checking on all the instructions being issued to the execution; and (3) logic for branch misprediction recovery. The associated hardware cost for implementing branch folding is prohibitively high for cost-sensitive embedded and DSP domains.

Several commercial DSP processors have special loop or branch instructions for removing the overhead associated with the loop control mechanism [Analog-ADSP21160][TI-TMS320C67][Motorola-MCore][Siemens-TriCore]. For example, the TriCore ISA has three special branch and loop instructions to handle program loops: JNEI, JNED, and LOOP. The JNEI and JNED instructions are like normal jump-not-equal instructions with an additional increment or decrement operations to update the dynamic loop count. The LOOP instruction can achieve zero execution time in all but the first and last iteration of the program loop with loop iteration count known at compile time.

Loop buffering is an architectural feature commonly found in many embedded and DSP processors. A loop buffer is a small dedicated instruction buffer placed between the execution core and a larger main instruction cache that allows efficient looping fetch. Instructions are supplied to the execution core either from the loop buffer or from the main cache. This buffer can be used to increase the speed of applications without increasing code size. Other buffering benefits include reduced power consumption due to more localized instruction fetch, accurate loop-back branch predictions, elimination of taken-branch stalls, and reduced memory bus contention if data and instruction fetch share the same memory bus.

The Lucent DSP16000 architecture [Lucent-DSP16000] includes a loop buffer that can hold up to 31 instructions. Two special instructions are used to control the loop buffer on the DSP16000: DO and REDO. The DO instruction can specify *n* instructions to be executed *k* times. The value of *n*, ranging from 1 to 31, indicates the number of instructions following the DO instruction to be placed in the loop buffer. The value of *k* represents the number of times the instructions within the loop buffer to be executed and it is compile-time constant than 128. If the loop iterations are bigger than 128, a dedicated register is used to encode the value. The first iteration results in the instructions following the DO to be fetched into the loop buffer. Ideally, the remaining *k*-1 iterations are serviced directly by the loop buffer. The REDO instruction is identical to DO, except that it executes the current existing instructions within the loop buffer *k* times. The advantage of this technique is that the back branch at the end of loop can be eliminated since the loop will be executed the predetermined number of times. The limitation of this technique includes: (1) it only can be applied to the innermost loop; (2) the number of loop iterations must be known at compile time; (3) the number of instructions in the loop cannot exceed the size of the loop buffer; (4) there cannot be other internal branches (e.g. if-else blocks) within the loop body other than the loop back branch.

The innermost loop restriction can be addressed by incorporating more loop registers to keep track of extra loop states. For example, the StarCore SC140 processor [StarCore-SC140] includes a loop buffer and four sets of loop registers to allow for four levels of nested counted loop execution. On the other hand, the STMicroelectronics ST120 [ST-ST120] provides hardware support for up to three loops, which may be nested, overlapped, or independent of one another.

To address the unknown loop iteration count issue, Lee et al. took a hardware approach to dynamically detect and capture small tight loops with short backward branches [Lee99]. J. Rivers et al. extended the work in [Lee99] by generalizing the loop controller to enable capturing loops that: (1) contain if-else blocks; and (2) bigger than the loop buffer [Rivers03]. While both techniques are capable of handling loops with unknown loop iteration count at compile time, they have to execute the last loop back branch in all iterations, even for those loops with constant loop iteration count.

The restriction of including additional transfer of control within loop body is partially addressed by applying compiler code transformation in [Uh99] and [Sias01]. Uh et al. showed that using many conventional compiler optimizations can increase the utilization of a loop buffer in Lucent DSP16000 on a set of small DSP kernels. Among these transformations were function inlining and the use of the conditional execution support to include loops with simple control diamonds [Uh99]. Sias et al. showed that the internal control flow within loop can be further reduced by applying if-conversion to transform control flow into predicated blocks [Sias01].

The zero-overhead loop support in FITS is similar to [Analog-ADSP21160] in that it is stack-based. Stack-based zero-overhead loop execution can support not only the innermost loop but also nested loops as well. Moreover, since it does not store the actual instructions within loops, there is no need for additional storage, which can result in extra power consumption and area overheads

6.5 Extended and Customized ISA.

Adopting special custom instructions to improve computational efficiency and throughput is a well established area. These special custom instructions can be either an extension to an existing general-purpose ISA or a standalone custom designed application-specific ISA by itself. Examples for the first paradigm are Intel's MMX and SSE [Intel-IA32] or AMD's 3DNow! [Fomithchev00] for common multimedia applications. Examples for the second paradigm are commonly found in ASP designs, such as NVIDIA GeForce [NVIDIA-GeForce] and ATI Radeon [ATI-Radeon] for graphics and gaming, CryptoManiac [Wu01] for cryptography, and [Saini04] for speech synthesis. ISA extension makes the already overly large instruction space of general-purpose ISA even bigger by adding more new instructions. Increased instruction space requires larger opcode fields, which can result in smaller operand size, reduced address modes, or larger instructions that can have detrimental effects on performance and code size. ISA customization requires designing a new datapath for each new application. The non-recurring engineering (NRE) cost and turnaround time for designing a new processor for each application is prohibitively high.

6.6 Dual-width ISA

Dual-width ISA designs, such as Thumb [ARM-Thumb], Thumb2 [ARM-Thumb2], MIPS16 [MIPS-MIPS16], MeP [Toshiba-MeP], ST100 [ST-ST100], and

ARCtangent [ARC-ARCtangent], have been proposed to reduce code size to address the issue of limited memory bandwidth in embedded media computing. In addition to the regular 32-bit instructions, these designs support 16-bit instructions, which specify a subset of the default 32-bit instructions. The idea is to use 16-bit instructions for performance non-critical regions to trade off execution time for smaller memory footprint. For performance critical regions, 32-bit instructions will have to resume execution to make sure important tasks can be completed on time. Ideally, designers want to use 16-bit instructions as much as possible. Because 16-bit instructions alone cannot guarantee the required performance, designers have to keep 32-bit instructions as a fall-back option. Instruction coalescing [Krishnaswamy05] extends the Thumb architecture with the augmenting instructions (AX), which allow the execution of two 16-bit Thumb instructions as a single 32-bit ARM instruction. This avoids some of the performance penalty in replacing 32-bit code with 16-bit code in a dual-width ISA.

Our approach is different in that we believe that 16-bit instructions alone are sufficient to accommodate the requirements of most embedded media applications without the support of large-width instructions. Different applications may not require the same set of instructions. We propose a general-purpose microarchitecture that includes both standard operations enhanced with high-performance data-streaming processing capabilities, but only map a subset of these instructions that a particular program needs to a 16-bit instruction format. We introduce a novel programmable instruction decoder, which can re-map the instruction set definition at run time to accommodate special dynamic execution requirement of a program at any performance critical regions.

Thus, rather than starting with a 32-bit ISA and looking for places to partially substitute it with its 16-bit counterpart, we move straight into the single 16-bit ISA scheme and utilize an instruction encoding and enhanced streamlining computing synthesized to the requirements of each application. We have shown that 16-bit instruction sets are very effective in improving code density as well as reducing power consumption for on-chip caches in our prior work [Cheng04][Cheng05a][Cheng05b]. In this dissertation, we demonstrate that an enhanced general-purpose microarchitecture interfaced with a 16-bit instruction set can achieve significant performance advantages in high-performance embedded applications.

