FITS: Framework-based Instruction-set Tuning Synthesis for Embedded Systems (Extended Abstract)

Allen Cheng   Gary Tyson*   Trevor 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

We propose a new instruction synthesis paradigm that falls between a general-purpose embedded processor and a synthesized application specific processor (ASP). This is achieved by replacing the fixed instruction and register decoding of general purpose embedded processor with programmable decoders that can achieve ASP performance with the fabrication advantages of a mass produced single chip solution.

Keywords

16-bit ISA, Instruction synthesis, Embedded processor, ASP, Instruction encoding, Reconfigurable processors, Configurable architecture, Code density, Low-power, Energy efficient

1 INTRODUCTION

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 for 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.

With the wide-spread use of Intellectual Property (IP) cores and the development of configurable processors, customized instruction set synthesis has become feasible to better differentiate products in today's competitive markets. For embedded systems, especially for portable handhelds, code size, energy efficiency, chip area, and time to market are often the most important design constraints to be considered. Designers for 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 embedded ASPs. FITS offers designers a tunable, general-purpose processor solution to meet the 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 a programmable decoder. Through the programmable decoder, we can optimize the instruction encoding, address modes, operand and immediate bit widths, 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 full range of embedded applications that would normally require 32-bit ISAs; (2) it reduces energy consumption 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 numerous functions, but would map only a subset of those to the synthesized instruction set. By only mapping those operating 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 operating that can be found in a large instruction embedded processor.

The contributions of our work are threefold. First, we provide thorough analyses of characteristics of embedded

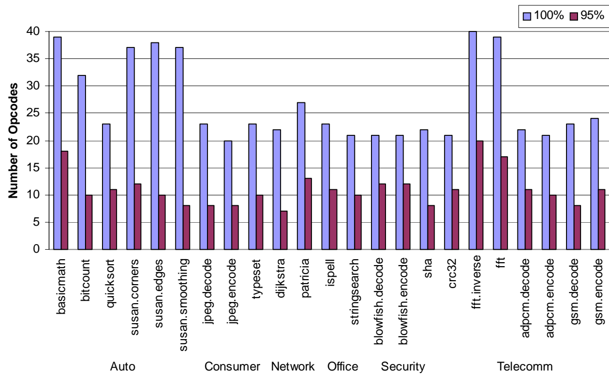

Figure 1: Dynamic Utilization of Opcodes

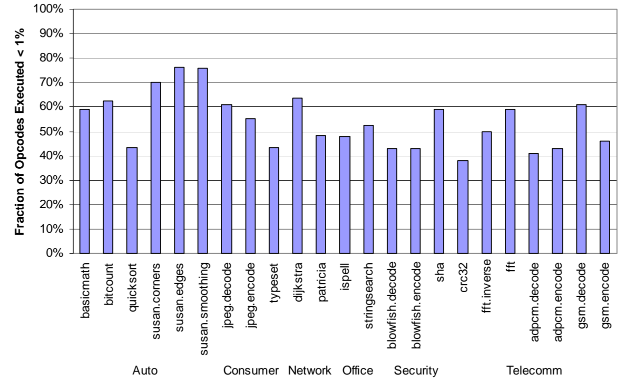

Figure 2: Fraction of Opcodes Utilized < 1% of the Time

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 RISC pipelined architectures. Third, we demonstrate the effectiveness of this framework that utilizes state-of-the-art instruction encoding techniques to leverage compact and efficient designs comparable to a ASP with substantially less engineering cost. Due to the scope of the project and the space limitations imposed here, we will limit ourselves to discuss two issues in this study: the justification that a 16-bit instruction encoding is sufficient for most embedded applications, and a detailed discussion of why a FITS model is best suited to achieve the proper encoding.

2 RELATED WORK

In today's embedded computing industry, one commonly adopted ISA scheme is dual instruction sets. Dual instruction set processors, such as the ARM [2], MIPS16 [5], ST100 [8], and ARCtangent-A5 [1], address the limited memory and energy constraint 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 while trading off the execution time for smaller memory footprint and better energy consumption. As complex applications that require intensive computing power are being ported to embedded platforms, performance or execution speed is becoming equally important.

