PowerFITS: Reduce Dynamic and Static I-Cache Power Using Application Specific Instruction Set Synthesis

Allen C. Cheng Gary S. Tyson† Trevor N. Mudge

Advanced Computer Architecture Lab
The University of Michigan
Ann Arbor, MI 48109-2122
{accheng,tnm}@eecs.umich.edu

†*Department of Computer Science*
Florida State University
Tallahassee, FL 32306-4530
tyson@cs.fsu.edu

Abstract

Power consumption, performance, area, and cost are critical concerns in designing microprocessors for embedded systems such as portable handheld computing and personal telecommunication devices. In previous work [1], we introduced the concept of framework-based instruction-set tuning synthesis (FITS), which is a new instruction synthesis paradigm that falls between a general-purpose embedded processor and a synthesized application specific processor (ASP). We address these design constraints through FITS by improving the code density. A FITS processor improves code density by tailoring the instruction set to the requirement of a target application to reduce the code size. This is achieved by replacing the fixed instruction and register decoding of general purpose embedded processor with programmable decoders that can achieve ASP performance, low power consumption, and compact chip area with the fabrication advantages of a mass produced single chip solution to amortize the cost. Instruction cache has been recognized as one of the most predominant source of power dissipation in a microprocessor. For instance, in Intel's StrongARM processor, 27% of total chip power loss goes into the instruction cache [2]. In this paper, we demonstrate how FITS can be applied to improve the instruction cache power efficiency. Experimental results show that our synthesized instruction sets result in significant power reduction in the instruction cache compared to ARM instructions. For 21 benchmarks from the MiBench suite [3], our simulation results indicate on average: a 49.4% saving for switching power; a 43.9% saving for internal power; a 14.9% saving for leakage power; a 46.6% saving for total cache power with up to 60.3% saving for peak power.

1. Introduction

Power consumption is now a leading design constraint in microprocessor designs, especially in low-end embedded system market [4]. 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. 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 [2]. This paper presents a novel instruction synthesis technique that could reduce significant instruction cache power loss.

In recent years, embedded systems have received increased attention due to the rapid market growth for high-performance portable devices such as phones, PDAs and digital cameras, MP3 players, mobile personal communicators. These applications require more instruction throughput while retaining strict limits on cost, power dissipation, code size, etc. This necessitates a new system architecture that can exploit the special characteristics of these embedded applications to meet the ever-tighter constraints of time, budgets, and technology.

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 a set of applications that share many common characteristics. Thus, an ASP design contains only those capabilities necessary to execute its target workloads. The result is that ASPs can achieve levels of performance and efficiency that are unattainable in general-purpose processors, as shown in [5][6].

With the wide-spread use of Intellectual Property (IP) cores and the development of configurable processors and tools, customized instruction set synthesis has become feasible to better differentiate products in today's competitive markets [7]. For embedded systems, especially for portable handhelds, performance, power dissipation, chip area, cost, and time to market are often the most important design constraints to be considered. Designers for

0-7803-8965-4/05/$20.00 ©2005 IEEE

contemporary 32-bit systems are struggling to achieve even the minimal satisfactory balance between these factors.

In this paper, we present a cost-effective Framework-based Instruction-set Tuning Synthesis (FITS) technique for designing embedded ASPs. FITS offers a tunable, general-purpose processor solution to meet the code size, performance, and time to market constraints with minimal impact on area. FITS delays instruction set synthesis until after processor fabrication. Post fabrication synthesis is performed by replacing the fixed instruction and register access decoder of conventional designs with programmable decoders. Through the programmable decoders, we can optimize the instruction encoding, address modes, and operand and immediate bit widths to match the requirements of the target application. FITS is cost-effective in that: (1) it reduces the code size by synthesizing 16-bit ISAs with minimal performance degradation for many embedded applications that would normally require 32-bit ISAs; (2) it reduces power consumption by requiring a smaller on-chip cache and by deactivating those parts of the datapath that are not mapped to any instructions of the synthesized architecture; (3) it reduces cost and time to market for new products by utilizing a single processor platform across a wide range of applications, while retaining the ability to optimize the instruction set and register organization for the specific needs of each application. The datapath of a FITS processor would be similar to a general-purpose embedded processor such as ARM, containing a full range of functions, but would map only a subset of those to the synthesized instruction set. 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 be found in a large instruction embedded processor.

