# Energy- and Area-Efficient Architectures through Application Clustering and Architectural Heterogeneity

LUKASZ STROZEK and DAVID BROOKS
Harvard University

Customizing architectures for particular applications is a promising approach to yield highly energy-efficient designs for embedded systems. This work explores the benefits of architectural customization for a class of embedded architectures typically used in energy- and area-constrained application domains, such as sensor nodes and multimedia processing. We implement a process flow that performs an automatic synthesis and evaluation of the different architectures based on run-time profiles of applications and determines an efficient architecture, with consideration for both energy and area constraints. An expressive architectural model, used by our engine, is introduced that takes advantage of efficient opcode allocation, several memory addressing modes, and operand types. By profiling embedded benchmarks from a variety of sensor and multimedia applications, we show that the energy savings resulting from various architectural optimizations relative to the base architectures (e.g., MIPS and MSP430) are significant and can reach 50%, depending on the application. We then identify the set of architectures that achieves near-optimal savings for a group of applications. Finally, we propose the use of heterogeneous ISA processors implementing those architectures as a solution to capitalize on energy savings provided by application customization while executing a range of applications efficiently.

This article is an extension of the conference paper entitled “Efficient Architectures through Application Clustering and Architectural Heterogeneity” and presented at the *2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. The following new material was introduced, which constitutes about 25% of this article’s contents: (a) focus on the transformation routines and the model parameters in 3, (b) the determination of efficient area-constrained architectures in 4.2, (c) an efficient opcode allocation scheme in 3.3,(d) lessons learned from an application of complex benchmarks in 4.2, and (e) additional optimizations that decrease the running time of the entire process. These additions extend the scope of the previous work: They propose a novel approach for looking at area-constrained efficient designs and offer insights into possible applications of the described process in general-purpose architectures.

This work is supported by NSF grants CCF-0048313 (CAREER), CNS-0330244, Intel, and IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, Intel, or IBM.

Author’s address: Lukasz Strozek and David Brooks, Harvard University, Massachusetts; email: stozek@post.harvard.edu.

DOI 10.1145/1509864.1509868 http://doi.acm.org/10.1145/1509864.1509868

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems; C.1.0 [**Computer Systems Organization**]: Processor Architectures—*General*

General Terms: Performance, Design

Additional Key Words and Phrases: Efficient custom architectures, heterogeneous ISA processors

**ACM Reference Format:**

Strozek, L. and Brooks, D. 2009. Energy- and area-efficient architectures through application clustering and architectural heterogeneity. ACM Trans. Architec. Code Optim. 6, 1, Article 4 (March 2009), 31 pages. DOI = 10.1145/1509864.1509868 http://doi.acm.org/10.1145/1509864.1509868.

## 1. INTRODUCTION

Historically, computer architects have focused on designing instruction sets and microarchitectures that perform well across a broad space of user programs. These architectures are necessary to drive high-volume manufacturing for the general-purpose computing market, and designers often focus on benchmark suites, such as SPEC, that are inspired by a variety of existing applications.

However, as architectures become more and more complex, their performance for some programs becomes far from optimal. This is not a problem if the objective is to execute arbitrary user applications from many application domains. However, due to the increased popularity of embedded devices, such as sensor networks and portable media devices, the opposite trend begins to emerge: Machines are built with very specialized applications in mind. In this case, embedded chip designers should consider the potential of specialized architectures for these high-volume markets, particularly in light of the energy-efficiency demands of these domains.

For instance, consider most sensor network applications. They are executed on small computers (“motes”) equipped with a range of sensors and are usually deployed with a specific task in mind [Hill 2003]: They function as a fire detection system that is independent of the main building infrastructure [Fok et al. 2005], or around volcanoes, measuring seismic activity [Werner-Allen et al. 2005]. When the task to perform is well defined and does not vary over time, it might be advantageous to design specialized hardware for a particular application.

We begin with a custom architecture generation: We design a process that finds an optimal architecture through an automated process. It takes advantage of the fact that most architectures intended for this application domain can be described by a handful of parameters, and creates a generic model of a microcontroller. With minimal human input, the system determines architectures appropriate for a given application, creates a hardware description language (HDL) model for them, then synthesizes the model to output a chip layout design ready to be fabricated. We focus on the analysis of microcontrollers of similar complexity and market focus, such as the TI MSP430 [Texas Instruments 2006], MIPS R2000 [MIPS Technologies], or ARM7 [Budd and Milne 1996]—simple instruction sets implemented on unpipelined or minimally pipelined microarchitectures that are increasingly popular for energy- and area-constrained sensor and media workloads. We then extend this technique to a set of architectures comprising a heterogeneous ISA processor, a chip that implements a set of different ISA architectures.

The main contribution of this article is the exploration of heterogeneous multicores for sets of applications through application evaluation and an automatic synthesis and evaluation of different architectures. By identifying pareto optimal curves of architectures for a given application, an efficient application/architecture clustering can be found.

The process used in this article takes a runtime profile of an application, synthesizes families of "optimal" architectures, and performs efficient code transformations to improve the efficiency of these architectures, where optimality is defined as a function of performance and power. We then relax the single-application requirement and partition a set of all given applications into "clusters," that is, groups with similar efficient architectures. For each cluster, the selected custom architecture is more efficient than the off-the-shelf microcontroller (note that in order to make any comparisons, we will define efficiency/optimality in terms of a specific metric—the energy-delay-square product). Finally, given a partition, we determine the right number of architectures (comprising different cores in a heterogeneous ISA processor) by trading off energy benefits against hardware costs.

Note that we do not advocate multithreaded/multicore designs for sensor devices. In the proposed architecture, only one core is used at a time, but applications can select the most energy-efficient core.

The rest of the article is organized as follows: Section 2 presents the process from a functional point of view and describes (and justifies the validity of) the workflow. Section 3 introduces the architectural model and describes each of the parameters. Section 4 presents the results and analyzes them. Section 5 discusses previous work in automated architecture generation, relevant to the objectives and results of this article, and proposes future work. Finally, Section 6 concludes the article.

## 2. PROCESS FLOW

The process on which our results are based is an architecture generator. Given a particular user program, we would like to produce a layout design of a microprocessor. However, since the search space of possible microprocessors is practically infinite, to make the search for an optimal design feasible, we create a model of a generic microprocessor that takes several parameters and generates the design based on those parameters only. Since such a model limits the search space drastically (in our case, to some 3,840 instances of different microprocessor designs), it should be very expressive in order to emulate many diverse microprocessor designs. In particular, our model includes parameters, such as register file size; various memory addressing modes; the presence of complex instructions, such as a divider or a multiplier; and immediate operands. To determine the best architecture, the process, therefore, accepts a user program as input, and determines a set of parameters that yield the optimal architecture. Two factors affect the decision:

—The profile of any candidate architecture, determined in advance, that will be used to determine the execution time and energy usage of the resulting microcontroller. Since this profile is generated offline, and only once for every architecture embraced by the model, we simply look up the precomputed information about a particular architecture in a database.