To meet all of these demands, system designers are often asked to manually blend instruction sets by compiling performance critical code to the 32-bit ISA and the rest to the 16-bit ISA. However, manual code blending is not the best solution because it often requires significant re-engineering efforts to profile and modify the application; this is not only time consuming but can be prone to inserting errors into an existing application. Recognizing this dilemma, ARM has recently introduced the Thumb-2 [3]. Unlike its predecessor, the Thumb-2 is a blended ISA that combines both 16-bit and 32-bit instructions in a single instruction set that enables the designer to mix the instructions together without mode switching. By combining both 16-bit and 32-bit instructions, Thumb-2 uses 26 percent less memory than pure 32-bit ARM code and offers 25 percent more performance than 16-bit Thumb code alone. However, compared to the conventional data-path for a single instruction set, having a data-path that can accommodate two instruction sets of different width will increase circuit complexity and complicate decoding logic, yielding larger die area and higher energy consumption than a data-path supporting only 16-bit instructions.

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, application 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 EMBEDDED WORKLOAD ANALYSES

This section presents important characteristics of embedded applications in terms of their opcode space, operand space, immediate space, and physical register space. A representative subset of the MiBench suite [6] is studied and results of each requirement are discussed in the following sections respectively.

3.1 Opcode Space Requirement

The opcode space in an ISA specifies the number of different instructions a processor may perform. Higher utilization of instructions by an application means more instruction bits need be allocated to opcodes for that application.

Figure 1 shows the number of unique opcodes that were executed dynamically. The 100% bar represents the total