The remainder of this paper is organized as follows: Section 2 describes related work in instruction set synthesis for embedded systems. Section 3 presents the FITS design methodology and architectural innovations. Section 4 and 5 explain the power modeling tools used in this study and the simulation environment. Section 6 presents the simulation results and provides detailed analyses for the benefits in power consumption, performance, and code size. We conclude and discuss the future work in Section 7.

2. Related Work

The instruction cache has been recognized as a major source of power consumption in embedded systems. Several techniques that address this issue focus on code compression [8][9]. The main observation is that a subset of the ISA is used for programs so the most commonly executed instructions are compressed to reduce the power dissipated in the memory hierarchy and buses. In [10], Kadri et al. evaluated the effect of code compression on power consumption. They use the dictionary-based software decompression method based on IBM's CodePack technique [11] used in the IBM PowerPC microprocessors [12]. They concluded that both execution time performance and power consumption are very sensitive to the level one instruction cache size. They also observed that the performance and power consumption overhead due to software-based dictionary compression is insignificant.

In the recent years, many embedded processor architectures introduced are manually customizable. The Xtensa [13] is a customizable embedded RISC processor, which consists of a basic set of instructions that exists in all Xtensa implementations plus a set of configurable and extensible options. The designer has the ability to choose from optional functional units, memory interfaces, and peripherals. User-defined instructions are also supported using RTL. Similar configurability and extensibility also exist in embedded VLIW cores as seen in the Lx [14]. The customization of processor to an application domain emerges as a compromise between the fast but inflexible ASIC and the flexible but slow FPGA.

Dual instruction set processors, such as Thumb [15], Thumb-2 [16], MIPS16 [17], ST100 [18], and ARCtangent-A5 [19], have been proposed to reduce power dissipation by improving the code density. These dual instruction set designs address the limited memory and energy constraints by supporting a 16-bit instruction set along with the 32-bit instruction set. The 16-bit instruction provides a subset of the functionality of the 32-bit instruction set to trade off the execution time for smaller memory footprint and better power consumption. Since the 16-bit ISA along cannot give the performance desired, designers need to keep the 32-bit version for the performance reason.

Our approach is different from others in that we believe that a 16-bit ISA can accommodate the requirements of almost all embedded applications without the support of some larger instructions. However, applications may not require the same set of instructions, so we propose an architecture with the full range of functional capabilities found in a 32-bit embedded processor, but only map a subset of instructions that a particular program needs to the 16-bit instruction format. 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 synthesized to the requirements of each application.

3. Synthesis Framework

This section 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 requirement of a target application. FITS improves code density by adopting 16-bit instruction set instead of the conventional 32-bit. Since the instruction width is reduced by half, the total code size can be reduced by half as long as

