High-quality ISA synthesis for low-power cache designs in embedded microprocessors

A. C. Cheng
G. S. Tyson

Energy efficiency, performance, area, and cost are critical concerns in designing microprocessors for embedded systems, such as portable handheld computing and personal telecommunication devices. This work introduces framework-based instruction set architecture (ISA) synthesis, which reduces code size and energy consumption by tailoring the instruction set to the requirement of a targeted application. This is achieved by replacing the fixed instruction and register decoding of general-purpose embedded processors with programmable decoders that can achieve application-specific processor performance, low energy consumption, and smaller code size while maintaining the fabrication advantages of a mass-produced single-chip solution. Experimental results show that our synthesized instruction set results in significant power reduction in the L1 instruction cache compared with ARM® instructions.

Introduction

Power consumption is a leading design constraint in microprocessor designs, especially in the low-end embedded systems market [1]. In addition to requiring costly heat removal, excessive power consumption in embedded devices reduces battery life. Since battery power density is increasing at a rate of only approximately 5% per year, extending battery lifetime must come from improvements in the energy efficiency of system components. Memory structures are by far the most predominant source of power dissipation. For instance, in the Intel StrongARM** processor, caches consume more than 40% of total chip power, with 27% being devoted to the instruction cache (I-cache) [2]. We address this issue by presenting a novel instruction synthesis technique that can reduce significant I-cache power loss.

Embedded system applications, such as cell phones, personal digital assistants, digital cameras, MP3 players, and mobile personal communicators, require growing instruction throughput within limits of cost, power dissipation, and code size. One approach is to move away from general-purpose processors to application-specific processors (ASPs) designed to provide only those capabilities necessary to execute the targeted workload, thereby achieving higher levels of performance and efficiency than are attainable with general-purpose processors [3, 4].

In this paper, we present the framework-based instruction-set tuning synthesis (FITS) technique, which enables designers to use a tunable, general-purpose processor solution to meet code size, energy, and time-to-market constraints with minimal impact on area. FITS delays instruction-set synthesis until after processor fabrication. With a fixed microarchitecture, synthesis is performed by replacing the fixed instruction and register access decoder with programmable decoders through which we can optimize the instruction encoding, address modes, operand, and immediate bit widths to match the requirements of a targeted application. FITS is cost-effective in three ways: It reduces the code size by synthesizing 16-bit-length instructions with minimal performance degradation for a full range of embedded applications that would normally require 32-bit instruction set architectures (ISAs); it reduces energy consumption by deactivating those parts of the datapath not mapped to any instructions of the synthesized architecture; and 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

0018-8646/06/$5.00 © 2006 IBM

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 that of a general-purpose embedded processor such as ARM, containing numerous functions but mapping only a subset of those to the synthesized instruction set. This makes it possible to encode all instructions in a short, 16-bit format while retaining all of the special-purpose operations found in high-end 32-bit embedded processors.

The contributions of our work are threefold. First, we provide thorough analyses of the characteristics of embedded applications showing that a 16-bit instruction set is sufficient for these high-performance embedded applications. Second, we present a novel cost-effective 16-bit instruction-set synthesis framework optimized for modern reduced instruction set computing (RISC) pipelined architectures. Third, we demonstrate the effectiveness of this framework of leverage compact, energy-efficient, high-performance designs comparable to an ASP with substantially less engineering cost.

Related work

Kadri et al. [5] observed that energy consumption and program execution time are very sensitive to the L1 I-cache size. One way to address this issue is to compress the code, which can decrease the number of cache misses because a smaller footprint of instructions is being accessed. The IBM CodePack technique [6], included in its PowerPC* processors [7], used Huffman tables to compress cache blocks. Xie et al. [8] proposed a code-compression algorithm based on arithmetic coding in combination with a precalculated Markov model. Because these code-compression schemes compress all of the instructions in the program, the decompression overhead occurs at every instruction fetch. Benini et al. [9] and Lekatsas et al. [10] propose a dictionary-based code-compression algorithm to compress only instructions that appear frequently. These code-compression approaches have the disadvantage of complicating instruction fetch and decode logic, because instructions can differ in size.

Instruction reuse is another popular approach to reduce code size. Procedural abstraction [11] is a compiler optimization that 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 [12], which indicate 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. An 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 usually slows down program execution. Spatial locality may also be reduced, which may decrease cache performance.

Configurable architecture is a recent trend to improve program encoding efficiency. Configurable processors, such as the Tensilica Xtensa** [13] and Lx [14], consist of a basic instruction set that exists in all implementations extended by configurable resources. Designers have the ability to choose from optional functional units, memory interfaces, and peripherals. Customizations are made through user-defined instructions. The advantage is that common code sequences may be replaced with one or a few user-defined instructions to save code size. However, it is extremely difficult to design a general-purpose configurable datapath that is well balanced with respect to speed, area, and energy.