—The profile (and the trace information) of the user program provided as input, converted to be compatible with a particular architecture. Since different computer architectures feature different instruction sets, it is necessary to convert the user program to a program with identical functionality, but written using a particular instruction set that we call a "universal assembly language" (the universal assembly is a generic language featuring a number of simple operators). Since the user program is written in a superset of all allowed instruction sets, we wrote a custom software profiler that is able to execute arbitrary applications conforming to the model's requirements.

Those two sources present us with trade-offs; for example, the hardware profiler may report that a simpler architecture results in a microprocessor that consumes less power and has less area, but at the same time, the software profiler may report that on the same architecture, the resulting program takes more cycles to execute. Therefore, the process finds a pareto optimal family of architectures [Fudenberg and Tirole 1983].

## 2.1 Implementation Details

The implementation of the model is a Verilog file that contains the description of the data path. Since memory has such an important impact on the performance of any microcontroller, it cannot be ignored in this model. For that purpose, a memory compiler included in the Faraday standard cell toolkit [UMC Faraday ] called Memaker is used to generate memory, models for the data memory, code memory, and the register file. Different architectures require memories of different bit-widths and number of words. The memory generated by Memaker is incorporated into the final design of the microcontroller.

Consider the process flow shown in Figures 1 and 2, consisting of two parts: offline and online analysis. Offline analysis generates the architecture-specific data that can be stored in the database (this data is independent of the application being analyzed). First, given a particular set of model parameters, a Verilog model is generated (note that we are generating a model of a microcontroller and interfacing it with third-party external memory that is included in the overall model). This model is analyzed by Synopsys Design Analyzer, which synthesizes it and converts it into a mapped design. The Synopsys tool also combines this design with the three memory modules and reports the chip area, the worst-case delay (clock frequency) and power consumed. A correction needs to be made for switching power: Synopsys provides metrics for switching power assuming a certain switching frequency; a linear model is then assumed to determine actual switching power when executing a particular application. Memory usage statistics from the user program are also passed into the power model for the memory (obtained from Memaker). For synthesis, the 1.2V 130nm UMC Faraday standard cells are used [UMC Faraday]. Once Design Analyzer

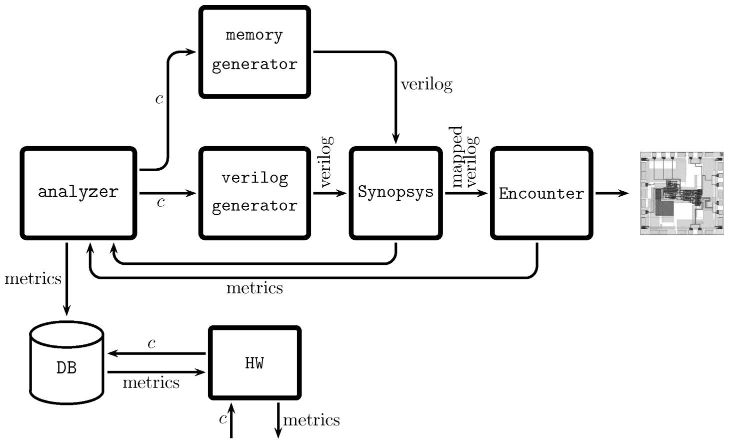
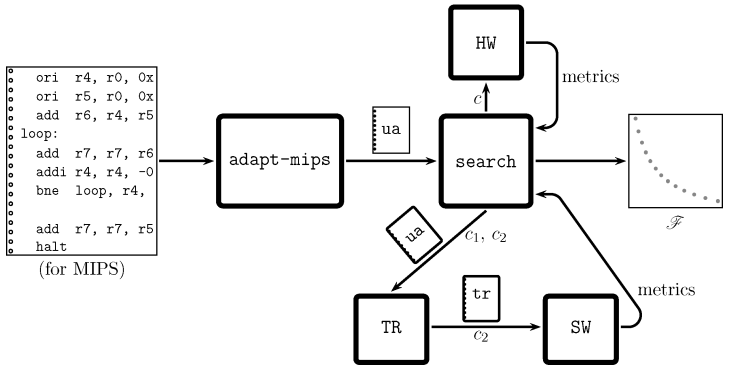


Fig. 1. **The offline part of the process flow**. For each configuration $c$, the analyzer generates the Verilog model, gathers the metrics (the worst-case delay of the circuit, the chip area, and the static and dynamic power consumption) from Synopsys Design Analyzer and SoC Encounter, and saves them in a database. In other words, every possible datapath is generated and its characteristics are saved for later use.


```
  ori  r4, r0, 0x
  ori  r5, r0, 0x
  add  r6, r4, r5
loop:
  add  r7, r7, r6
  addi r4, r4, -0
  bne  loop, r4,

  add  r7, r7, r5
  halt
```

Fig. 2. **The online part of the process flow**. Given a (for example) MIPS assembly file, the adapter first converts it to a file compatible with the universal assembly. For each configuration to test $c_2$, the search module passes the file to the translator, and then the software profiler. Given the metrics from the hardware profiler and the software profiler, the search module determines the family $\mathscr{F}$ of optimal architectures.

generates the mapped design, Cadence SoC Encounter [Cadence Incorporate] performs cell placement and routes the design to provide more accurate area and worst-case delay analysis. The output of those two tools is combined, and the resulting data is stored in the database.

Online analysis is the main component of the process. The code for the user program is written originally in some assembly language for some architecture. This architecture must be one of the architectures embraced by the model, but the instruction set can be arbitrary. However, it is relatively easy to convert that input program to a program that is written in the universal assembly with instructions from the instruction set defined by us. The universal assembly resembles an intermediate language used by compilers, yet all the instructions of the underlying instruction set can easily be implemented in Verilog.

Once the program is adapted (converted to universal assembly), it is then passed on to the search module. The user supplies the search module with a few model parameters that must be held fixed. Those parameters, called *irreducible*, do not vary and are usually common for the original architecture and the optimal architecture.

Since the user program has been written for a particular architecture, it may not necessarily be run on any arbitrary architecture. If not, it needs to be translated (reduced) into a program that will. This means that certain instructions might need to be written as series of instructions, or even entire procedure calls, if the target architecture is too simple to support those instructions. It is important to note that all translations are lossless, that is, they convert the program into a program with identical functionality. It may take more cycles to execute, but for an outside observer, it does not differ in function from the original program. The architecture model we implemented ensures that a lossless translation is always possible; that is, for any configuration of the reducible parameters (all parameters that do not have to be kept fixed), it is possible to translate the program to one supported by a model generated with any other configuration of those parameters. Moreover, converting a simple architecture into a more complex architecture means that applications need to be further optimized. Usually, certain groups of instructions are collapsed, which significantly increases program performance.

Note that most architectures contain specialized instructions or features which cannot be losslessly translated into instructions of other architectures (e.g., Load Linked Word in MIPS). However, those instructions are usually uncommon in most applications—for all benchmarks that we have considered, inconvertible features comprised less than 2% of all instructions and, since our analysis ignores I/O time, took less than 5% of the CPU's execution time.