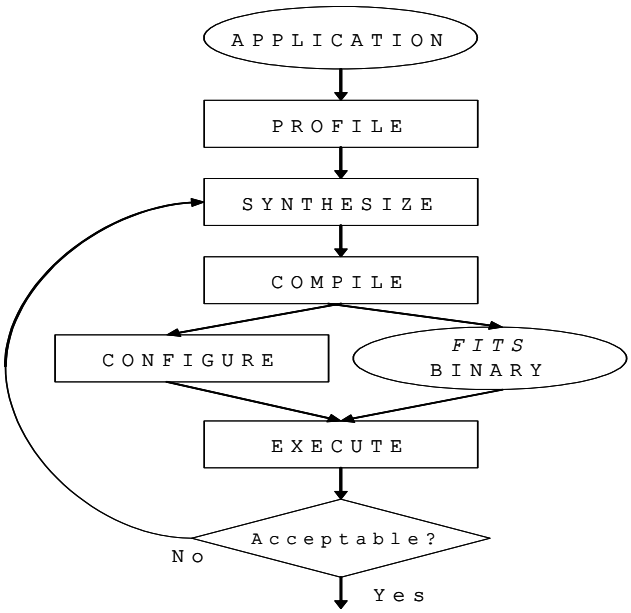


Figure 1: FITS System Design Flow

what was originally done in a single 32-bit instruction can also be done in a single 16-bit instruction. In the later section, 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 the best part of both worlds: a 16-bit dense code that can achieve the 32-bit high performance.

3.1. FITS Methodology

FITS is an application-specific hardware software co-design approach that matches microarchitectural resources to application performance needs, 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, looping instructions, etc.). Limitations on the functions provided are 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, 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 the 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 is consisted of five stages: profile, synthesize, compile, configure, and execute. As illustrated in Figure 1, the target application is first analyzed by the FITS profiler to extract its characteristics. The output of the profile stage is a list of extensive requirement 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 the application. This is the stage where the instruction selection and encoding take place. When the instruction synthesis finishes, the definition of a complete ISA is formed. The FITS compiler would then take the instruction set definition to compile the given application into a 16-bit FITS binary.

When the code generation is completed, the programmable decoder is configured using the instruction decoding and register organization specified by the compiler. Any unused datapaths are turned off at this stage to save power consumption. Once everything completes successfully, we execute the FITS binary on a FITS processor. If all of the requirements are met, a cost-effective solution has been produced. Otherwise, we go back to the synthesize stage and repeat the process again.

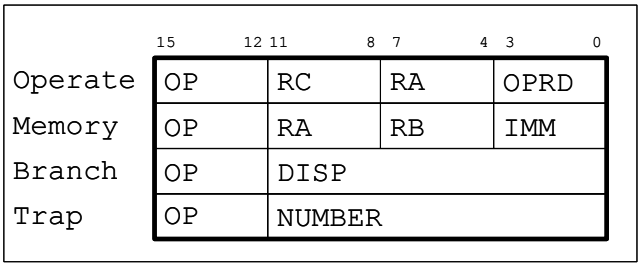


Figure 2: Sample Instruction Formats

3.3. Synthesis Heuristic

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 Set (BIS) and a Supplemental Instruction Set (SIS). A BIS includes instructions found across all applications (e.g. branch, compare, add, etc.); a SIS includes instructions required to make the instruction set Turing-complete [20][21]. 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.

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. The application-specific instruction set (AIS) is determined by evaluating the performance of various 16-bit encoding methods. Register allocation is also designed to trade off the register file size and encoding with register spill frequency.

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 operand 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.

Since the space requirements for different categories of immediates demonstrate distinctive trends, it makes sense to partition the immediate synthesis problem into three subcategories 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 [22] 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.

3.4. FITS 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 target application. For the illustration purpose, Figure 2 included an example instruction formats used by 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 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.

4. 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 [23]. Clearly, power has become one of the most serious design constraints in today's process generations. 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 Section 5.

4.1. Power Metrics

In CMOS logic circuits, the overall power consumption is defined as the sum of dynamic power and static power consumption [24]:

$$P = ACV^2 f + VI_{leak} \quad (1)$$

The first term models the dynamic 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 static power lost due to leakage current, I_{leak}. We ignore the short circuit power caused by the momentary short circuit current at a gate's output whenever the gate switches. The reason is its relatively small contribution to the dynamic power and can thus be absorbed by the dynamic power, if necessary. In addition to dynamic power and static power, peak power is 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.

Dynamic power loss is activity based because it is directly related to the toggling frequency 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. Therefore, these 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. As it will be shown in the later section, this is exactly how FITS optimization achieves power savings for an instruction cache.