Dual-instruction-set processors—such as the ARM Thumb** [15] and Thumb-2 [16], MIPS Technologies MIPS16 [17], STMicroelectronics ST100** [18], and the ARC International ARCtangent** [19]—have been proposed to improve memory use and power dissipation by improving the code density. They support 16-bit and 32-bit instruction sets. The 16-bit instruction set provides a subset of the 32-bit instruction set functionality to trade off the execution time for smaller memory footprint and better energy consumption. Some of the functionality of the native 32-bit instructions must be abandoned to obtain a more compact 16-bit encoding version. Nevertheless, the 16-bit instructions alone cannot give the performance desired, so the 32-bit instructions are still needed. Instruction coalescing [20] extends the Thumb architecture with augmenting instructions that 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.

More recently, Hines et al. [21] proposed an instruction-packing technique to reduce code size. Instruction packing removes instruction-fetch cost by placing frequently occurring instructions in special registers. The advantage is that code size is reduced without the use of large dictionaries. The difference between this technique and FITS is at instruction decode; FITS uses a programmable instruction decoder to achieve application-specific customization by allowing a subset of instructions implemented in the microarchitecture to be mapped to the ISA for each different application. The advantage is that instructions are half-sized (16 bits 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 the pipeline.

FITS framework

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 adopting a 16-bit instruction set instead of the conventional 32-bit one. Because 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 a later section, we show that FITS can achieve a code size reduction close to 50%. 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.

Methodology

A FITS processor consists of a fairly large set of functional units, including standard arithmetic logic unit (ALU) operations and other useful instructions (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 be useful to only 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 microarchitectures with which it can work—from general-purpose digital signal processors and embedded processors, such as ARM, to application-specific customized datapaths. FITS provides the same level of customization as many ASPs, trading somewhat greater chip area requirements for elimination of 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 nonvolatile state in a FITS processor. At this point, the processor instruction set and register file organization is complete. If the application is later upgraded with increased functionality, FITS can reconfigure the decoders to match the new requirements. In general, FITS can transform any general-purpose machine into an application-specific processor platform with overprovisioned resources that can be dynamically configured to adapt to the needs of different applications.

System design flow

The system design flow of FITS consists of five stages: profile, synthesize, compile, configure, and execute. 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 analyses related to each element that makes up an instruction set, such as opcode field, operand field, immediate field, and register pressure.

FITS then uses this information as a guideline to synthesize an appropriate instruction set. Instruction selection and encoding take place at this stage. Instructions are selected on the basis of their referenced frequencies. When the instruction synthesis finishes, the definition of a complete ISA is formed. The FITS compiler then uses the instruction-set definition to compile the given application into a 16-bit FITS binary. Any unused portions of the datapath are turned off to reduce power consumption [22]. Until this point, when the instruction synthesis is completed, everything is performed offline. During 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 minimal. More details on the initialization are given in the programmable decoder section. Once everything completes successfully, the compact FITS code is executed without performance degradation.

Instruction synthesis

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). A SIS includes instructions required to make the instruction set Turing-complete [23, 24]. 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 the purpose of clarity, they are separated into two different instruction sets, and we include both of them in all applications.

FITS also includes an application-specific instruction set (AIS), taken from the set of functional units in the

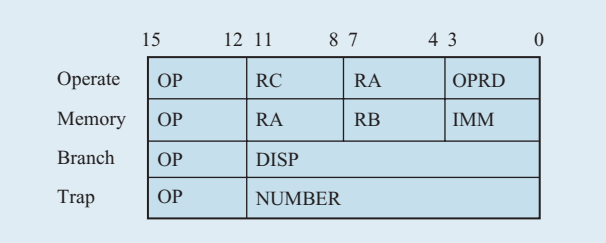

Figure 1

Example of a FITS instruction format.

microarchitecture that are necessary for the application to meet its performance goals. The 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 (e.g., add) when the instruction can be used almost all of the time 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 for an extra opcode bit to indicate a mode switch.

The space requirements for different categories of immediate operands demonstrate distinctive trends; thus, it makes sense to partition the immediate synthesis problem into three subcategories and perform a category-based synthesis accordingly. FITS adopts a utilization-based technique to encode the immediate-operand space. It identifies the most frequently accessed immediate operands and places them in programmable, nonvolatile memory storage, replacing the immediate operand of the instruction with an index into the immediate storage. This is similar to the dictionary compression method in [25], except that FITS can dynamically reconfigure the total immediate field width and adjust widths of other instruction fields accordingly to best reflect the application requirements, and FITS targets only the immediate fields rather than a whole instruction.

Instruction formats

FITS instructions are all 16 bits in various different instruction formats specifying zero, one, two, or three 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 application. **Figure 1** is an example instruction format used for the CRC32 program from the MiBench telecommunication benchmark group.