6.7 Reconfigurable Systems

Reconfigurable architecture is a recent trend introduced to improve program encoding efficiency. Reconfigurable processors, like Xtensa [Gonzalez00] and Lx [Faraboschi00], consist of a basic set of instructions that exist in all implementations extended by reconfigurable resources. Designers have the ability to choose from optional functional units, memory interfaces, and peripherals. Customizations are made available through user-defined instructions. The advantage of this approach is that common code sequences may be replaced with one or few user-defined instructions to save code size. However, it is extremely difficult to design a general-purpose Reconfigurable datapath that is well balanced among speed, area, and energy.

The availability of large and cheap field programmable gate array (FPGA) logic promoted reconfigurable computing - a class of architecture alternatives for complex

digital systems. In most reconfigurable architectures, the main microprocessor is coupled with an array of programmable FPGA logic. Depending on the logic block size in the array and the level of coupling between the reconfigurable array and the main processor, a reconfigurable array can be integrated as a standalone coprocessor like Garp [Callahan00] or a function unit for a main processor such as CHIMAERA [Ye00] and Altera Nios II [Altera-NiosII]. Reconfigurable coprocessors perform more tasks independently without the constant supervision of the main processor. On the contrary, reconfigurable function units are more tightly integrated with the main processor. The execution of a reconfigurable function unit occurs on the datapath of the main processor. Moreover, a reconfigurable function unit normally shares the resources, such as register files, on the main processor. Despite promoters argue that FPGAs are flexible to be mapped to any interesting algorithm, the overhead in speed, area, and power associated with a FPGA is often too high to make performance-demanding applications benefit from these FPGA-integrated designs.

DISE [Corliss03] is another work that can dynamically replace dataflow subgraphs in the instruction stream with reconfigurable function units. A special instruction is used to signal the DISE engine, which then sends the appropriate control signals into the pipeline. This model requires a DISE aware operating system and processor, since the DISE subgraphs are specified in the binary at boot time, and must be replaced to execute the modified binary at runtime. Conversely, the FITS platform proposed in this paper does not affect the operating system, nor does it require any special binary translation engine to execute the program.

More recently, Clark et al. proposed configurable compute accelerators (CCA) [Clark04] to speed up the execution of dataflow subgraphs, and an interface [Clark05] that facilitates integrating different CCAs into the baseline processor. Subgraphs are identified and replaced with new microcode instructions at run time. These microcode instructions control the CCA, which is a big array of function units that generally can capture subgraphs of depth from 4 to 7. The microcode is then invoked when a reserved instruction is executed in the program code. The compiler is responsible for what code sequences should be mapped to CCA subgraphs and at what locations in the program these subgraphs should be loaded into a CCA configuration cache. CCA and VIP share some of the same advantages of accelerating program execution by streamlining dataflow computation without expanding the instruction set encoding. The key difference between the CCA approach and VIP is in how the large potential instruction space of chained functions is mapped to the instruction set. CCA reserves a single opcode which specifies the data dependencies to whichever configuration is programmed into the accelerator, while the decoupled instruction set provided in our underlying FITS architecture allow all function permutations to be mapped into the ISA without necessitating additional opcode space. A conventional ISA could not support the wide range of different functions that can be configured since the number of total instructions will quickly grow into the order of thousands just with 2 levels of function units and it will be well past that with 3 or more levels of chained functional units. CCA configures the programmable circuit with the chained functions that the compiler found useful, while the programmable nature of the FITS instruction decoder enables the microarchitecture to implement a fixed circuit capable of executing any of the function permutations; we simply map the one or more

permutations that the application requires to one of the instructions in the FITS ISA. While it may seem that implementing the circuit to perform all permutations would require much greater area than that of a programmable circuit would require, that is not the case. Since each permutation differs in only the control signals going to the multiplexers at each level of the chained function unit design; it is not area limitations that prevent the design of these chained function units (since the area requirement for implementing any one circuit are only slightly smaller than implementing all permutations), but it is the tremendous increase in operations that can be specified in the microarchitecture and the corresponding increase required in the opcode for conventional ISA that is the true limitation. CCA provides one method of avoiding the limitation (configurable circuits), while VIP/FITS provides a more flexible method (programmable instruction decode).

CHAPTER 7

CONCLUSIONS

7.1 Thesis Summary

In this dissertation, we proposed the design and implementation of FITS. FITS is an innovative architectural and microarchitectural framework that can effectively support high-performance embedded computing that requires low-power, low-cost, and rapid time to market. This section summarizes how FITS achieves each of these goals.

7.1.1 High-Performance Solution

FITS improves the performance by integrating proposed Versatile Integrated Processing (VIP) unit and a Zero-Overhead Loop Execution (ZOLE) unit into the microarchitecture. The VIP unit is a universal data-crunching engine that delivers superb data computing and data streaming performances. The area cost of adding new operations using VIP is extremely low: for every additional VIP unit added, the number of additional operations available will increase exponentially. Furthermore, because VIP is

synthesized in standard cells and chaining each extra VIP only costs few multiplexers, we can implement thousands of new specialized operations using less area than it would take to configure a single operation using programmable circuits like FPGAs, and would result in faster circuit speeds. The ZOLE unit streamlines the program control flow by removing expensive loop control overhead from both nested and non-nested loops.

Another aspect of performance improvement comes from the custom synthesized application-specific ISA tailored to the requirements of a given application. The application-specific instruction set tailoring is achieved by replacing the fixed instruction decoder of general-purpose embedded processors with a programmable decoder. The use of a programmable decoder allows designers to add new capabilities to microarchitecture without being restricted by the limited instruction space. The only other constraint of adding new operations is due to chip area, which has been addressed by the space-efficient VIP unit.

The net effect is that the underlying microarchitecture may contain an extremely large set of operations that can never be mapped to any single ISA. Yet, through the use of a programmable instruction decoder, FITS can choose the needed subset of operations being mapping to the premier instruction space for a given application. The instruction selection is determined at compile time. The definition of ISA is loaded to the programmable decoder at boot time. The programmable decoder can be dynamically reconfigured with different set of ISA definitions at run time, if necessary.

One other major advantage of using the programmable decoder is the benefit of decoupling the microarchitectural enhancements from the ISA so that new instructions can be integrated into the underlying microarchitecture, as much as the chip area goal

permits, without being restricted by limited opcode space nor being crippled with bigger instruction decoders. Designers are free to include additional functional capabilities to improve performance, even when those enhancements are useful for only a small percentage of applications since the inclusion of one operation does not require the elimination of another to fit in the instruction set encoding space.

7.1.2 Low-Power Solution

FITS reduces energy consumption by running same applications with much smaller code size and improved locality as a result of half-width ISA. The philosophy of FITS is that high performance and high code density can co-exist if we can match the instruction set to the requirement of a targeted application. FITS improves code density by utilizing instructions that are only 16-bit instead of 32-bit that are commonly used in most conventional machines. Since the instruction width is reduced by half, the total code size can be reduced by half as long as what was originally done in a single 32-bit instruction can also be done in a single 16-bit instruction. To best utilize the half-sized instruction width, the instruction space is allocated to only those operations that are necessary and useful to the given application. We have shown that FITS can achieve a code size reduction that is close to 50% with better performance through application-specific customization.

Half-sized program with better locality means it is possible to replace original instruction caches with those that are only half big and still can yield better cache miss rates. Smaller instruction caches with better hit rates can save both dynamic and static

power consumption. Better cache hit rates also means less traffic from the processor to off-chip memories, which can further improve both performance and power consumption.