4.2. Power Modeling Tool

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 [25] pointed out, cycle simulators intentionally omit considerable implementation detail to speed up simulation speed. The challenge is to select the necessary details that must be put back in to produce accurate power figures.

In this paper, we used a modified version of the "sim-panalyzer" [26] 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 [27]. "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 target 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 compliant with that of an actual fabricated StrongARM® design [2].

5. Experiments

To do an analysis of power consumption and performance evaluation, four different processor configurations were simulated with "sim-panalyzer." A representative subset of the MiBench suite [3] is compiled into the ARM binary using the GCC tool chain [28]. 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 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 was modeled after Intel's SA-1100 StrongARM® embedded microprocessors [29].

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.

6. Results and Analysis

We evaluated the effectiveness of FITS 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 established and clearly seen.

6.1. 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. The insights gained from the embedded workload analysis in [1] inspire us with high potential and plentiful opportunities in FITS. 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 shown in Figure 3 and Figure 4. 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.

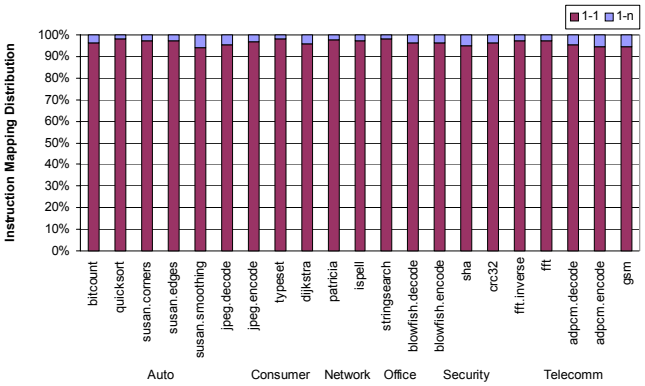


Figure 3: ARM-to-FITS Static Mapping

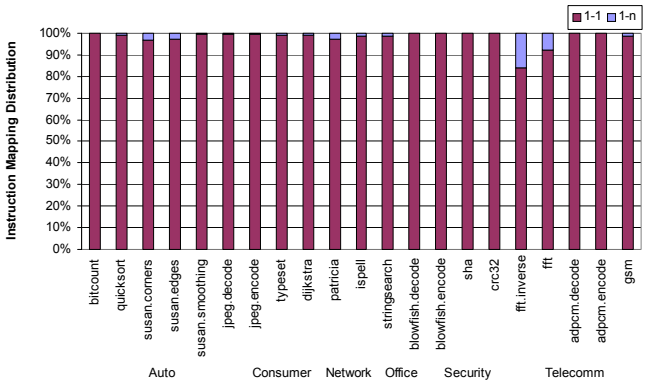


Figure 4: ARM-to-FITS Dynamic Mapping

6.2. Code Size Benefits

Figure 5 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: there was no ARM-THUMB intermixing. Since we do not plan to intermix ARM and FITS together, a better baseline for FITS to compare against should be the pure 16-bit THUMB. 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. 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 instruction fields efficiently.

Like most general-purpose ISAs, THUMB supports a wide range of instructions in order to be able to specify lots