## 2.2 Discussion of Alternative Solutions

One alternative is to design custom hardware built to perform a certain class of tasks. In the case of high-volume custom processors, it may be difficult to come up with a custom architecture because the design is not synthesized; rather, the engineering team proceeds straight to circuit design and layout. However, the reality is that full custom design is very expensive, and hence is only common in ultra-high performance (e.g., high-end Intel processors). In fact, nonrecurring engineering costs are too high to justify custom design on the low-end embedded processors, which cost a couple of dollars at most. With increasing design complexity and better CAD tool support, full custom design is becoming much less common [Bergamaschi et al. 1995] where performance is not critical, for example in high-volume embedded processors, which are mostly synthesized.

The process presented in this article performs assembly-to-assembly translation of user programs. While such translation results in programs that are correct (programs that are identical in function to the original programs), it is unlikely that the translation is optimal. In particular, if the target architecture is known at compile time, the compiler could use the extra information to make additional optimization decisions. For example, converting from an architecture that supports indirect loads to an architecture that does not allow them means that every indirect load must be rewritten in terms of two loads. If a compiler had been given an architecture with no indirect loads, it might be possible to make optimizations that would avoid them altogether in some cases.

Hence, another approach that could be used to perform the reductions is to use a retargetable compiler instead of an assembly-to-assembly translator. This approach, however, has certain limitations. First, while some compilers have an ability to modify a target slightly (for instance, by parameterizing the number of registers of the target architecture), we are not aware of any existing compiler that can be reconfigured to target all of the architectures embraced by our model. Specifically, for some parameters, such as the architecture type, different compilers are required altogether, and for some parameter, configurations, no compiler exists. Even if all compilers were available, they would vary in performance and such variation would be impossible to decouple from the simulation results. Finally, given the unpipelined design space that we consider, many compiler optimizations would not be necessary.

Rather than developing a retargetable optimizing compiler that can embrace all the architectures that we consider, we adopt the assembly-to-assembly translation approach and borrow algorithms from the compiler world where appropriate. Specifically, the register reallocation algorithm has been inspired by solutions seen in open-source compilers. Finally, the results presented in this article are conservative: We find more efficient architectures but makes no claim about the absolute efficiency. It is possible that with additional optimizations early in the compilation stage, a more efficient architecture could be found, but the architectures found by this process are nonetheless more efficient than the original one.

Given the discussion above and the simplicity of our base architecture, we feel that this approach is sufficient. Furthermore, Section 4.5 shows that our results are relatively insensitive to the choice of the baseline architecture and compiler.

## 3. ARCHITECTURAL MODEL

The results presented in this article rely heavily on the underlying architecture model. In order for the process to find optimal data path designs, this model must be as expressive as possible, allowing for programs written for existing architectures (such as MIPS or TI MSP430) to be easily adapted. It must have a balanced number of parameters and always allow a lossless conversion between configurations.

The underlying architecture is RISC-like, and the supported operands are registers, immediate values, and memory offsets. The architecture supports reading from and writing to external memory, which significantly extends its functionality. However, since the model includes fairly simple architectures, the Verilog implementation is not pipelined (so as to reflect the MSP430). In most configurations, instructions take three cycles to execute. Some configurations have instructions that require a greater number of cycles (e.g., the stack architecture might need to address the data memory up to four times).

Table I. Model Parameters and Their Values

| Parameter Name | Values | Description |
|---|---|---|
| REGISTER_COUNT | 0 (8)<br>1 (16)<br>2 (32)<br>3 (64) | The number of registers. |
| ARCHITECTURE_TYPE | 0 (ARCH_ACCUMULATOR)<br>1 (ARCH_STACK)<br>2 (ARCH_OFFSET)<br>3 (ARCH_REG_OFFSET)<br>4 (ARCH_INDIRECT) | How is memory accessed? If through loads/stores (type $\geq$ ARCH_OFFSET), how is the address generated? In such case, cumulative. |
| COMPLEX_UNIT | 0 (COMPLEX_NONE)<br>1 (COMPLEX_SHIFT)<br>2 (COMPLEX_MUL)<br>3 (COMPLEX_DIV) | Include complex arithmetic modules? Cumulative: a particular value includes all preceding modules. |
| THREE_SOURCES | 0 (no)<br>1 (yes) | Include instructions with three source registers (*i.e.,* beqr, bgtr)? |
| IMMEDIATE_WIDTH | 0 (IMM_NONE)<br>1 (IMM_HALF)<br>2 (IMM_FULL) | Include instructions with immediate values? |
| DATAPATH_WIDTH | 0 (4-bit)<br>1 (8-bit)<br>2 (16-bit)<br>3 (32-bit) | The size of memory word and the size of each register (in bits). |
| EXTENDED_ADDRESSING | 0 (no)<br>1 (yes) | Allow for long jumps and addressing 2^(2·DATAPATH_WIDTH) bytes? |

ARCHITECTURE_TYPE, when greater than 1, and COMPLEX_UNIT are cumulative, which means that particular value of a parameter includes the functionality of all the values less than it.

First, thier we discuss the parameters of the model, their possible values, and, their impact on the resulting architectures. Next, we describe the translations necessary to convert between various architectures and the optimizations that we employed. We then introduce the problem of opcode generation. Finally, we put the model in the context of existing architectures.

## 3.1 Model Parameters

This section describes all the parameters of the data path that determine the complexity of the subarchitectures that we model. We consider seven parameters, each of which can take one of several values. All parameters are divided into two classes: reducible parameters and irreducible parameters. A complete table of parameters, together with their values and descriptions, is shown in Table I. The bottom two parameters are irreducible. The search only includes reducible parameters so that the total number of distinct architectures considered is 480.

The choice of parameters was not arbitrary. The degree to which the resulting metrics are sensitive to the change of initial parameters varied depending on the parameter chosen. The parameters that we decided to include in the model had the greatest impact in the energy, power, and chip area metrics. Moreover, several other candidates were evaluated and rejected for some reason.

Specifically, we constrained ourselves to simple nonpipelined microcontrollers, so we did not consider pipelining or multiple issue.

The parameters that we decided to include in the model were fairly uncorrelated, with the average correlation equal to 0.18 and the 90th percentile equal to 0.31. This had two implications: First, low correlations meant that no parameter could have been replaced by a combination of other parameters. Moreover, the savings were path-invariant: Individual energy savings were similar to one another regardless of the order in which the optimizations were applied to the original architecture.

The model allows for three different ways to access data memory: accumulator, stack, and load-store. In accumulator mode, the register `r1` becomes a memory-mapped accumulator. In stack mode, the register `r1` becomes a stack pointer—using it as a source register pops data off the stack; using it as a destination register pushes data on the stack. If the architecture is a load-store architecture, different addressing modes are available: offset, register-offset, and indirect.

The reason for explicitly singling out a memory-mapped accumulator and a stack architecture is based on an observation that many applications written for embedded systems (including several of the benchmarks used in this article) can be rewritten to take advantage of those specific memory addressing modes.

The parameter that affects the energy and execution time of the generated architectures the most is `REGISTER_COUNT`. The classical trade-off (more registers means a slower and larger datapath; fewer registers costs memory for immediate results) is brought to a test when translations between architectures with various register counts are possible. This is achieved with help of a register-reallocation algorithm.