7.1.3 Low-Cost and Fast Time to Market Solution

FITS reduces the chip production cost and shortens the design turnaround time through the use of a general-purpose, functionally-rich underlying microarchitecture. Rather than fabricating a new chip to map each new application, we choose a single general-purpose microarchitecture platform augmented with VIP and ZOLE units, so there are an extremely large set (i.e. in the order of thousands) of operations that can be selected to map the requirements of any application of interests. Because of the nature of this single general-purpose microarchitectural platform, FITS can reduce the chip production cost and shortens time to market by leveraging the fabrication advantages of a mass-produced, single-chip solution that amortizes the one-time high NRE cost and lengthy design turnaround time.

Through the use of programmable decoder, general-purpose microarchitecture enhanced with VIP and ZOLE units, FITS provides a new class of low-cost, low-power architecture designs for embedded microprocessors that can achieve application-specific processor performance with fast time to market.

7.2 Future Directions

Future directions of this dissertation research are aimed to broaden the impact of FITS framework by expanding its benefits to both consumers and chip vendors in the IT industry. One aspect of this agenda is to make FITS a reliable, secure, and scalable (RSS) framework.

- Reliable - so FITS can be assured
- Secure - so FITS can be trusted
- Scalable - so FITS can be extended

Making FITS reliable means making it dependable. This will involve in research on fault-tolerant computing, design-for-test and various chip testing techniques, defect detection and prevention, concurrent checking of system operation, run-time failure detection and recovery techniques.

Making FITS secure means making it trustworthy. This will involve in research on implementing cryptographic support in FITS. Cryptographic algorithms are designed so that by observing only the inputs and outputs of the algorithm it is computationally infeasible to break the cipher or to guess the secret key used in encryption and decryption. A well designed cryptographic algorithm does not leak enough useful information during its operation to compromise the security of the system it is trying to protect. However, when a physical implementation of an algorithm is considered, information like the timing characteristics of the circuit implementing the algorithm, power consumption, behavior as a result of internal faults, and timing of the circuit can provide sufficient information to compromise the security of the system.

This type of data is widely accessible since anyone can buy a piece of cryptographic hardware such as smart cards, SIM cards, USB tokens, and perform fault or power analyses on it. Attacks based on the use of this implementation specific information are known as Side Channel Attacks (SCA). Contrary to traditional cryptanalysis attacks, very small amount of side-channel information is enough to completely break a crypto system. One of the goals of this future research direction is to develop methods and designs to make such attacks infeasible.

Making FITS Scalable means two things: (1) FITS can take advantages of continuously shrinking process technology so long as the Moore's Law should live; (2) FITS can be the building block for large multi-processor based designs. To achieve both aspects of scalability, continuous architectural and microarchitectural research advancement must be made.

Part of these future research endeavors is not only developing useful techniques that enable FITS to have these useful RSS properties, but also understanding the underlying principles behind inventing these techniques. We can then apply these understanding to automate the entire design synthesis process. The grand challenge I propose here is to automatically generate an efficient, scalable, reliable, and secure FITS processors in its entirety from a simple goal-directed high level specification.

Another research direction that will broaden the impact of FITS is to empower end-user programmers with the ability to create their own application-specific functional specifications and to automate transferring such specifications into real hardware designs. Oftentimes, it is the end-user programmers who have the application-specific knowledge and the coding-pattern information necessary to implement useful functional synthesis.

My dissertation research has provided a practical foundation framework for extensible processor architectures, at both ISA and microarchitecture levels. As a result, end-users, with little or no knowledge of the backend program analysis, can easily extend existing processor architectures without worrying about breaking them. In this dissertation, I apply this foundation framework to examine how application-specific optimizations can be made practical and useful. In the future, I want to investigate techniques for allowing end-user programmers to develop their own application-specific hardware design languages (HDL) that can be easily incorporated into mainstream HDL, such as Verilog and VHDL.