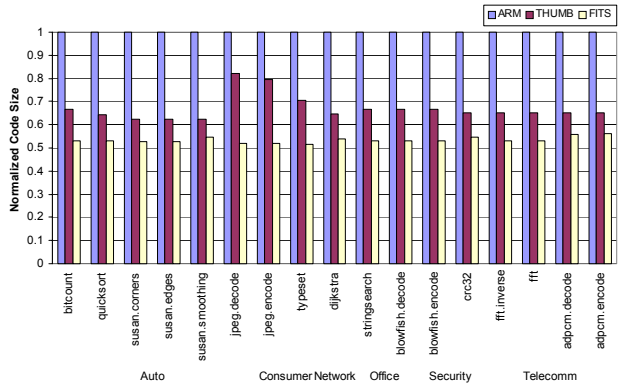


Figure 5: Code Size Footprint

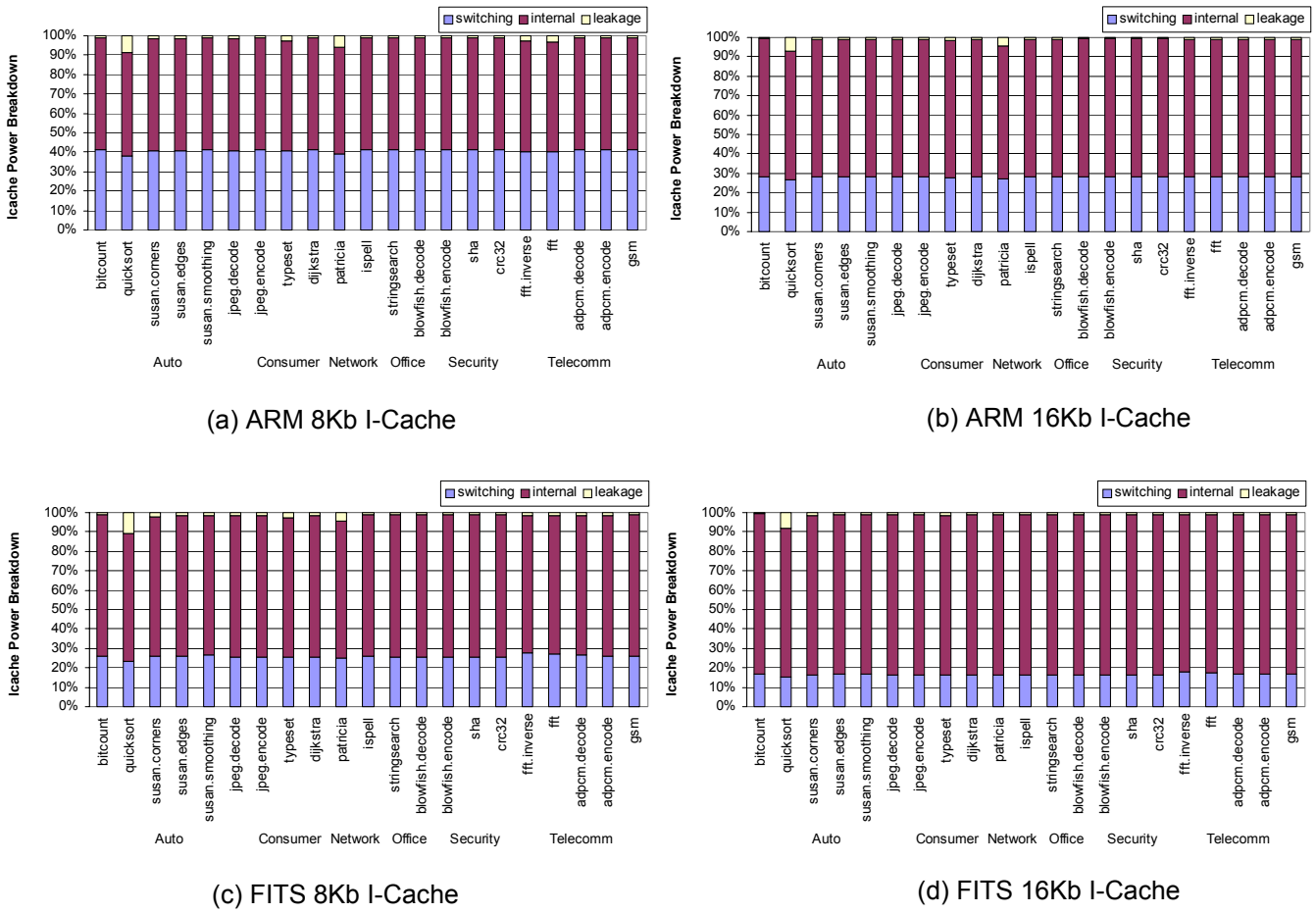
of applications. However, this general-purpose capability requires more opcode space and makes the other instruction fields, such as register and immediate operands, smaller. When the register operand width is reduced, the processor can specify less architect registers and thus increasing the register pressure. Higher register pressure causes more spillings and thus increasing the number of memory references in the program. This is a 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.