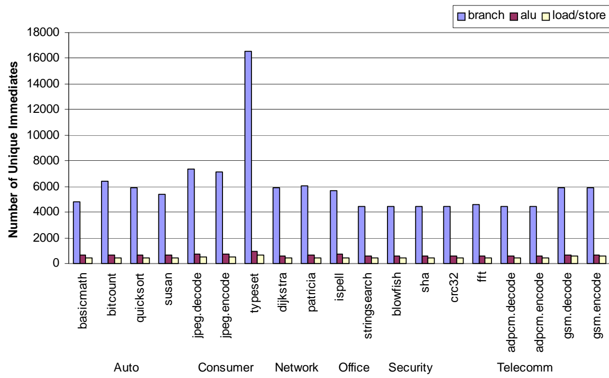

Figure 3: Static Utilization of Unique Immediate Constants

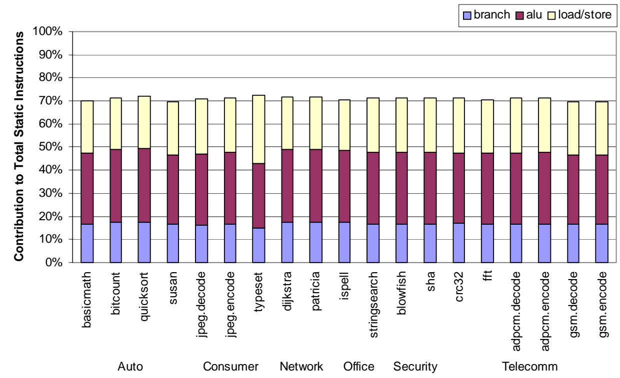

Figure 4: Static Distribution of Immediate-Instructions

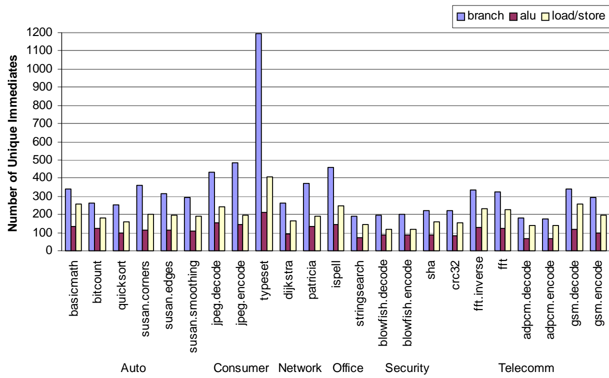

Figure 5: Dynamic Utilization of Unique Immediate Constants

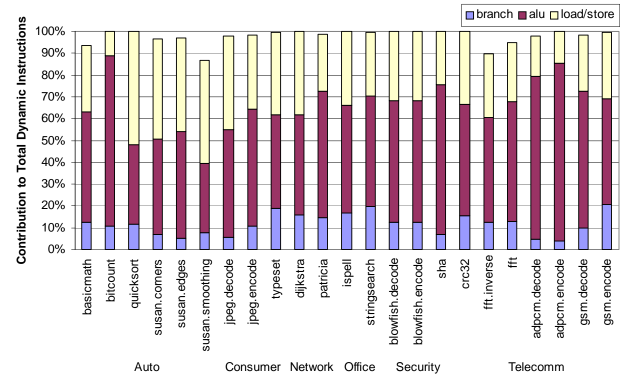

Figure 6: DynamicDistribution of Immediate-Instructions

number of opcodes that a program has utilized. Among all 23 programs, 16 of them (69.6%) utilize 27 or less opcodes; 7 of them (30.4%) utilize between 32 to 40 opcodes. The 95% bar indicates the number of opcodes needed to account for at least 95% of total dynamic executed instructions. Ignoring less than 5% of total dynamic instructions reduces the opcode requirement significantly: 20 out of 23 programs use at most 13 opcodes, while the highest demand does not exceed 20. The reason for the significant reduction in opcode requirement is because not all opcodes are executed equally frequently. This is illustrated in Figure 2 which shows that many opcodes are utilized very rarely: on average, 55.6% opcodes contribute to less than 1% of total dynamically executed instructions. These infrequently executed opcodes can be instead translated into software to save the instruction space without affecting performance significantly.

3.2 Immediate Space Requirement

We classify all immediates into three categories: branch immediates, ALU immediates, and memory (load/store) immediates. The branch immediates are various PC addresses that the program branches to. The ALU immediates are constant values used by ALU instructions to process data. The memory immediates are constant offsets used by load and store instructions to calculate the effective memory addresses. Since the immediate field often occupies significant amount of instruction bits in most ISAs, we carefully examine the space they require both statically and dynamically.

3.2.1 *Static Profiling Immediate Analysis*

Immediate instructions are instructions which have immediate constants embedded in them. Figure 3 illustrates two important aspects of immediate usage. First, it shows their use spreads across the entire benchmark suite: together, they account for approximately 71% of a program on average. Second, it shows a clear distribution for each type of immediate instructions within a program: on average, ALU immediate instructions have the biggest share, with 30.7% contribution to total program size. 23.5% of total program size are memory immediate instructions, and branch immediate instructions have the smallest contribution, with 16.8%.

Static utilization of these immediates may help us to determine the size of immediate operand in instructions. Figure 4 shows the number of unique immediate constants utilized by each category respectively. Despite their small contribution to total code size, branch instructions use the largest number of unique immediate constants and range from 4427 to 16531 with an average at 6020 and a median at 5681. However, all programs except the *typset* use less than 8000 branch immediates. The number of unique ALU and mem-

ory immediates utilized is much smaller than branch immediates. The ALU immediates range from 558 to 956 different values with an average at 648 and a median at 623; while the memory immediates range from 422 to 683 different values with an average at 464 and a median at 428.

3.2.2 *Dynamic Profiling Immediate Analysis*

One disadvantage of looking at static profiling analysis alone is that we may overshoot the requirement since not all immediates are dynamically executed equally frequently. In contrast to the static profiling approach, dynamic profiling allows us to identify the mostly frequently used immediate constants, and thus enabling us to pinpoint the greatest need of immediate constants and allocate the instructions bits accordingly. Therefore, it is necessary to perform both static and dynamic profiling analyses to get a more balanced view of application execution needs.