APPENDICES

APPENDIX A: Instruction Cache Breakdown

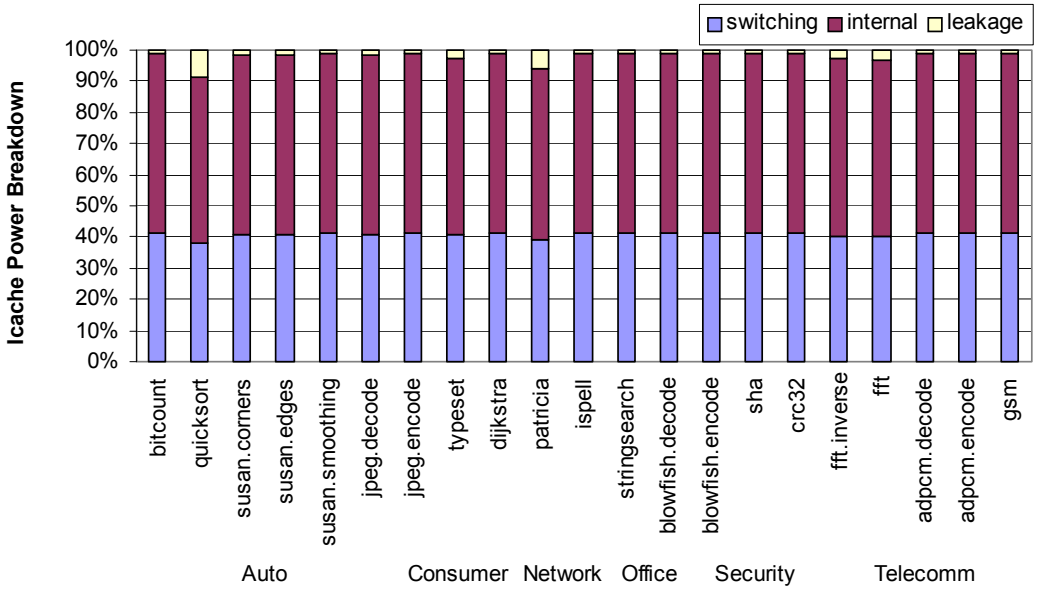

Figure A.1: Instruction Cache Power Breakdown for ARM8

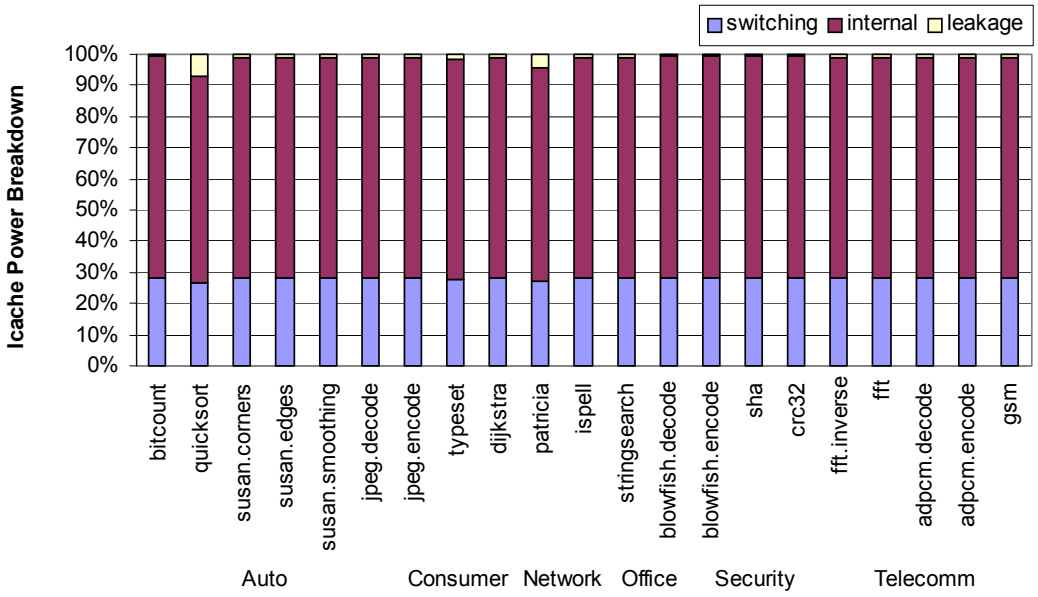

Figure A.2: Instruction Cache Power Breakdown for ARM16

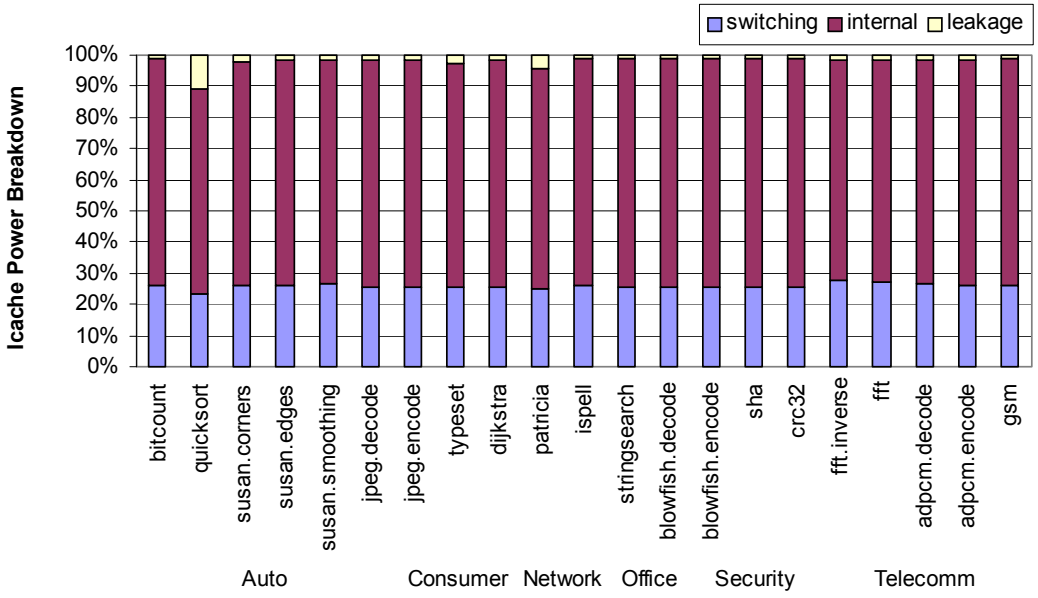

Figure A.3: Instruction Cache Power Breakdown for FITS8

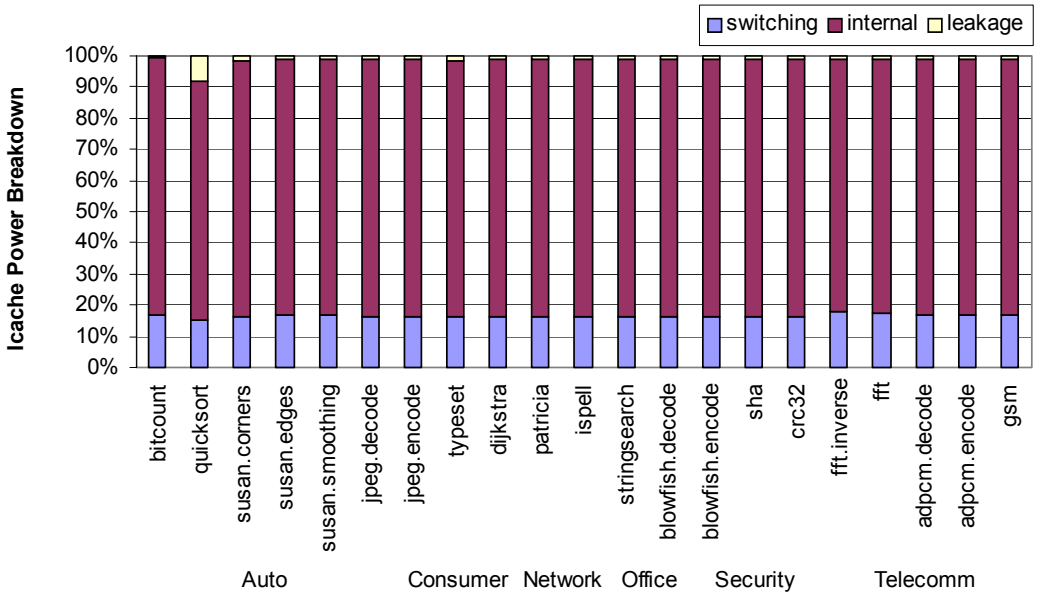

Figure A.4: Instruction Cache Power Breakdown for FITS16

APPENDIX B: Instruction Cache power savings

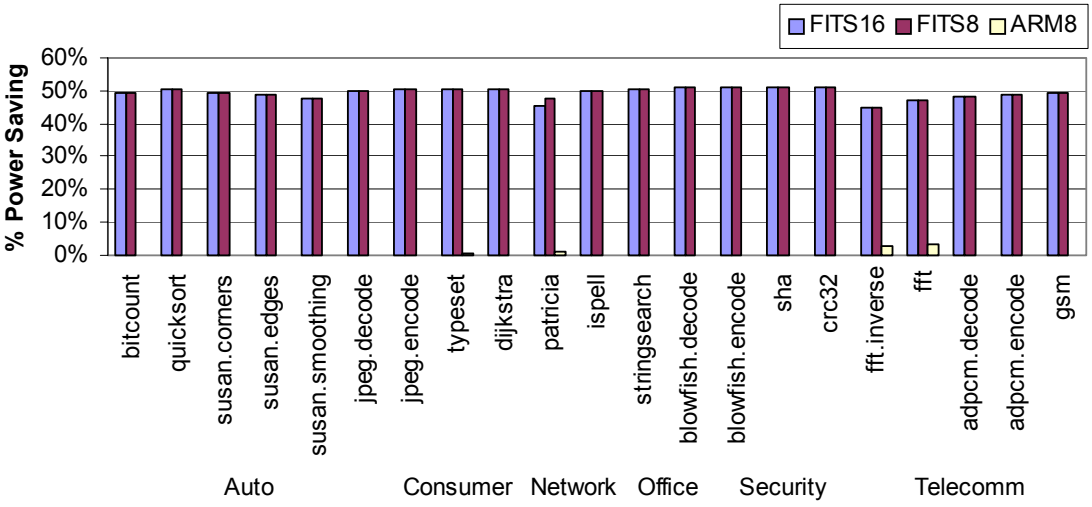

Figure B.1: Instruction Cache Switching Power Saving

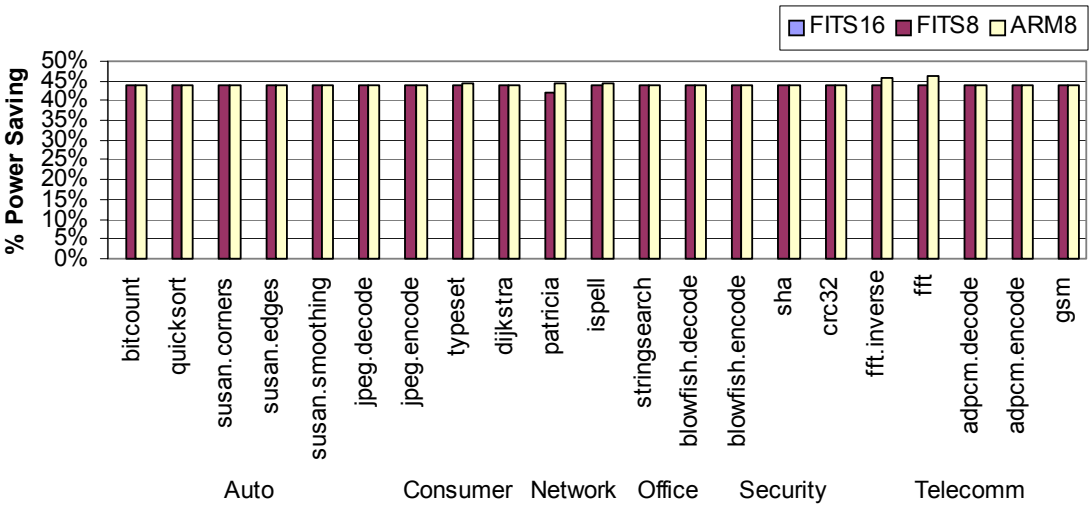

Figure B.2: Instruction Cache Internal Power Saving

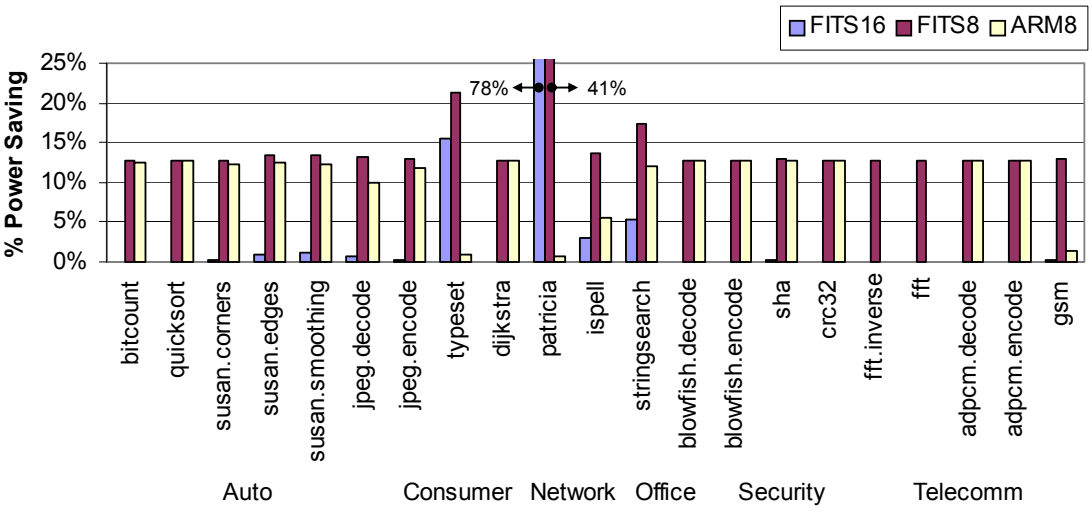

Figure B.3: Instruction Cache Leakage Power Saving

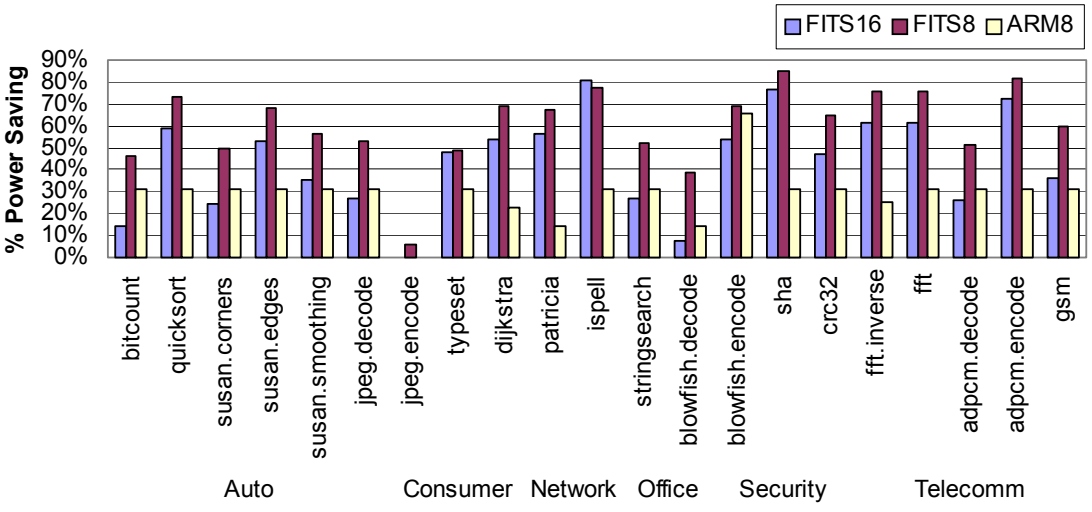

Figure B.4: Instruction Cache Peak Power Saving

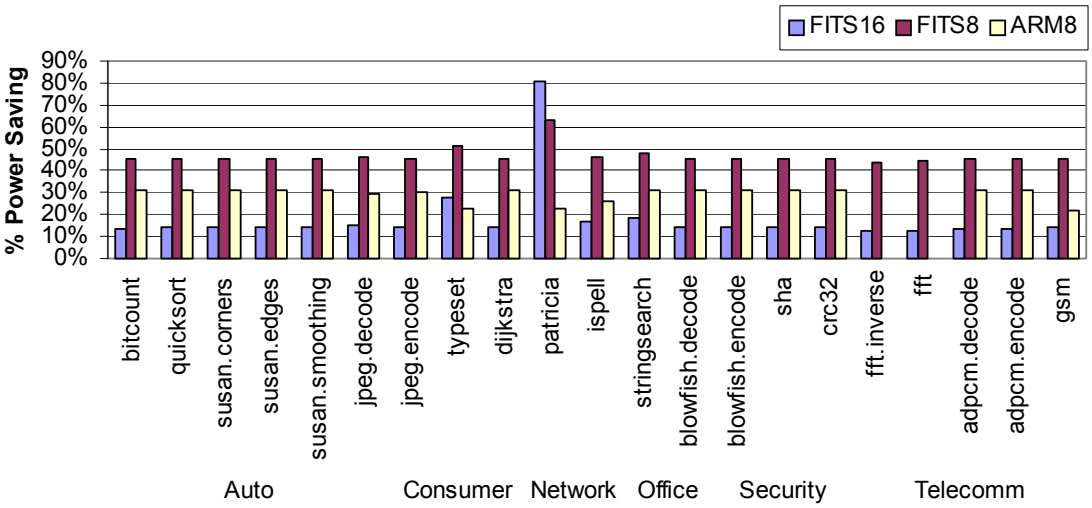

Figure B.5: Instruction Cache Total Power Saving

APPENDIX C: Chip-wide Power Savings

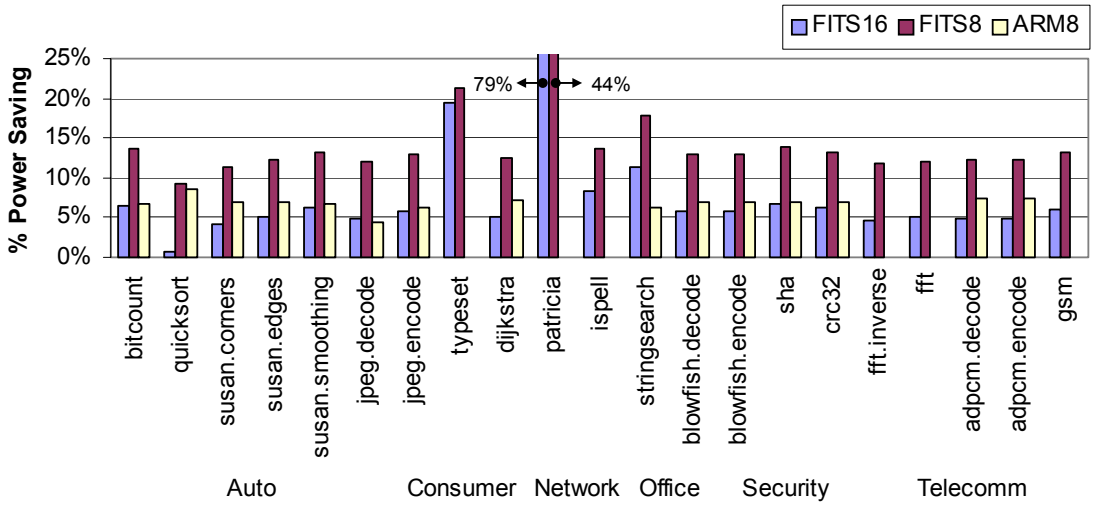

Figure C.1: Chip-wide Switching Power Saving

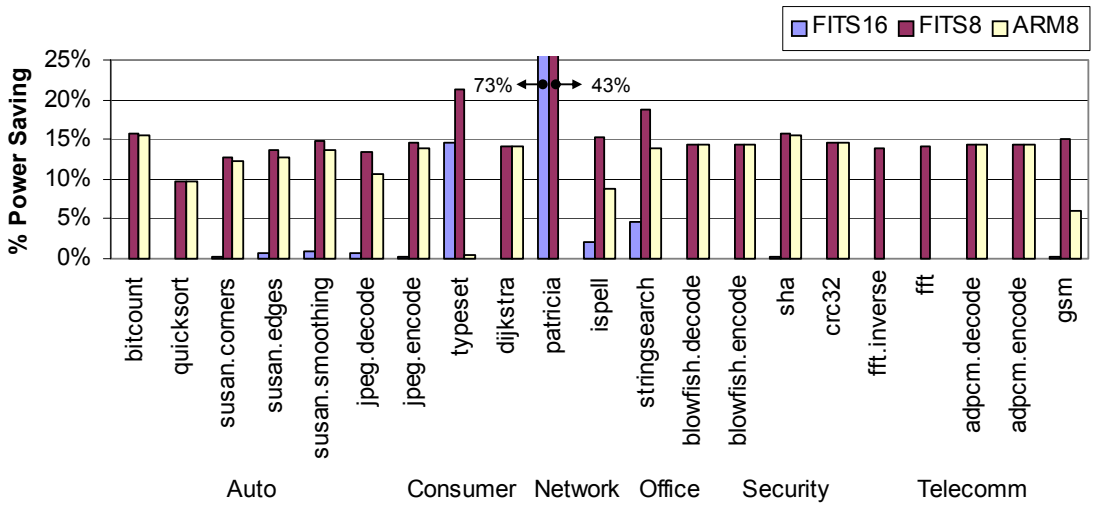

Figure C.2: Chip-wide Internal Power Saving

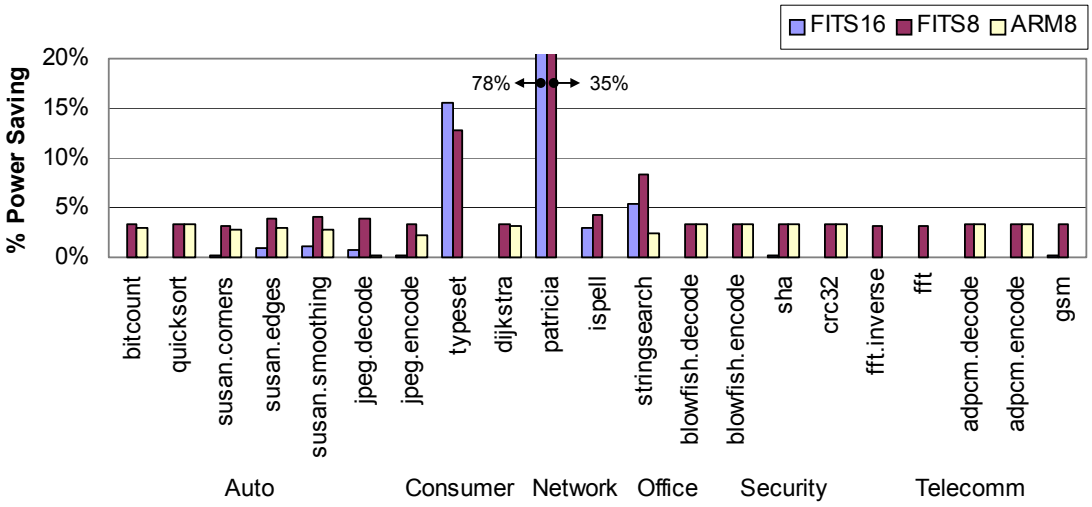

Figure C.3: Chip-wide Leakage Power Saving

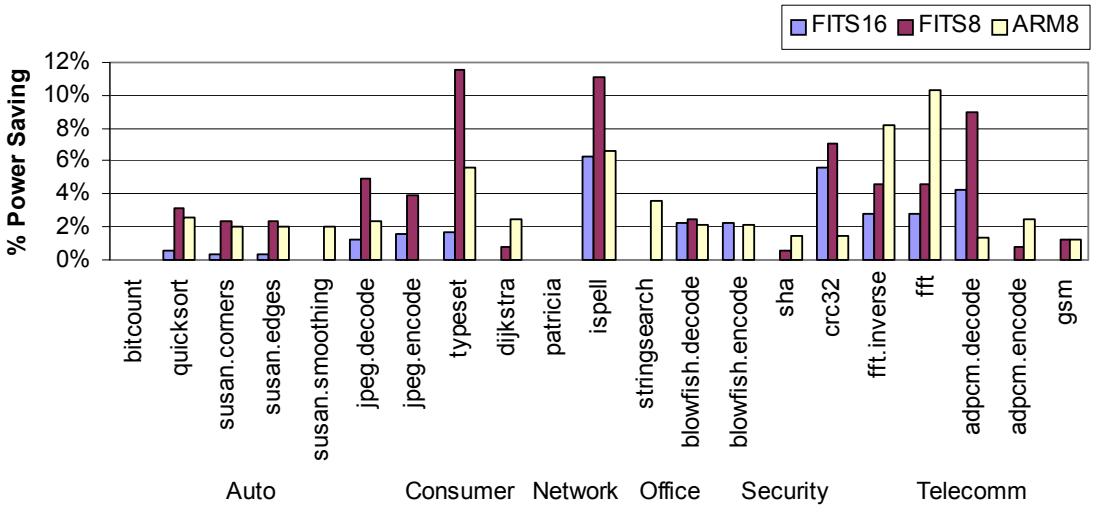

Figure C.4: Chip-wide Peak Power Saving

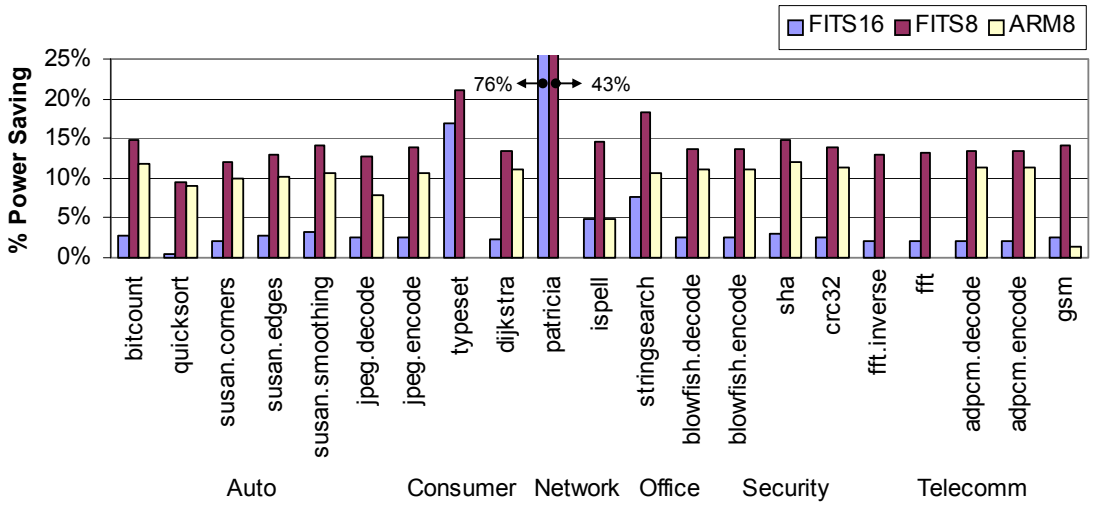

Figure C.5: Chip-wide Total Power Saving

BIBLIOGRAPHY

BIBLIOGRAPHY

[Ahn04] J. Ahn et al., "Evaluating the Imagine Stream Architecture," Proceedings of the International Symposium on Computer Architecture (ISCA), 2004.