6.3. Power Dissipation Benefits

The best way to reduce overall chip power dissipation is to attack each of the microarchitectural components using power. In this paper, 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 section 5, 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



(a) ARM 8Kb I-Cache

(b) ARM 16Kb I-Cache

(c) FITS 8Kb I-Cache

(d) FITS 16Kb I-Cache

Figure 6: I-Cache Power Breakdown

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 four processors run at a fixed 200 MHz operating frequency and the differences among their simulation times were not significant. Since energy is the product of power and time, with negligible difference in the time component, the ratio of energy saving would roughly have identical distribution as the ratio of power saving.

6.3.1. Instruction Cache Power Breakdown

From the instruction cache power breakdown, 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,

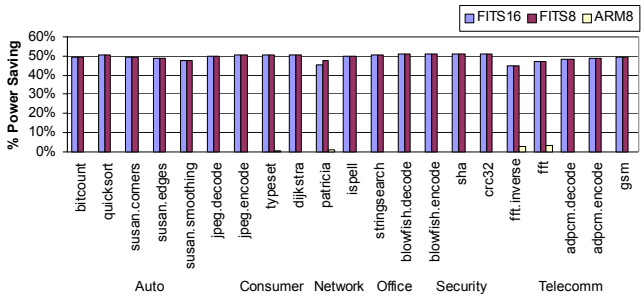


Figure 7: I-Cache Switching Power Saving

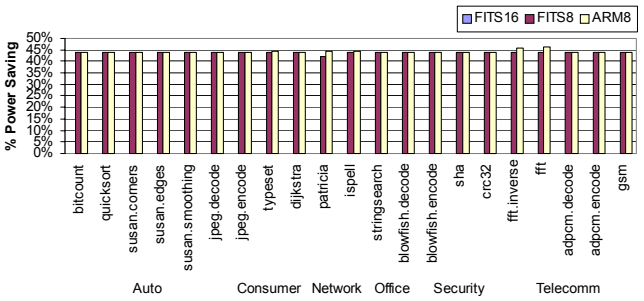


Figure 8: I-Cache Internal Power Saving

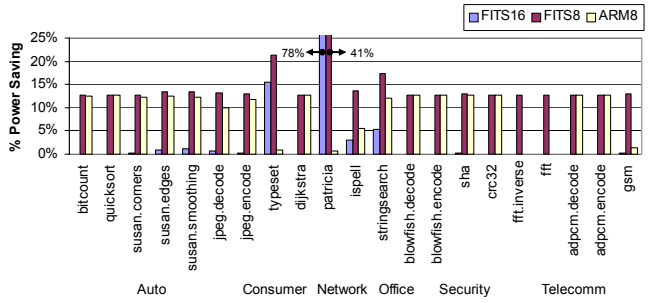


Figure 9: I-Cache Leakage Power Saving

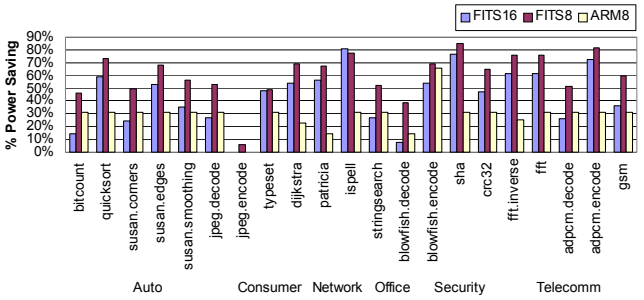


Figure 10: I-Cache Peak Power Saving

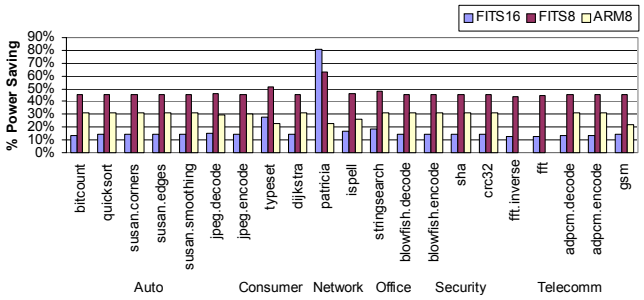


Figure 11: Total I-Cache Power Saving

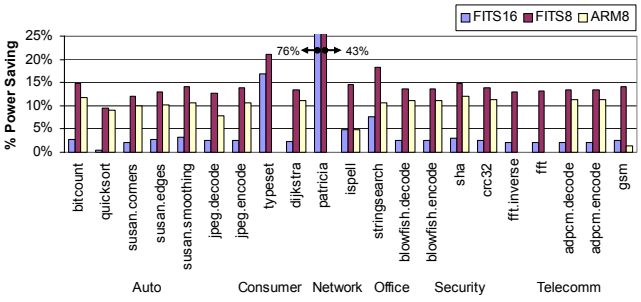


Figure 12: Total Chip Power Saving

which means less gate switches and the switching power is reduced.