Figure 5 shows the dynamic distribution of immediate instructions. It strengthens the trend illustrated by Figure 3: On average, 97.7% of all executed instructions are immediate instructions. Within this overwhelmingly large fraction, 53.9% are ALU immediate instructions; 32.2% are memory immediate instructions, and 11.7% are branch immediate instructions.

Figure 6 shows the number of unique immediate constants utilized by each category respectively. Despite the fact they have the smallest share of total dynamic instruction counts, branch instructions again use the largest number of unique immediate constants, and range from 117 to 1193 with an average at 335 and a median at 295. The number of unique ALU and memory immediates dynamically utilized is again smaller than branch immediates. The ALU immediates range from 63 to 213 different values with both an average and a median at 113. The memory immediates range from 121 to 408 different values with an average at 197 and a median at 192.

4 SYNTHESIS FRAMEWORK

This section describes how the FITS design approach can select only those operations that are required for an application, thereby reducing the instruction footprint of the application.

4.1 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 occasionally 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 (as shown in the previous section), 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.

4.2 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. The BIS and SIS together contain enough functionality to simulate any instructions not mapped for an application.

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.

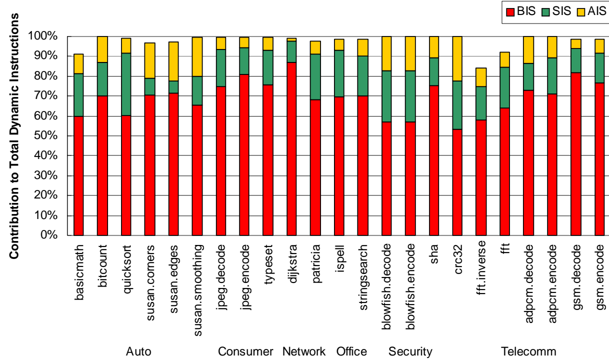

Figure 7: Synthesized Instruction Subset Categories

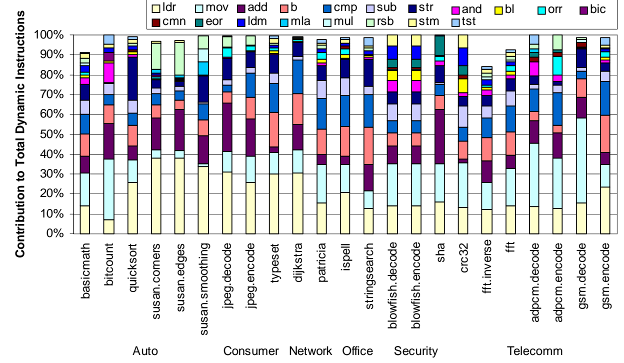

Figure 8: Synthesized Final Instruction Set

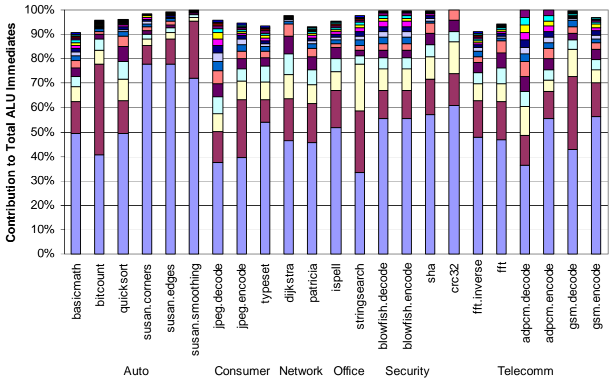

Figure 9: Synthesized Top 16 ALU Immediates

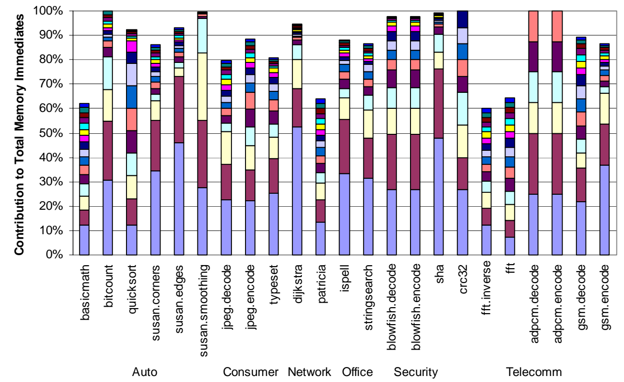

Figure 10: Synthesized Top 16 Memory Immediates

Since the space requirements for different categories of immediates demonstrate distinctive trends, as shown in 3.3, it makes sense to partition the immediate synthesis problem into three sub-categories and perform a category-based synthesis accordingly. FITS adopts an utilization-based technique to encode the immediate operand space. FITS identifies the most frequently accessed immediates and places them them in programmable, non-volatile memory storage, replacing the instruction immediate with an index into the immediate storage. This is similar to the dictionary compression method in [4] 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.

5 EXPERIMENTS AND RESULTS

For our experiments, we evaluated the effectiveness of FITS across a wide range of embedded applications contained in the MiBench. We used the SimpleScalar toolset [7] to examine the quality of the synthesized instruction set in terms of the dynamic execution needs for opcodes and immediates in which it captures.

5.1 Synthesized Instructions