[Analog-ADSP21160] Analog Devices Inc, "ADSP-21160 SHARC DSP Instruction Set Reference."

[Allen02] A. Allan et al., "2001 Technology Roadmap for Semiconductors," Computer, Vol. 35, No. 1, January 2002, pp. 42-53.

[Altera-NiosII] Altera Corporation, Nios II Processor Reference Handbook, v 5.0, 2005; see http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[ARC-ARCtangent] ARC International, S. Zammattio, "How to Reduce Time-to-Market for System-on-Chip Design,"2002; see http://www.arc.com/documentation/whitepapers/

[ARM-ARM] ARM Limited, ARM7TDMI (Rev. 4) technical Manual, 2001; see http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf.

[ARM-Thumb] ARM Limited, ARM7TDMI (Rev. 4) technical Manual, 2001; see http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf.

[ARM-Thumb2] ARM Limited, ARM Thumb-2 Core Technology, 2003; see http://www.arm.com/pdfs/Thumb2%20Core%20Technology%20Whitepaper%20-%20Final4.pdf

[ATI-Radeon] ATI Technologies Systems Corp., Radeon X850 Graphics Technology, http://www.ati.com, 2005

[Artisan-MemoryGenerator] Artisan Memory Generator, Artisan Components, Inc., http://www.artisan.com.

[Austin02] T. Austin et al., "SimpleScalar: An Infrastructure for Computer System Modeling," IEEE Computer, Vol. 35, February 2002, pp. 59-67.

[Benini99] L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," International Symposium on Low-Power Electronics and Design (ISLPED), Aug. 1999, pp. 206-211.

[Callahan00] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," IEEE Computer, Volume 33, Issue 4, pp. 62 – 69, 2000.