Operate instructions such as arithmetic, compare, and logical are used for data processing. They use a source register RA and a source operand OPRD and write the 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 program counter. Subroutine calls place the return address in the register specified by the first four bits of the DISP field. The trap instructions perform interrupts, exceptions, task switching, and other complex operations that must be done atomically.

FITS programmable decoder

In almost all modern processors, datapath control lines for an instruction are hardwired and regulate the behavior (e.g., reads or writes) of different parts of the processor datapath, such as changes to the program counter, register files, ALU selection, memories, and other processor states. A nonvolatile read-only memory (ROM) is usually used to store control lines associated with each instruction. This conventional instruction decoder scheme is illustrated in **Figure 2(a)**. 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.

In contrast to this conventional hardwired fixed decoding scheme, the instruction decoding of a FITS processor is programmable [**Figure 2(b)**]. The FITS instruction decoder consists of a standard n-to-2^n binary row decoder and a 2^n-entry SRAM, where n is the opcode width and 2^n is the number of instructions specified by the ISA. (Although DRAM has a higher density that may use only one transistor per bit, a periodic refresh operation is required to keep its memory contents from disappearing. Thus, SRAM was chosen for its better performance.) For all of the embedded applications we studied, 4-bit-wide opcode was large enough to meet the execution requirements, which makes the number of SRAM entries equal to 16. The 16-entry SRAM is used to store the instruction control information, which ordinarily would be stored in the ROM of a conventional decoder. Since the width of an SRAM entry is the same as that of a

ROM entry, each SRAM entry is wide enough to store all control line signals for one instruction. When the 4-bit opcode input is inserted, one of 16 outputs is activated, and the corresponding instruction control signals are fetched and sent down the pipeline to set the datapath accordingly.

By making the instruction decoder programmable, designers can freely select a subset of predefined microarchitecture functions that are best suited to the targeted application. These are then mapped to the ISA of a FITS processor by loading their corresponding instruction control signals into the programmable decoder. Specifically, a custom instruction set is defined and synthesized by the compiler. At start-up, the programmable decoder is initialized with the synthesized instruction set. We introduced a MAP instruction and included it in all synthesized ISA to perform this decoder initialization. The MAP instruction updates the FITS decoder with the instruction control information stored in the original ROM decoder. The decoder initialization overhead is small. The reading of the ROM decoder and writing to the FITS decoder can be done in one cycle. With a 4-bit opcode that specifies up to 16 instructions, we pay a one-time cost of 16 cycles of start-up to load the FITS programmable decoder. The ROM decoder is accessed only during chip initialization, after which there is no need to access it again, and it is turned off. After initialization, all instruction decoding is handled directly by the FITS decoder. Thus, the number of read and write accesses to the FITS decoder for instruction decoding is the same as that to a conventional ROM decoder.

Experimental setup

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 ARM Artisan** Memory Generator under worst-case process conditions. The technology used is the Taiwan Semiconductor Manufacturing Company (TSMC) six-layer metal 0.18-μm CMOS process. Benchmark programs from all six categories of the MiBench embedded test suite [26] are compiled into the ARM binary using the GNU compiler collection tool chain. We ran full simulation on all 21 compatible programs to their completion without skipping any instructions.

Four different processor configurations were simulated with Sim-Panalyzer [27]. To clearly demonstrate the effectiveness of FITS in reducing I-cache power dissipation, we restrict the experiment to allow only a single controlled variable: I-cache size. There are two different I-cache sizes: 16 Kb or 8 Kb (16 Kb is the default cache size in the SA-1100 core). For simplicity, simulations of the original ARM code with a 16-Kb and an 8-Kb I-cache are abbreviated as ARM16 and ARM8, respectively; similarly, simulations of the FITS-optimized code with a 16-Kb and an 8-Kb I-cache are abbreviated as FITS16 and FITS8, respectively. The rest of the microarchitecture remained the same and was modeled after that of the Intel SA-1100 StrongARM embedded microprocessor [28].

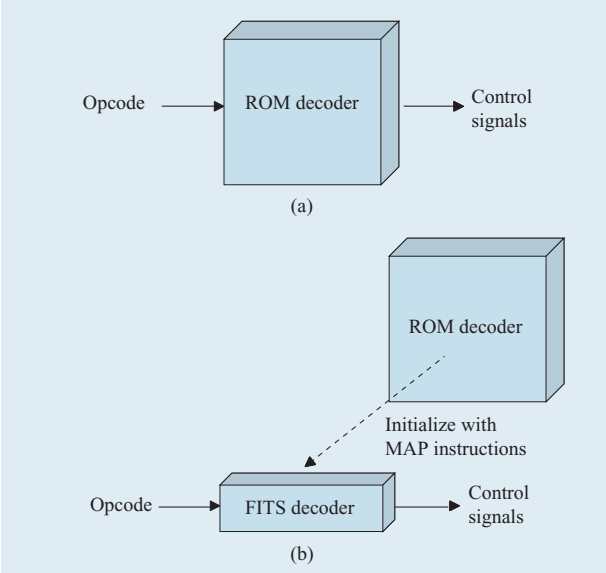

Figure 2

(a) Conventional fixed instruction decoder; (b) FITS programmable instruction decoder.