When `DATAPATH_WIDTH` is low, so is the size of any address, and so the programmer is restricted to very short programs that operate on small amounts of memory. To give the programmer a chance to extend the program's addressing space, `EXTENDED_MODE` has been introduced. Special instructions are introduced that allow a program to make long jumps (with twice the number of address bits) and select data banks (thus effectively increasing the range of addressable memory).

`IMM_WIDTH` describes how the model treats immediate values. They can be disallowed (in which case it becomes difficult to load constants into memory or registers, but the instruction width reduces dramatically, especially in architectures that have high `DATAPATH_WIDTH`. They can be allowed, which comes at a high cost associated with a large instruction width. A hybrid approach allows immediate values, but restricts their values to half the bitwidth this provides an interesting compromise between the two extremes and is often a path toward a more efficient architecture.

One should note that all the parameters are orthogonal; that is, any parameter can be changed independently of the other ones, and any configuration yields a valid architecture.

## 3.2 Instruction Set Architecture and Translations

The instruction set defined in this article consists of the following classes of instructions: arithmetic instructions (featuring, depending on `COMPLEX_UNIT`, just simple logical operations, shifts, or even a multiplier and a divider), immediate arithmetic instructions (provided that `IMM_WIDTH` is nonzero), branches, jumps and loads, and stores.

We now discuss some of the key parameter reduction techniques. Since translations are lossless and complete, there must exist a way to fully express the capabilities of one ISA in terms of another. A special class of instructions is introduced to facilitate those reductions. In particular,

—`REGISTER_COUNT` is reduced using a standard register reallocation algorithm featuring a linear scan algorithm [Poletto and Sarkar 1999]. The objective function is to minimize the number of spills to memory (since memory operations are much more costly). When a spill does occur, two of the registers are used as temporary registers for memory operations. Additional four registers (`r0` through `r4` are fixed) to accommodate jumps to variable addresses.

—Converting between different architecture types is somewhat easier. Instructions `sacc` (set the accumulator pointer) and `ssp` (set stack pointer) have been introduced to easily convert between accumulator and stack architecture and other architectures. `sdp` (set data pointer) is introduced to convert offset architectures to register offset ones.

—Converting between different values of `COMPLEX_UNIT` requires software emulation. To convert from `COMPLEX_SHIFT` to a simpler architecture, a routine is included in the code that performs the shift. Similarly, a multiplier and a divider are included, if necessary. Note that while the software-emulated divide operation takes many more cycles than a built-in divide if the division happens rarely, it might be advantageous to eliminate this module from the architecture.

—Converting between `THREE_SOURCES` is trivial—an instruction that requires three source registers can be written in terms of two other instructions.

—Converting between values of `IMMEDIATE_WIDTH` needs software emulation of the immediate operand. When converted to half-size, any operation that requires an immediate operand is rewritten using two immediate loads and a shift. When converted to no immediates, any immediate must be reconstructed with a sequence of logical operations such as shift and nor. This is obviously very costly if a large number of immediates are used, but it significantly reduces instruction size and thus the size of instruction memory.

If the translation featured only reductions, then there would never be any reason to translate from a simpler architecture to a more complicated one. For this reason, translation also features certain optimizations that may make it advantageous to increase the complexity of the architecture for particular applications. The optimizations include:

—Implicit accumulator used as source—if a load is immediately followed by an operation that uses the loaded data as one of the sources, and if the data is not used later, the two instructions are reduced to one operation that uses accumulator as a source register.

—Similarly, if implicit stack or implicit accumulator are used as destination, the instructions can be optimized.

—Implicit offset-based loads and stores—often, loads and stores are local, that is they operate within a variable offset from a fixed base (which is the value of a register). In such cases, it is advantageous to implement the program using an offset-based load-store architecture.

—If a certain routine is used instead of a missing arithemtic unit (such as a multipler), the entire runtime call can now be replaced with a simple instruction. The translator locates such instances and simplifies the code by collapsing multiple instructions. This is often the source of most substantial savings.

—Immediate value loading—often, on architectures that are missing immediate values, programs load very small immediate values, such as 1, 2, or $-1$, by performing a sequence of bit shifts. The translator can identify such cases and replace series of operations with a single operation with an immediate operand.

The opcodes for each instruction are selected so as to minimize the length of the longest instruction. Hence, the opcodes are variable-width, but the instructions are fixed-width to simplify the PC logic. The bit-width of instruction memory depends most significantly on `IMM_WIDTH` and `DATAPATH_WIDTH`.

## 3.3 ISA Generation

The chip area is highly sensitive to the width of the instruction bus. Hence, any generated architecture must include an optimized instruction encoding that does not waste bits. A major component of this is the determination of opcodes for the instructions included in the architecture. Because the instruction set is a function of the configuration chosen, an efficient method of generating opcodes is required.

The architectural model was chosen to support fixed-length instructions with variable-length opcodes. Fixed-length instructions offer fast jumps and fetches, and variable-length opcodes offer dramatically reduced instruction widths based on a simple observation that some instructions require fewer operands than others, and so we can afford to assign long opcodes to instructions that have few operands.

The problem can be stated as follows: Given a set of instructions $\mathscr{I}$ where the operands of an $i$-th instruction are a total of $L_i$ bits wide, determine a set of opcodes $\mathscr{O}$, where the size of the $i$-th opcode is $O_i$, such that the worst-case total instruction width is minimized, that is, determine

$$\arg\min_{\mathscr{O}} \max_{i} L_i + O_i \tag{1}$$

It is tempting to look at the relative frequencies of the operations used in applications and use a variation of Huffman codes [Huffman 1952] to solve this problem. However, in this case, Huffman does not produce optimal opcodes. Since instructions are fixed in width, the hardware cost is given by the *worst-case* instruction length, not the amortized case length, which Huffman is designed to address. In fact, we prove that a remarkably simple greedy algorithm generates opcodes that yield provably shortest total instruction width. The idea behind the algorithm is to assign the shortest possible code to an otherwise longest instruction and then continue assigning codes that have a prefix property. If we are successful in assigning a code to every instruction, we have found an optimal assignment (up to isometries), since the longest instruction has the shortest possible opcode. The running time of this algorithm is $O(n \log n)$, where $n = |\mathscr{I}|$.

It is important to note the trade-off between the instruction width and decode complexity—simpler opcodes require less hardware to decode. It can be shown, however, that the chip area benefits from decreasing the instruction width by one bit far outweigh the benefits resulting from creating a simpler decode logic. This can be briefly justified if one realizes that a more complicated decode logic affects just the instruction decoder, while a longer instruction affects the entire microcontroller.

### 3.4 Existing Architectures Embraced by the Model

In theory, any RISC-like architecture should be easily portable to one of the subarchitectures described above. In practice, different commercial architectures have features specific to the particular architecture (such as the existence of special instructions or registers, the handling of exceptions and system calls) that make it difficult to proceed with the port. However, with minor changes, programs written for MIPS and TI MSP 430 are supported by the model.