[Cheng04] A. Cheng, G. Tyson, and T. Mudge, "FITS: Framework-based Instruction-set Tuning Synthesis for Embedded Application Specific Processors," Proceedings of the ACM/IEEE Design Automation Conference (DAC), pp. 920-923, June 2004.

[Cheng05a] A. Cheng, G. Tyson, and T. Mudge, "PowerFITS: Reduce Dynamic and Static I-Cache Power Using Application Specific Instruction Set Synthesis," Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 32-41, March 2005.

[Cheng05b] A. Cheng and G. Tyson, "An Energy Efficient Instruction Set Synthesis Framework for Low Power Embedded System Designs," IEEE Transactions on Computers (TC), Volume 54, Issue 6, pp. 698-713, June 2005.

[Church36] A. Church, "An Unsolvable Problem of Elementary Number Theory," American Journal of Mathematics, 58, 1936, pp 345-363.

[Clark03] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration through Automated Instruction Set Customization," Proceedings of the International Symposium on Microarchitecture (MICRO), December 2003. pp. 129-140.

[Clark04] N. Clark et al., "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," Proceedings of the International Symposium on Microarchitecture (MICRO), 2004.

[Clark05] N. Clark et al., "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 272-283, 2005.

[Compaq-Alpha] Alpha Architecture Handbook, Order Number: EC-QD2KC-TE, Compaq Computer Corp., 1998

[Corliss03] M. Corliss, E. Lewis, and A. Roth, "DISE: A programmable macro engine for customizing applications," Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 362-373, 2003.

[Davidson96] J. W. Davidson, and S. Jinturkar, "Aggressive Loop Unrolling in a Regargetable, Optimizing Compiler," Proceedings of Compiler Construction Conference, pp. 59-73, April 1996.