Results and analysis

In this section, we first present a comprehensive evaluation of the costs and benefits of the FITS programmable decoder in area, access latency, and power consumption. Following the decoder analysis is a discussion of the effectiveness of the FITS framework at the application level using the following metrics: instruction mapping rate, code size saving, power reduction, and performance measurement.

FITS programmable decoder evaluation

To understand the cost of incorporating programmable instruction decoding into a FITS processor, we compared the area, access latency, and power consumption of fixed and programmable decoders. To guarantee that the worst-case performance was satisfactory, all data points presented were taken from the slow process corner, which assumes the maximum propagation delay, lowest operating voltage, and highest junction temperature. Three lines were plotted: ROM represents the data points of a regular fixed instruction decoder; FITS represents the

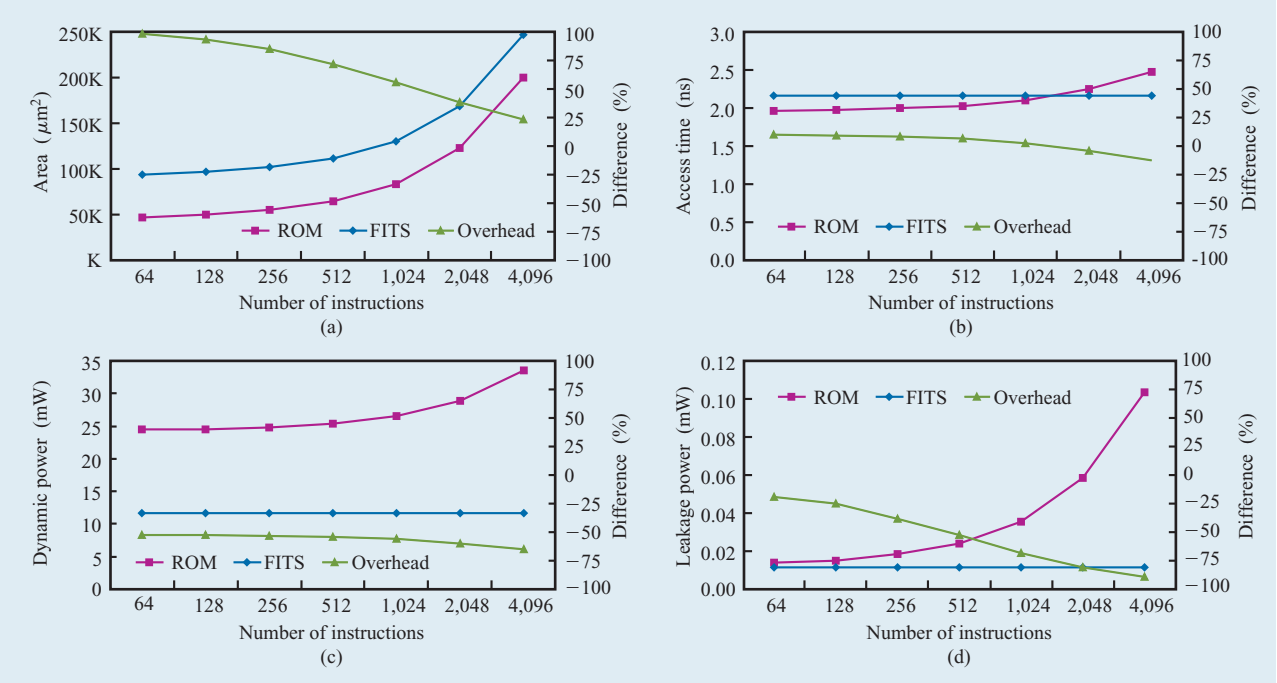

Figure 3

Comparisons between fixed and programmable decoders: (a) footprint area and (b) access time; (c) dynamic and (d) leakage power consumption.

data points of the FITS programmable instruction decoder; and overhead represents the overhead associated with the FITS programmable decoder. We express overhead in percentage difference and compute it using the formula

$$overhead = \frac{FITS - ROM}{ROM} \times 100\%.$$

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

Footprint area analysis

Figure 3(a) shows the footprint area, in square micrometers (μm^2), 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 16×32 SRAM is less than 47K μm^2 in the TSMC 0.18-μm process. This additional area is very small compared with the total chip area, which generally ranges from tens to hundreds of square millimeters (mm^2) 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 at 98% for 64 instructions and drops to only 23% for 4,096 instructions. This is because the size of the SRAM can be kept the same although the size of the ROM decoder must increase along with the increasing number of instructions.