—MIPS R2000 programs can easily be converted into programs that run on a subarchitecture with the following configuration: 32 registers; Register-offset load-store architecture type, a shifter, a multiplier, and a divider included; three source registers included, instructions with half-width immediate values included; and a 32-bit data path with no extended addressing. Since the universal instruction set is inspired by MIPS, it is no surprise that MIPS programs can be run by one of the subarchitectures. However, MIPS programs are modified so that they do not use system calls or rely on exceptions to work (except overflow and division by zero)

—Texas Instruments MSP 430, used widely in sensor network applications, features a small instruction set with a number of special features. Programs written for the MSP 430 can be run by a subarchitecture with the following configuration: 16 registers, indirect load-store architecture type, no complex arithmetic circuits, no three source registers, instructions with half-width immediate values included, and a 16-bit data path with no extended addressing. A program must be modified to not rely on interrupt vectors, data from peripherals (however, some functionality can be emulated), or status register bits beyond Carry, Overflow, and Zero

## 4. RESULTS AND ANALYSIS

The process described in this article can be used to analyze various applications and determine optimal architectures for single programs, or entire classes of applications. In this section, various benchmarks are used to validate our claim.

First, for three popular benchmarks, a Pareto optimal family of architectures is determined. Given this family, the user can then apply a utility function to determine an optimal architecture that satisfies a particular condition. Specifically, we pick an $ED^2P$ utility metric and applies it to the family, thus determining the architecture that maximizes this utility. This architecture is then compared to the original architecture (MSP430) with respect to performance and energy.

Since having a custom architecture for each application is impractical, the set of all benchmarks is partitioned into clusters, that is, groups with similar performance for a particular architecture (called the optimal cluster architecture). The optimal cluster architecture is assigned so that it maximizes the total utility (in terms of $ED^2P$) of all applications in the cluster.

Finally, we attempt to determine whether using heterogeneous multicores is advantageous by looking at the relationship between the number of optimal architectures allowed for an entire class of benchmarks and the performance benefits. We find that for each class, a small number of architectures offers benefits nearly as large as the extreme, one-architecture-per-application solution.

## 4.1 Experimental Setup

The following experiments use benchmarks from four sources: MiBench [Guthaus et al. 2001], a freely available embedded benchmark suite; RAW [Babb et al. 1997], a suite for general purpose computing; and standalone applications and portions of the TinyOS kernel and user program code [Hill 2003]. Table II describes all the benchmarks used. The benchmarks have been compiled for two architectures (MIPS and MSP430) using the MIPS `gcc-2.6.3` cross-compiler and the GCC toolchain MSPGCC, respectively. Most of the results presented in this article use the MSP430 as a reference architecture, though for validation purposes, we compile the benchmarks for MIPS and rerun the process to see how sensitive our results are to the initial conditions. Three application classes are identified: sensor network applications, multimedia applications, and general-purpose applications.

Most of the benchmarks we focus on are very appropriate for sensor networks. Since many of these applications are fairly simple, we also investigate the impact of more complex workloads. Benchmarks such as `perl` or `gcc` are used to show the Pareto optimal family of architectures depends on the complexity of the application. While we do not expect users to run gcc on sensor systems, we chose gcc as a representative of more complex application. We found heuristically that portions of `gcc` had characteristics that were similar to complex embedded applications, such as advanced network routing protocols with fail-over and efficient routing.

Note that since the model does not support every single instruction in the base architecture, the applications that we experimented on required some tweaking. While this is relatively straightforward to do for simple applications, for complex applications, this process becomes infeasible. Hence, complex applications (especially applications requiring support for operations unsupported in our model, such as input/output or interrupts/exception handling) were not

Table II. Benchmarks Used

| Application Class | Benchmark | Source | Description |
|---|---|---|---|
| Sensor Network | dijkstra | MiBench | Shortest Path algorithm |
| | patricia | MiBench | Trees with sparse leaf nodes |
| | rijndael | MiBench | 192-bit key Block cipher |
| | TEA | Standalone | Tiny Encryption Algorithm |
| | TinyDB | Standalone | A Query Engine Application |
| | surge | Standalone | A multihop routing application |
| | kinit | TinyOS | Kernel Initialization Routines |
| | queue | TinyOS | Queue control mechanism |
| Multimedia | jpeg | MiBench | JPEG encoder and decoder |
| | lame | MiBench | MP3 encoder |
| | mad | MiBench | High-quality MPEG audio decoder |
| | tiffdither | MiBench | Dithers B&W image |
| | tiffmedian | MiBench | Reduces Color Palette of Image |
| | mp4enc | Standalone | MP4 encryption algorithm |
| | bicubic | Standalone | Bicubic resize algorithm |
| | pngdec | Standalone | PNG decode algorithm |
| General Purpose | FFT | MiBench | Integer Fast Fourier Transform |
| | CRC32 | MiBench | Cyclic Redundancy Check |
| | stringsearch | MiBench | Case Insensitive Comparison |
| | newton | Standalone | Newton's Approximation of Roots |
| | qsort | Standalone | Quicksort algorithm |
| | life | RAW | Conway's Game of Life |
| | matmult | RAW | Integer Matrix Multiply |
| | jacobi | RAW | Jacobi Relaxation |

Three classes of applications are identified: sensor network, multimedia, and general purpose. All benchmarks come from four sources: MiBench, TinyOS, RAW, and applications not part of a standard suite.

executed end-to-end. Instead, we sampled instructions from the trace of the application. Specifically, `perl` and `gcc` are complex SPEC benchmarks that required sampling.

To determine a Pareto curve, we use the following metrics:

—machine performance (measured in microseconds), that is, the time it takes a particular machine to execute a particular benchmark (or part of a benchmark)

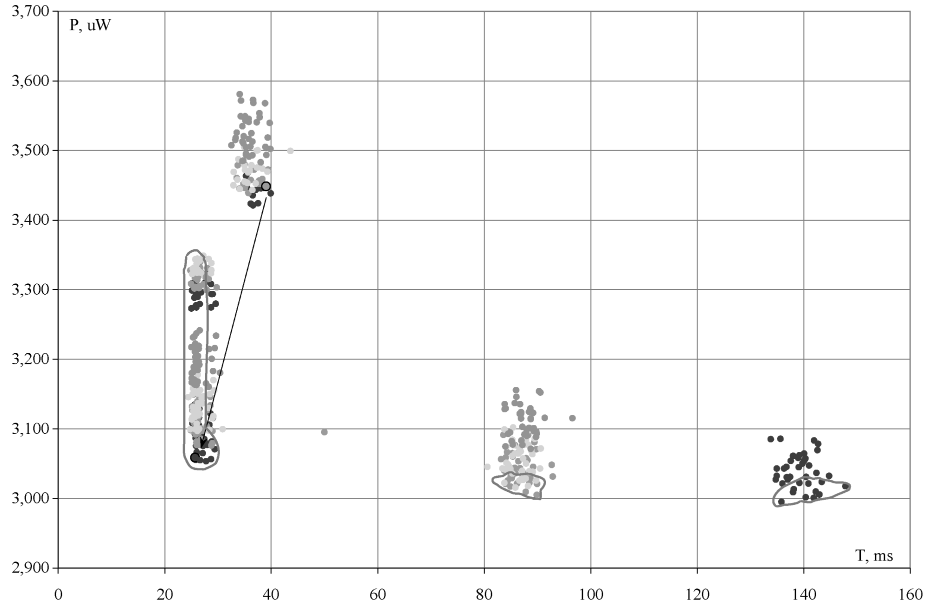
—energy (or power) consumed by the machine while executing the application

The energy-delay-squared product ($ED^2P$) is used as the utility function throughout the experiment [Brooks et al. 2000]. This has the advantage of providing a voltage-invariant view of machine performance. Hence, in Section 4.2, we determine the Pareto optimal architectures by trading off performance and energy (power) and applying the $ED^2P$ utility function to focus on a particular architecture. Similarly, in Sections 4.3 and 4.4, we use the $ED^2P$ metric to determine optimal architectures for a group of applications.