Third, with the 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 same sized caches have same number of gates. 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 Figure 6(a) to Figure 6(b) and Figure 6(a) to Figure 6(c), 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 Figure 6(b), 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.

6.3.2. 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 above. 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.

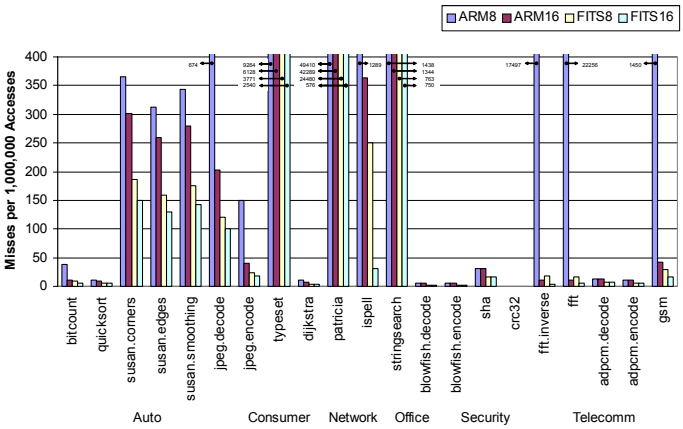


Figure 13: Instruction Cache Miss Rate

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. As shown in Figure 7, 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 in Figure 8 and Figure 9, 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. From Figure 10, 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 cache power consumption results which combine all the component savings above. As shown in Figure 11, FITS8 gives the highest

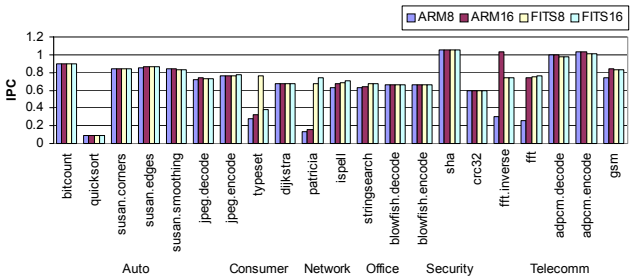


Figure 14: Instructions Per Cycle (IPC)

47% average total cache power saving followed by ARM8 and FITS16 with each saves 27% and 18% respectively. Figure 12 shows how these instruction cache power savings would be translated into the total chip power savings: on average, 15% total chip power saving for FITS8; 8% for ARM8; 7% for FITS16.

6.4. 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). 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 section 7.3, 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.