Access time analysis

Figure 3(b) 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 10% overhead when the ROM decoder is small (64 words only). Moreover, this access time overhead decreases to less than 3% when the number of instructions reaches 1,024, after which accessing the FITS programmable decoder becomes faster than accessing the ROM decoder: 4% faster for 2,048 instruction words and 13% faster for 4,096 instruction words. Most important of all, with the processor clock frequency targeted at 100 MHz, all read and write accesses to the FITS programmable decoder can easily be finished within one cycle, even under the worst-case scenario.

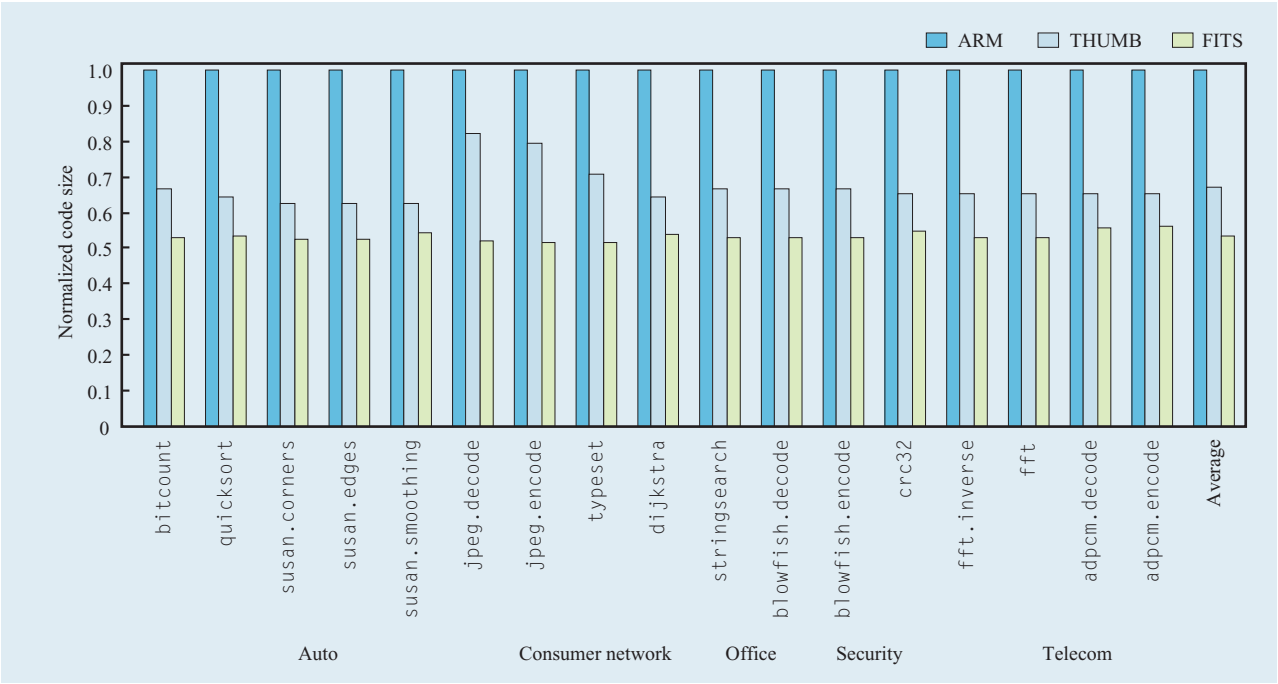

Figure 4

Code size footprint.

Power consumption analysis

The dynamic and leakage power consumption, in milliwatts (mW), of the fixed decoder and the programmable FITS decoder are shown in **Figures 3(c)** and **3(d)**, respectively. The dynamic ac current assumes 50% read and write operations, where all addresses and 50% of the input and output pins switch. The leakage power assumes inactive memory cells with all input and output pins being held stable. The power consumption of the FITS programmable decoder is less than that of a regular fixed decoder, as indicated by negative overhead lines in the figure. As depicted in Figure 3(c), 53% to 66% of dynamic power savings can be achieved by the FITS decoder as the number of instructions supported increases from 64 to 4,096. Similarly, a 20% to 90% leakage power savings can be achieved by the FITS decoder as the number of instructions supported increases from 64 to 4,096, as shown in Figure 3(d). These power savings are due to the fact that the FITS decoder accesses only the small 16-entry SRAM during program execution, whereas a regular fixed decoder must access a much larger ROM that consumes more power to operate. In the FITS decoder, the ROM is powered off after initialization, so there is no power overhead associated with it.