[Debray00] S. Debray et al., "Compiler Techniques for Code Compaction," ACM Transactions on Programming languages and Systems (TOPLAS), Vol. 22, No. 2, March 2000, pp. 378-415.

[Ditzel87] D. Ditzel, H. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proceedings of the International Symposium on Computer Architecture (ISCA), 1987.

[Faraboschi00] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. "Lx: A technology platform for customizable VLIW embedded processing," Proceedings of the International Symposium on Computer Architecture (ISCA), June 2000, pp. 203-213.

[Fomithchev00] M. Fomithchev, "AMD 3DNow!," Dr. Dobb's Journal, vol. 25, No. 8, pp. 40-42, 2000.

[GCC04] GNU Compiler Collection, http://gcc.gnu.org, 2004.

[Gonzalez00] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," IEEE Micro, Vol. 20, No. 2, Mar.-Apr. 2000, pp. 60-70.

[GPROF] GNU Profiler gprof online manual, http://www.gnu.org/software/binutils/manual/gprof-2.9.1.

[Guthaus01] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," Proceedings of the International Workshop on Workload Characterization, December 2001, pp. 3-14.

[Hines05] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers," Proceedings of the International Symposium on Computer Architecture (ISCA), 2005, pp. 260 – 271.

[Huang00] M. Huang, J. Renau, S. Yoo, J. Torrellas, "A framework for dynamic energy efficiency and temperature management," Proceedings of the Annual International Symposium on Microarchitecture (MICRO), 2000, pp. 202-213.

[IBM-CodePack] IBM Corporation, "CodePack PowerPc Code Compression Utility User's Manual 3.0," 1998.