6.4.1. Cache Miss Rate

Figure 13 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 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.

6.4.2. Instruction Per Cycle (IPC) Rate

Figure 14 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 those 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. This is a result of high instruction mapping rates and low cache miss rates.

7. Conclusions

The goal of this research is to argue for a new approach to the design of a class of embedded processors that reduces power and code size, while maintaining satisfactory performance. We feel that by delaying the mapping of instruction set to microarchitecture to a point after chip fabrication, it will be possible to match the dense coding capabilities of ASP while retaining the fabrication advantages of a single chip design. Using the FITS design methodology enables a cost-effective 16-bit ISA synthesis solution while reducing design time and complexity, by decoupling the microarchitectural enhancements available on chip from the encoding issues of mapping to the subset of instructions required by a single application. Our analysis shows that for a wide range of embedded applications it is feasible to utilize a 16-bit instruction format, but that each application may require a different selection of operations and storage components. By delaying instruction assignment and register file organization until a program is loaded, it is possible to aggressively design the microarchitecture, including operations that are only occasionally useful, without the code bloat that would occur on a conventional machine.

8. References

[1] A. Cheng, G. Tyson, and T. Mudge, "FITS: Framework-based Instruction-set Tuning Synthesis for Embedded Application Specific Processors," Design Automation Conference (DAC), Jun. 2004.

[2] J. Montanaro, et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," IEEE Journal of Solid-State Circuits, Vol. 31, Nov. 1996.

[3] M. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," Proc. IEEE 4th Annual Workshop on Workload Characterization, Dec. 2001.

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

[5] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," 28th Int. Symposium on Computer Architecture (ISCA), June 2001.

[6] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," Int. Symp. on Microarchitecture (MICRO), Dec. 2003.

[7] R. Priebe and C. Ussery, "A Configurable Platform for Advanced System-On-Chip Applications," ICSPAT 2000, Dallas, TX, October 2000.

[8] L. Benini et al., "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," Int. Symp. on Low-Power Electronics and Design (ISLPED), Aug. 1999.

[9] H. Lekatsas, J. Henkel and W. Wolf, "Code Compression for Low Power Embedded System Design," In Proceedings of the 37th Design Automation Conference (DAC), June 2000.

[10] N.Kadri, S.Niar, and A.R.Baba-Ali, "Impact of Code Compression on the Power Consumption in Embedded Systems," Int. Conf. on Embedded Systems and Applications, Jun. 2003

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

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

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

[14] P. Faraboschi et al. "Lx: A technology platform for customizable VLIW embedded processing," Int. Symp. on Computer Architecture (ISCA), pages 203--213, June 2000.

[15] ARM7TDMI technical Manual. ARM Ltd., http://www.arm.com

[16] ARM Thumb®-2 Core Technology, ARM Ltd., http://www.arm.com/armtech/Thumb-2.

[17] K. D. Kissell, "MIPS16: High-density MIPS for the Embedded Market," in Proc. of Real Time Systems (RTS), 1997.

[18] ST100 Technical Manual, STMicroelectronics, http://www.st.com.

[19] ARCtangent-A5 microprocessor Technical Manual, ARC Cores, http://www.arccores.com.

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

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

[22] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing Code Size with Run-time Decompression," Int. Symp. on High-Performance Computer Architecture (HPCA), Jan. 2000.

[23] A. Allan et al., "2001 Technology Roadmap for Semiconductors," Computer, Vol. 35, Issue 1, pp. 42-53, Jan. 2002.

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

[25] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald, "Challenges For Architectural Level Power Modeling in Power Aware Computing," Kluwer Academic Publishers, 2001.

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

[27] T. Austin et al., "SimpleScalar: An Infrastructure for Computer System Modeling," IEEE Computer, Vol. 35, Feb. 2002.

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

[29] Intel Corporation, "SA-110 Microprocessor Technical Reference Manual," ftp://download.intel.com.