Code size benefits

Figure 4 compares the program code density achieved by different code generations: ARM, THUMB, and FITS. (For the purpose of easy assimilation, all test result figures that follow this section were reduced to show only the averages.) The FITS bars represent the program code size after the ARM-to-FITS translation. The ARM and THUMB bars respectively represent the program code size compiled in pure 32-bit ARM and 16-bit THUMB. The ARM–THUMB intermixing result was omitted because 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 achieved by THUMB and FITS 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 half—on average, 47% of the total ARM segment could be eliminated. THUMB could not achieve the code size savings of FITS because THUMB is not able to utilize its 16-bit instruction fields as efficiently owing to its general-purpose nature. Thus, for an application that has several performance-critical regions, many 32-bit ARM

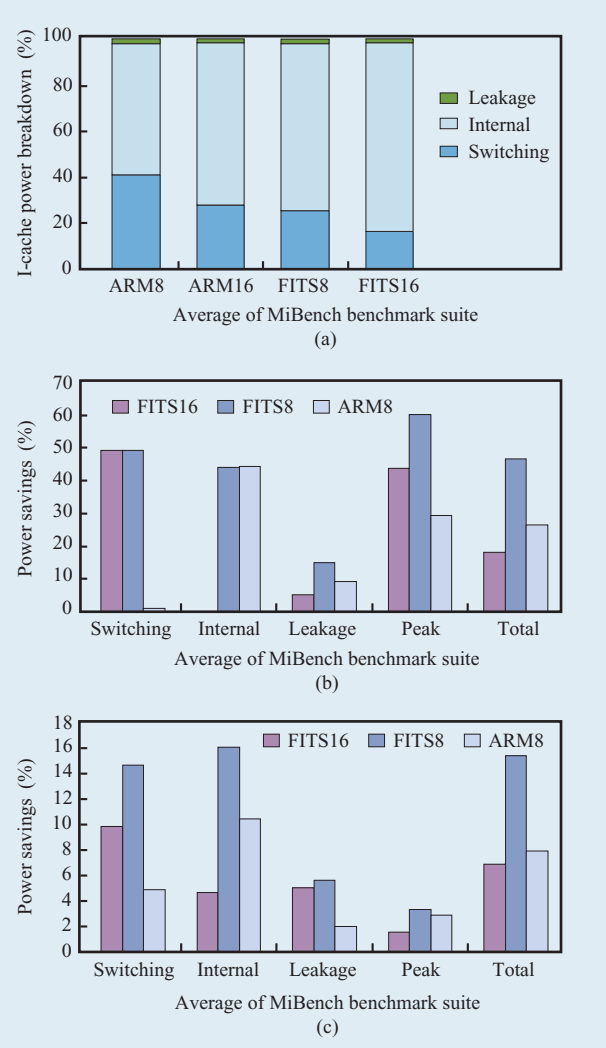

Figure 5

Power dissipation benefits: (a) I-cache power breakdown; (b) I-cache power savings; (c) chip-wide power savings.

instructions would still have 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 many applications. However, this general-purpose capability requires more opcode space and reduces the size of the other instruction fields, such as register and immediate operands. When the register operand width is reduced, the processor can specify fewer architect registers, thus increasing the register pressure. Higher register pressure causes more spillings, thus increasing the number of memory references in the program. For this reason, THUMB is not able to achieve the level of code size savings provided by FITS.

As illustrated by the performance results given later, the code size saving achieved by FITS does not come at the expense of lost performance, for two reasons. First and foremost, FITS aggressively optimized and adopted the utilization-driven synthesis heuristic, making it very effective in determining the target instructions for synthesis without any noticeable performance loss. Second, the resultant half-sized FITS code effectively makes the L1 I-cache almost twice as large as before. Thus, the FITS execution core was able to take advantage of the higher spatial locality exhibited to greatly raise the cache hit rate, thus increasing the performance.

Power dissipation benefits

To reduce overall chip power dissipation, we focused on attacking I-cache power consumption. We show the breakdown of I-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 switching, internal, leakage, and peak powers. The reduction of each component power is then translated into the total I-cache power reduction. Finally, the I-cache power saving is mapped into the corresponding overall chip-wide power saving.

We modeled dynamic, static, and peak power dissipation. Dynamic power was further broken down into switching power and internal power to facilitate monitoring power reduction. Switching power is the power consumed by the output driver and the output load capacitance of the I-cache microarchitecture. Internal power is the dynamic power of the I-cache microarchitecture itself.

I-cache power breakdown

From the I-cache power breakdown shown in **Figure 5(a)**, the following power use trends are noticed. First, the total I-cache power is dominated by the dynamic power (i.e., the switching power plus the internal power). This is expected because the SA-1100 is a relatively low-end embedded microprocessor built with a less aggressive fabrication technology of 0.35 μm, so we would not encounter the same level of current leakage found on designs fabricated with deeper submicron technology.

Second, as the size of the I-cache increases, the percentage of switching power goes down, the percentage of internal power goes up, and the percentage of leakage power remains approximately the same; this is because a larger cache comprises more gates and thus more internal and leakage power. In addition, given the same cache block size and associativity, a larger cache would yield a better hit rate, which means fewer gate switches and reduced switching power.