The synthesized final instruction set consists of three subsets of instructions: BIS, SIS, AIS as shown in Figure 7. Figure 8 illustrates the final synthesized instruction set. BIS consists of five opcodes: *load*, *move*, *add*, *branch*, and *compare*. BIS accounts for the majority of the dynamic instruction execution needs: on average, 69.1% of total executed instructions are those from the BIS. SIS also consists of five instructions: *subtract*, *store*, *and*, *branch with link*, and *or*. SIS accounts for 17.9% of total executed instructions on average. The size of AIS is different from one application to another. The union of entire suite's AIS consists of: *bit clear*, *compare negative*, *exclusive or*, *load multiple*, *multiply accumulate*, *multiply*, *reverse subtract*, *store multiple*, and *test bits*. Depending on the individual execution characteristics, each application includes at most 3 to 5 of them and each AIS from different application accounts for 10.8% of total executed instructions on average. Together, the contributions made by BIS, SIS, and AIS account for 97.8% of total executed instructions. This proportion would be even higher if we disregard the contributions made by the floating point instructions and normalize our results accordingly.

5.2 Synthesized Immediates

Figure 9 shows the contribution of the top 16 synthesized ALU immediates to the total number of accessed frequencies of the entire ALU immediate space. It is satisfying to learn that with as few as only 16 unique immediates, the synthesized 4-bit immediate scheme can capture, on average, 96.9% of total number of references made to the ALU

immediate space. It is also interesting to observe that on average, 51.8% of the contribution was made by the most frequently referenced immediate.

Figure 10 shows the results for synthesized memory immediates. On average, 87.4% of total references made to entire memory immediate space were captured by the top 16 synthesized immediates. The exceptions in *basicmath*, *patria*, *fft* and *fft.inverse* were due to excessive utilization of floating point memory immediates. Again, results on both Figure 9 and Figure 10 will be even higher if we normalize the figures to reflect only the behaviors of integer instructions.

6 CONCLUSIONS AND FUTURE WORK

The goal of this research is to argue for a new approach to the design of a class of embedded processors. 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.

7 REFERENCES

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

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

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

[4] C. Lefurgy, E. Piccininni, and T. Mudge, “Reducing Code Size with Run-time Decompression,” in Proceedings of 6th International Symposium on High-Performance Computer Architecture (HPCA), Jan. 2000, pp. 218-227.

[5] K. D. Kissell, “MIPS16: High-density MIPS for the Embedded Market,” in Proceedings of Real Time Systems '97 (RTS97), 1997.

[6] MiBench v.1.0, http://www.eecs.umich.edu/mibench.

[7] SimpleScalar LLC, http://www.simplescalar.com.

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