## 4.2 Determining Pareto Optimal Architectures

We first consider the problem of determining the optimal architecture for each benchmark. The savings we obtain yield an upper bound on how much savings can be achieved through application-specific architectures. Figures 3 through 4

(a) fft



(b) lame

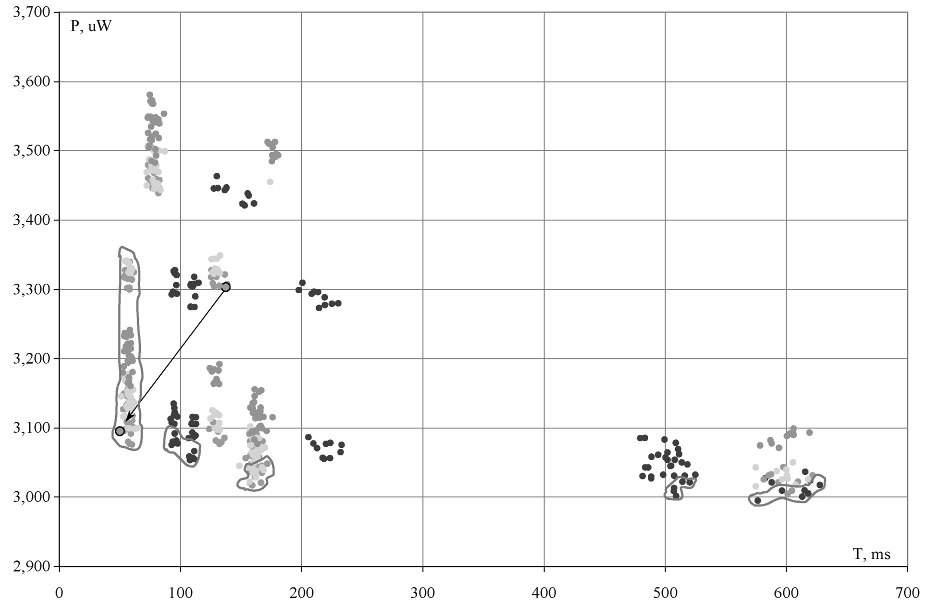


Fig. 3. **Pareto optimal families of architectures.** The figures show all the architectures plotted on the performance-power graph for (a) `fft` and (b) `lame`. In this experiment, the running times of the benchmarks have not been normalized. Different shades of data points correspond to different numbers of registers. The circled data points are the Pareto optimal family of architectures. Two outlined data points, connected with an arrow are the original architecture (where the arrow begins) and the architecture that maximizes the utility function $ED^2P$ (where the arrow ends).

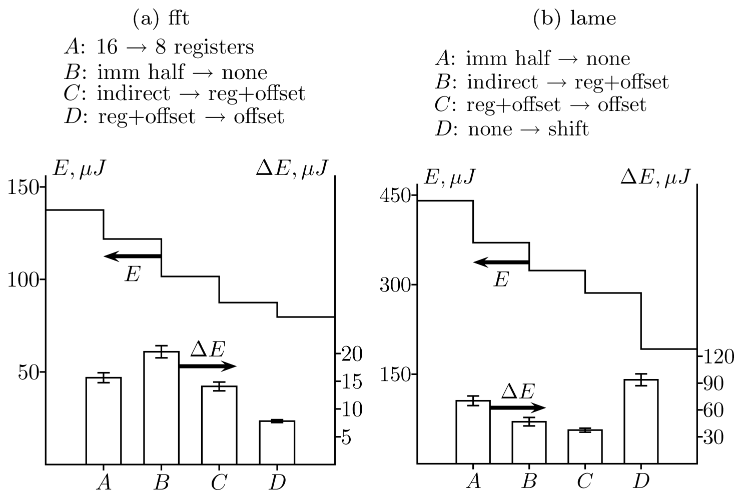


Fig. 4. **A detailed illustration of energy savings.** For graphs 3(a) and (b) respectively, these graphs show a detailed analysis of energy savings. The bars show average savings for each of the four optimizations listed above the graph. The step line shows cumulative savings for each additional optimization, starting from the base architecture, and ending on an architecture that outperforms it the most.

guide us through the entire process for three sample benchmarks, `fft`, `lame`, and `rijndael`. These benchmarks have been compiled for the MSP430 and translated into every possible configuration. For each configuration, the machine performance and power consumption are plotted on a scatter diagram. Then, a family of architectures within 5% of the Pareto ideal curve is determined. For reference, the architecture that corresponds to MSP430 is emphasized on the diagram.

Figures 3(a) and 3(b) display performance-power graphs for two sample benchmarks, `fft` and `lame`, respectively. The Pareto optimal architectures lie on a very steep curve (it is almost vertical for small values of $T$ and almost horizontal for small values of $P$), which signifies that small savings in one metric are often made at a sacrifice of large increases in another. Groups of architectures so apparent in Figure 3(a) appear because the metrics are more sensitive to some parameters than they are to others. In this case, such a parameter is `COMPLEX_UNIT`. It is interesting to note horizontal gaps in the graphs (e.g., in Figure 3(a), the next fastest family of architectures is about three times slower than the leading one), much in contrast with a rather continuous vertical streak. This points to the fact that the execution time is much more sensitive on the type of architecture than power/energy consumption, which is more continuous.

Note also that Figure 3(b) is less sparse than Figure 3(a). This is due to the fact that `lame` performs computations which are much more complex than `fft` (which is a relatively simple algorithm), at the expense of lower branching. This complexity means more clusters of architectures, which is apparent in the graphs.