Third, with the I-cache size being equal, FITS uses a lower percentage of switching power, a higher percentage of internal power, and approximately the same percentage of leakage. The percentage of leakage power remains unchanged because there are equal numbers of gates in caches of the same size. The percentage of switching power is reduced because of the increased cache hit rate of FITS-sized code. The cache size is the same, so the percentage of internal power is increased because of the normalization effect after accounting for the reduction of switching power.

Fourth, if we compare the percentage of switching power between ARM8 and ARM16 and between ARM8 and FITS8, we find that applying the FITS optimization reduces more switching percentage than simply doubling the size of the cache. This speculation is confirmed by the I-cache power savings analysis that follows.

I-cache power saving

Figure 5(b) shows power reduction by FITS in each power component. As speculated in the power breakdown section, FITS-sized codes benefit greatly from the reduction of switching power. This is the power saving that clearly distinguishes a FITS-optimized cache from a normal ARM cache. Both FITS16 and FITS8 save approximately 50% cache switching power, while ARM8 saves virtually none. The switching power saving of FITS results from a better cache hit rate due to the better spatial locality exhibited by FITS-sized codes. On the other hand, ARM8 consumed as much overall switching power as the baseline 16-Kb cache, indicating that 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.

The peak power consumption depends on both the switching frequency and the number 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. Because peak power is sensitive to factors that affect both the dynamic and the static powers, the greater peak power savings of FITS16 and FITS8 indicate that FITS is a well-balanced, low-power technique for I-cache.

This claim is supported by the overall I-cache power consumption results, which combine all of the component savings above. FITS8 gives the highest (47%) average total I-cache power savings, followed by ARM8 and FITS16, which save 27% and 18%, respectively.

To see how effectively FITS reduces the total chip power, **Figure 5(c)** depicts how these I-cache power savings would be translated into the total chip power savings. FITS16 and FITS8 respectively save, on average, approximately 10% and 15% chip-wide switching power, while ARM8 saves 5%. For the chip-wide internal power savings, FITS16 and FITS8 respectively save, on average, approximately 5% and 16%, and ARM8 saves 10%. FITS16 and FITS8 respectively save, on average, approximately 5% to 6% of leakage power chip-wide, while ARM8 saves 2%. Peak power savings can be as high as 6% for FITS16, 12% for FITS8, and 10% for ARM8. The total power savings is 15% for FITS8, 8% for ARM8, and 7% for FITS16.

Performance benefits

To demonstrate that FITS does not save power at the expense of performance, we present results of both I-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 power by much. The IPC analysis shows overall system-wide performance between FITS and ARM. Combining both results leads to the conclusion that FITS is able to reduce power without compromising performance.

Cache miss rate

Figure 6(a) shows the I-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 because of their small code size footprint. FITS surpassed ARM with greatly improved cache performance: The half-sized FITS8 caches have smaller miss rates than the normal full-sized ARM16 caches because of 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) because 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, because 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.

Instruction per cycle (IPC) rate

Figure 6(b) shows the IPC performance measures for all four processor configurations. Overall, the IPC performances for all four configurations are satisfactory because 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

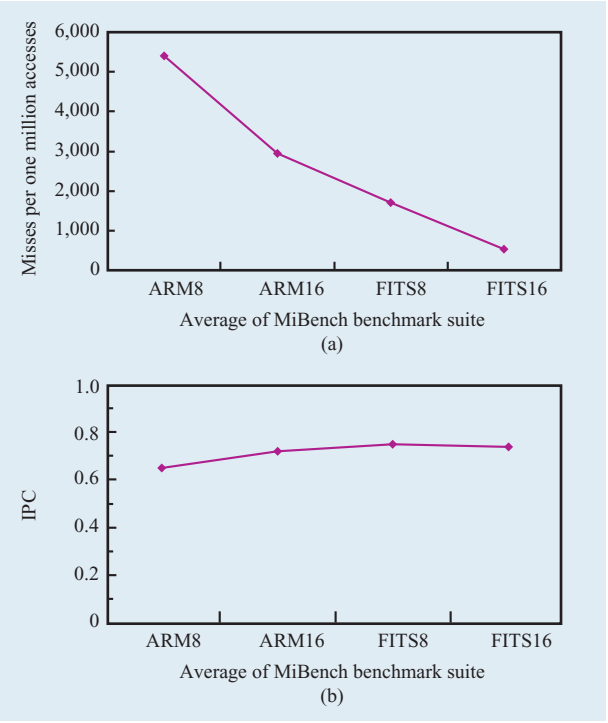

Figure 6

(a) I-cache miss rate; (b) instructions per cycle (IPC).

achieve roughly the same IPC as a 16-Kb ARM cache. We expect FITS to be performance-neutral, but we consistently find a small improvement, and in some applications, a large improvement. This is due to increased I-cache locality exhibited from packed FITS code.