[IBM-PowerPC] IBM Corporation, "PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors," Software Reference Manual, Pub. G522-0290-01, 2000.

[Intel-IA32] Intel Corporation, "IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, Order Number: 245470," 2001.

[Intel-IA64] Intel Corporation, "Intel IA-64 Architecture Software Developer's Manual, Volume 1: IA-64 Application Architecture," 2001.

[Intel-SA1100] Intel Corporation, "SA-1100 Microprocessor Technical Reference Manual," 2000; see http://www.acm.uiuc.edu/sigarch/resources/docs/sa110_27805802.pdf

[Joseph03] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), 2003, pp. 79-90.

[Kadri03] N.Kadri, S.Niar, and A.R.Baba-Ali, "Impact of Code Compression on the Power Consumption in Embedded Systems," International Conference on Embedded Systems and Applications (ESA), June. 2003, pp. 197-203.

[Kim01] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald, "Challenges For Architectural Level Power Modeling in Power Aware Computing (R. Melhem and R. Graybill eds.)," Kluwer Academic Publishers: Boston, MA, 2001.

[Kim03] N. S. Kim et al., "Leakage Current - Moore's Law Meets Static Power," IEEE Computer, Dec. 2003, pp. 68-75.

[Krishnaswamy05] Krishnaswamy and Gupta, "Dynamic Coalescing for 16-Bit Instructions," ACM Transactions on Embedded Computing Systems (TECS), Vol. 4, No. 1, February 2005, pp. 3–37.

[Kyo05] S. Kyo et al., "An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems," Proceedings of the International Symposium on Computer Architecture (ISCA), pp.134-145, 2005.

[Lau 03] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing code size with echo instructions," Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2003, pp. 84-94.

[Lee97] C. Lee et al., "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," Proceedings of International Symposium on Microarchitecture (MICRO), 1997.

[Lee99] L. H. Lee, W. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops," Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), August 1999.

[Lefurgy00] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing Code Size with Run-time Decompression," Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), Jan. 2000, pp. 218-227.

[Lekatsas00] H. Lekatsas, J. Henkel and W. Wolf, "Code Compression for Low Power Embedded System Design," Proceedings of the Design Automation Conference (DAC), June 2000. pp. 294-299.

[Lucent-DSP16000] Lucent Technologies, DSP16000 Digital Signal Processor Core Instruction Set Manual, 1997.

[Magklis03] Magklis G. et al., “Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor,” Proceedings of the International Symposium on Computer Architecture (ISCA), 2003, pp. 14-27.

[MIPS-MIPS16] MIPS Technologies, “MIPS32 Architecture for Programmers Vol. IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture,” March 2001; see http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary.

[MIPS-MIPS32] MIPS Technologies, “MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set, Document Number: MD00086,” 2003.

[MIRV01] The MIRV Compiler Project, http://www.eecs.umich.edu/mirv, 2001

[Montanaro96] J. Montanaro, et al., “A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor,” IEEE Journal of Solid-State Circuits, Vol. 31, No. 11, Nov. 1996, pp. 1703 – 1714.

[Motorola-MCore] Motorola Inc, “M-Core Reference Manual.”

[Mudge01] T. Mudge, “Power: A First-Class Architectural Design Constraint,” IEEE Computer, Vol. 34, No. 4, Apr. 2001, pp. 52-58.

[NVIDIA-GeForce] NVIDIA Corp., GeForce 7800 GTX GPU, http://www.nvidia.com, 2005.

[Oliver04] J. Oliver et al., “Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor,” Proceedings of the International Symposium on Computer Architecture (ISCA), 2004.

[Orpaz02] A. Orpaz and S. Weiss, "A study of CodePack: optimizing embedded code space," Proceedings of the International Symposium on Hardware/Software Codesign (CODES), 2002, pp.103-108.

[Panalyzer04] SimpleScalar-ARM Power Modeling Project, http://www.eecs.umich.edu/~panalyzer, 2004

[PattersonHennessy03] D. Patterson and J. Hennessy, “Computer Architecture: A Quantitative Approach,” third edition, Morgan Kaufmann Publishers, 2003.

[PattersonHennessy05] D. Patterson and J. Hennessy, “Computer Organization and Design – The Hardware / Software Interface”, 3rd edition, ISBN: 1-55860-604-1, Morgan Kaufmann Publishers, San Francisco, CA, 2005, pp. 348-349.

[Rivers03] J. Rivers, S. Asaad, J. Wellman, and J. Moreno, "Reducing Instruction Fetch Energy with Backwards Branch Control Information and Buffering," Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), August 2003.

[Saini04] R Saini et al., "Design of an application specific instruction set processor for parametric speech synthesis," Proceedings of the International Conference on VLSI Design, 2004.

[Sias01] J. Sias, H. Hunter, and W. Hwu, "Enhancing loop buffering of media and telecommunications applications using low-overhead predication," Proceedings of the Annual International Symposium on Microarchitecture (MICRO), pp. 262-273, December 2001.

[Siemens-TriCore] Siemens Inc, "TriCore Architecture Manual."

[Srinivasan05] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "Exploiting Structural Duplication for Lifetime Reliability Enhancement," Proceedings of the Annual International Symposium on Computer Architecture (ISCA), 2005.

[ST-ST100] STMicroelectronics, ST100 Technical Manual, 2003; see http://www.st.com/stonline/books/pdf/docs/10071.pdf

[ST-ST120] STMicroelectronics, ST120 DSP-MCU Programming Manual, December 2000.

[StarCore-SC140] StarCore DSP Technology, SC140 DSP Core Reference Manual, June 2000.

[TI-TMS320C67] Texas Instruments Inc, "TMS320C67 User's Guide."

[Toshiba-MeP] Media embedded Processor (MeP), Toshiba Corporation; see http://www.mepcore.com/.

[TSMC-18] TSMC 0.18 μm CMOS Process, Taiwan Semiconductor Manufacturing Company (TSMC) Ltd., http://www.tsmc.com.

[TSMC-ROM] TSMC 0.18 μm Process (CL018G) ROM-DIFF-HS Datasheet, Version 2004Q1V1, Artisan Components, Inc.

[TSMC-SRAM] TSMC 0.18 μm Process (CL018G) SRAM-SP-HS Datasheet, Version 2004Q1V1, Artisan Components, Inc.

[Turing36] A. Turing, "On Computable Numbers, with an Application to the Entscheidungs problem," Proceedings of the London Mathematical Society, series 2, 42, 1936, pp 230-265.

[Uh99] G. Uh, et al., “Efficient exploitation of a zero overhead loop buffer,” Proceedings of Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 10-19, May 1999.

[Wilkes53] M. Wilkes and J. Stringer, “Microprogramming an dthe design of the control circuits in an electronic digital computer,” Proceedings of the Cambridge Philosophical Society, Vol. 49, pp. 230-238, April, 1953.

[Wu01] L. Wu, C. Weaver, and T. Austin, “CryptoManiac: A Fast Flexible Architecture for Secure Communication,” Proceedings of the International Symposium on Computer Architecture (ISCA), June 2001, pp. 110-119.

[Xie01] Y. Xie, W. Wolf, and H. Lekatsas, “A Code Decompression Architecture for VLIW processors,” Proceedings of the International Symposium on Microarchitecture (MICRO), 2001, pp. 66-75.

[Ye00] Z. Ye et al., “CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit,” Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 225-235, 2000.