For simplicity, we use the the $ED^2P$ metric to constrain ourselves to one architecture as opposed to the entire family of them. As shown in Figure 4, for

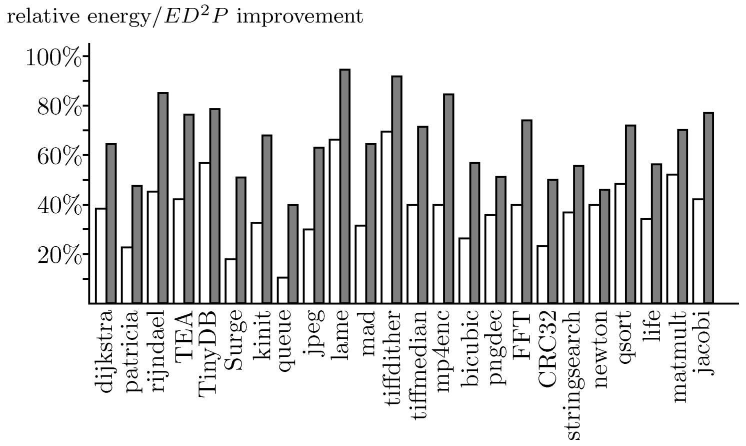


Fig. 5. **Energy and $ED^2P$ savings relative to MSP430.** For each benchmark, an optimal architecture is determined and the energy (white bars) and $ED^2P$ (gray bars) savings are identified.

that architecture, we determine how much energy is saved each time we change a value of one parameter of the configuration, when moving from MSP430 to the optimal architecture. For each change, the energy savings (averaged over all possible paths from MSP430 to the optimal architecture) are determined and reported. Low standard errors suggest that the energy savings are nearly path-invariant, that is, the savings are the same regardless of which path was taken from MSP430 to the optimal architecture. The reason for this is the modular design of the microcontroller model we presented: Each of the separate modules (the memory, the arithmetic unit, etc.) could be replaced with a more or less powerful one, so the energy savings are in large part due to these variations themselves and not the interactions between the modules.

This is an important result for two reasons. For one, it means that more meaningful analysis of the impact of various components on the total energy savings can be performed (i.e., we can make meaningful comments on how much energy introducing a 16-register module gives us for a particular class of applications). This consistency in savings across different configuration validates our findings.

In general, we find that the application path length dominates clock frequency in determining application performance. While power dissipation can fluctuate by 20% to 30% across the modeled architectures, application performance tends to dominate the total energy savings.

The total energy and $ED^2P$ savings are evaluated for every benchmark and plotted in Figure 5. We see that application customization can result in impressive savings for many of the benchmarks that we consider; for the $ED^2P$ metric, often on the order of 75% to 85%.

Furthermore, more complex benchmarks were considered to determine what happens to the Pareto optimal families as applications increase in complexity. The Pareto family has more members, with more diverse architectures appearing on the curve. This is particularly important as most real-life applications

Table III. Table Comparing the Consistency of Energy Savings as a Function of Application Complexity

| Application Class | Loop density | Instruction complexity | Variance |
|---|---|---|---|
| Sensor Network | 1.0 | 1.0 | 1.0 |
| General Purpose | 6.7 | 3.5 | 12.3 |
| Multimedia | 9.5 | 6.8 | 26.0 |
| Complex | 12.4 | 7.4 | 41.0 |

For each of the three application classes, and a class of complex applications (SPEC benchmarks such as `gcc` and `perl`), the relative loop density and instruction complexity is measured and the variance of energy savings across all applications (or, in case of the complex applications, across all samples of applications) is calculated. All numbers are relative to the simplest application class—the sensor network benchmarks. Low variance means that an architecture chosen as an optimal one for one application is likely to provide energy savings for other applications in the same group. Higher variance means that an optimal architecture is less likely to provide consistent savings when applied to other applications within the same group.

are relatively complex and one can take advantage of that by optimizing the diverse set of operations being performed on various architectures.

The drawback of applying these techniques to complex applications is that the variance of energy savings increases with application complexity, that is, architectures determined as optimal for particular complex applications are less likely to remain so when a different but similar application is chosen. Table III presents the result of an experiment in which efficient architectures were found for each application. The energy savings were grouped by application "complexity" and the variance of the resulting distributions was measured. The complexity of an application is heuristically defined as a function of the loop density (the number of jumps as a fraction of the length of the program trace) and instruction complexity (the average number of instructions with distinct operand types) used in the application. In case of more complex applications, a large number of samples was examined from each execution trace and a distribution of savings for each sample was determined.

Table III illustrates that running the process on more complex applications provides energy savings that are less predictable—the relatively higher variance means that complex applications yield optimal architectures that are less likely to provide energy savings for other, similar applications. A low variance of sensor network and general purpose applications means that an architecture found to be optimal for one application is likely to provide energy savings for other applications within the same group.

Figure 6 presents the code size for the optimal architectures. We reiterate that these results are derived with an optimization function purely driven by energy considerations. Because of this, code blow-up can be relatively large for the optimal architectures—the total running time is reduced because commonly executed paths have been made shorter at the expense of the less commonly used ones, often leading to an increase in the total code size. Still, the range of program memory requirements is typical of most MSP430 configurations. As an example, the marginal dollar cost of moving from an MSP430 configuration with a 16KB program memory to one with 32KB is under 10% [Texas Instruments 2006].

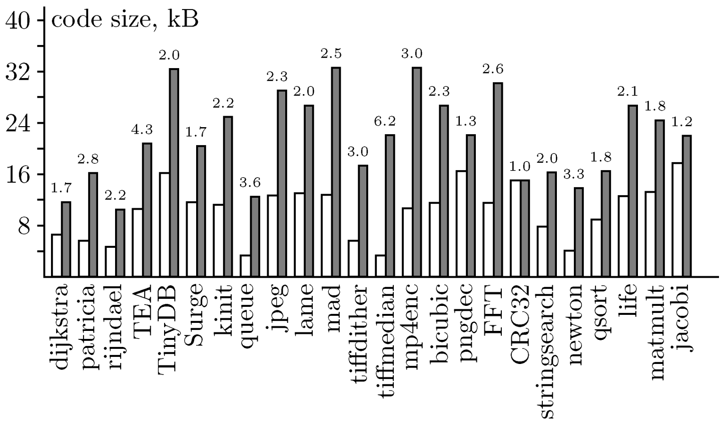


Fig. 6. **Code size relative to the MSP430.** For each benchmark, the initial code size (white bars) and the code size for the optimal architecture (gray bars) are reported.

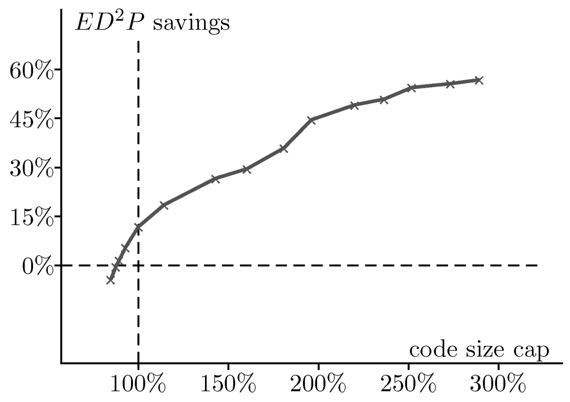


Fig. 7. $ED^2P$ **savings when the amount of code increase is capped.** As shown in Figure 6, the optimal architecture comes at a sacrifice of code size. When the average code size for generated applications is capped at a particular value (represented here as the percentage of the original average code size), the savings are smaller.