Conclusions

The goal of this research is to argue for a new approach to the design of a class of embedded processors that saves power and energy and reduces code size while maintaining satisfactory performance. This research shows that waiting until after chip fabrication to map the instruction set to the microarchitecture makes it possible to match the dense coding capabilities of ASP while retaining the fabrication advantages of a single-chip design. This delayed ISA mapping is achieved by using a programmable instruction decoder that has minimum overhead in area and access time while reducing both dynamic and leakage power consumption. Using the FITS design methodology enables a cost-effective 16-bit ISA synthesis solution while reducing design time and complexity; this is accomplished by decoupling the microarchitectural enhancements available onchip 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 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.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of ARM Ltd., Tensilica, Inc., STMicroelectronics, or ARC International in the United States, other countries, or both.

References

1. T. Mudge, "Power: A First-Class Architectural Design Constraint," *IEEE Computer* **34**, No. 4, 52–58 (2001).
2. J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf, "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE J. Solid-State Circuits* **31**, No. 11, 1703–1714 (1996).
3. L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, 2001, pp. 110–119.
4. N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, 2003, pp. 129–140.
5. N. Kadri, S. Niar, and A. R. Baba-Ali, "Impact of Code Compression on the Power Consumption in Embedded Systems," *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, 2003, pp. 197–203.
6. 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.
7. IBM Corporation, PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Software Reference Manual, Publication No. G522-0290–01, 2000.
8. Y. Xie, W. Wolf, and H. Lekatsas, "A Code Decompression Architecture for VLIW Processors," *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, 2001, pp. 66–75.
9. L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 1999, pp. 206–211.
10. H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low Power Embedded System Design," *Proceedings of the 37th Design Automation Conference (DAC)*, 2000, pp. 294–299.
11. S. K. Debray, W. Evans, R. Muth, and B. D. Sutter, "Compiler Techniques for Code Compaction," *ACM Trans. Program. Lang. & Syst.* **22**, No. 2, 378–415 (2000).
12. 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.
13. R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro* **20**, No. 2, 60–70 (2000).
14. 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)*, 2000, pp. 203–213.
15. ARM Limited, ARM7TDMI (Rev. 4) Technical Reference Manual, 2001; see *http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf*.
16. ARM Limited, Improving ARM Code Density and Performance, 2003; see *http://www.arm.com/pdfs/Thumb-2.pdf*.
17. MIPS Technologies, MIPS32 Architecture for Programmers, Vol. IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture, 2001; see *http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary*.
18. STMicroelectronics, ST100 Technical Manual, 2003; see *http://www.st.com/stonline/books/pdf/docs/10071.pdf*.
19. S. Zammattio, "How to Reduce Time-to-Market for System-on-Chip Design," ARC International, 2002; see *http://www.arc.com/documentation/whitepapers/*.
20. A. Krishnaswamy and R. Gupta, "Dynamic Coalescing for 16-Bit Instructions," *ACM Trans. Embedded Computing Syst.* **4**, No. 1, 3–37 (2005).
21. S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers," *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2005, pp. 260–271.
22. 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.
23. A. Church, "An Unsolvable Problem of Elementary Number Theory," *Amer. J. Math.* **58**, No. 2, 345–363 (1936).
24. A. Turing, "On Computable Numbers, with an Application to the Entscheidungs Problem," *Proc. Lond. Math. Soc. Ser. 2* **42**, 230–265 (1936).
25. C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing Code Size with Run-Time Decompression," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, 2000, pp. 218–227.
26. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proceedings of the 4th International Workshop on Workload Characterization*, 2001, pp. 3–14.
27. The SimpleScalar-Arm Power Modeling Project, 2004; see *http://www.eecs.umich.edu/~panalyzer*.
28. Intel Corporation, SA-110 Microprocessor Technical Reference Manual, 2000; see *http://www.acm.uiuc.edu/sigarch/resources/docs/sa110_27805802.pdf*.

Received June 21, 2005; accepted for publication July 26, 2005; Internet publication February 23, 2006

Allen C. Cheng *Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Avenue, Ann Arbor, Michigan 48109 (accheng@eecs.umich.edu)*. Dr. Cheng is a lecturer in the College of Engineering at the University of Michigan. He received his M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan. His research interests include low-cost, low-power, high-performance embedded processing platforms with a concentration on novel architectural innovation and compiler optimization. Dr. Cheng is a member of the IEEE.

Gary S. Tyson *Department of Computer Science, Florida State University, 161 Love Building, Tallahassee, Florida 32306 (tyson@cs.fsu.edu)*. Dr. Tyson has been an associate professor at Florida State University since 2003. He was previously a faculty member of the University of California at Riverside and the University of Michigan. He received his Ph.D. degree in computer science from the University of California at Davis. His research spans computer architecture and compiler optimization. Dr. Tyson is a member of the IEEE.