If one were area constrained, one could incorporate this constraint into the objective function. Figure 7 shows the $ED^2P$ savings, averaged over all benchmarks, when the code bloat is capped at a particular value. As shown in Figure 6, to achieve maximum $ED^2P$ savings, one must face a code increase to about 280% of its previous size. When this is capped at a lower value, the $ED^2P$ savings also decrease—at first gradually, and then fairly significantly. As a result, when the area cap is 160%, the $ED^2P$ savings are only half of their best values. Interestingly, when area is capped at 100% (meaning that the code size, on average, is not allowed to increase), we still maintain $ED^2P$ savings of about 10% relative to the base architecture. This is because for a majority of benchmarks, an implementation can be found that is more energy efficient but does not achieve the savings at a cost of bloated code. Only when the area

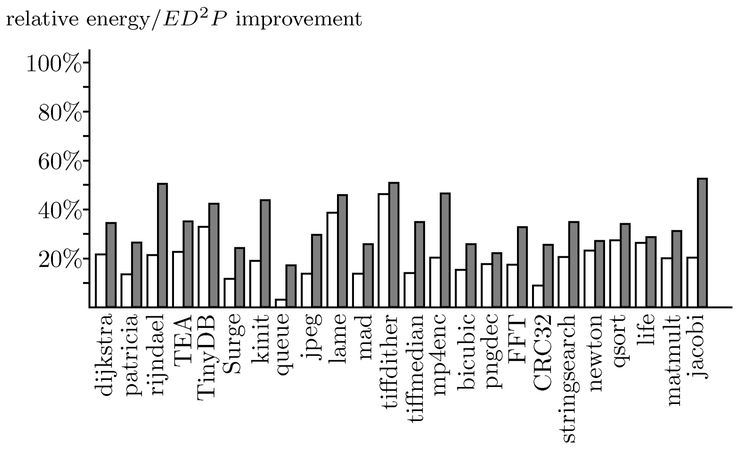


Fig. 8. **The code size increase when average case capped at 160%.** The code increase is given for an architecture whose code size increase is capped at 160%.

is capped at 80% do we begin to see $ED^2P$ losses. Figure 8 shows the itemized energy and $ED^2P$ savings when the area cap is 160% (i.e., when the savings are about half of their ideal values).

In reality, most high-volume computing architectures will need to execute more than one application. In such cases, architectures that may be optimal for one application will be far from it for another. The next section considers these situations.

## 4.3 Determining Architecture-Efficient Application Clusters

It is impractical to assume that we have a custom architecture for each application we run. However, it might be advantageous to identify clusters of applications that run efficiently on a particular architecture. This is particularly useful if the applications that a machine is executing share their characteristics with mostly one of the clusters. In such a case, a custom architecture will perform better for every application.

We first perform a similar analysis to one presented in Section 4.2. On a performance-power graph, we plot each architecture that maximizes the utility function ($ED^2P$) for a particular benchmark. We want to partition the set of benchmarks into "clusters" and assign an architecture to each cluster. Assuming that all benchmarks within a cluster will be executed on that assigned architecture, we want to maximize the total utility across all benchmarks. However, to penalize the creation of small clusters, we adjust the maximization function by including a term that increases with cluster size. In other words, our maximization function is of the form

$$\alpha \sum_{C_i} |C_i|^\tau + \beta \sum_{C_i} \sum_{B_{ij} \in C_i} U(A_i, B_{ij}) \tag{2}$$

where $C_i$ is the $i$-th cluster, $B_{ij}$ is a benchmark included in the $i$-th cluster, $A_i$ is

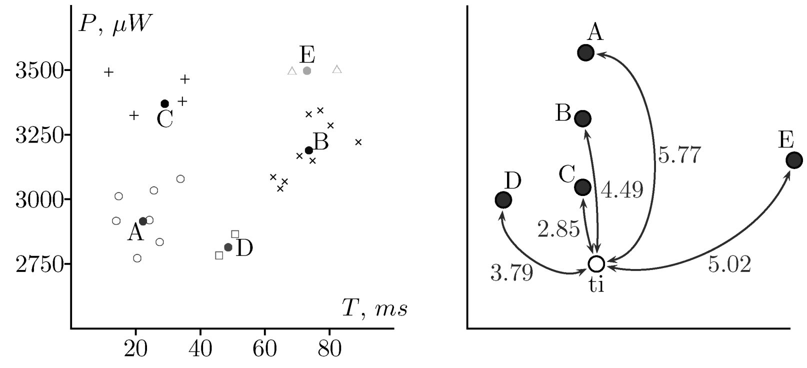


Fig. 9. **Benchmark clusters and optimal cluster architectures.** (a) For each benchmark, an architecture that maximizes the utility function is plotted on the performance-power graph. Benchmarks are combined in clusters and an optimal architecture for each cluster (also marked on the graph with a solid dot) is selected. The optimal cluster architecture executes all the benchmarks in sequence. Same-shape data points belong to the same cluster. Architectures $A$ through $E$ correspond to clusters $A$ through $E$ in Table IV. (b) A visualization of differences between optimal cluster architecture and MSP430—this is a synthetic graph where the axes carry no particular meaning, but the distances between all pairs of architectures are preserved. ti is the reference architecture, the MSP430. The distance can be thought of as the weighted number of optimization changes that need to be made to the MSP430 in order to arrive at the given architecture.

the cluster architecture, and $U$ is the utility function. $\tau$, $\alpha$, and $\beta$ are determined empirically.

Figure 9(a) shows the optimal architecture for each benchmark, the partition of architectures into clusters, and the optimal architecture for each cluster. Table IV details the configurations for each of the cluster architectures. Note that while this architecture is not optimal for all the applications, it is the best compromise configuration to execute all the applications in one cluster on the machine.

It is interesting to note that some parameter values are rarely included in the efficient architectures for a cluster. For instance, `ARCH_ACCUMULATOR` or full-width immediate instructions are not well represented. It is tempting to preclude such values from the search. However, there are specific benchmarks that favor architectures whose parameters take those values—in such cases, the savings are significant. Hence, even if no cluster includes a particular parameter value, it is possible that for a different set of representative applications, the resulting clusters might include those less popular parameter values.

The configuration of the optimal architecture for each of the clusters can tell us something about the applications belonging to the cluster. For instance, applications that use implicit stack (e.g., portions of `queue` and `surge`), should take advantage of a stack architecture—what takes two instructions (performing a load, and decrementing the pointer) can be compressed to one instruction. Moreover, the architecture for cluster $E$ uses eight registers and an accumulator—`life`, for example, does not use much parallelism and does not require a large amount of local immediate memory. Finally, `stringsearch` rarely requires immediate operands (or, if it does, they are usually small numbers or

Table IV. Optimal Architectures for Each Benchmark Cluster

| Cluster | Sample Benchmarks | Savings | | |
|---|---|---|---|---|
| | | Performance | Energy | $ED^2P$ |
| *A* | rijndael, TEA, lame, tiffmedian, ... | 14.5% | 17.4% | 39.6% |
| *B* | dijkstra, bicubic, fft, jpeg, ... | 11.7% | 13.1% | 32.2% |
| *C* | TinyDB, stringsearch, qsort, matmult | 17.2% | 19.3% | 44.7% |
| *D* | queue, surge | 11.5% | 18.4% | 36.1% |
| *E* | kinit, life | 8.8% | 13.2% | 27.8% |

| Cl. | Optimal Architecture | | | | $d(A)$ | $d(B)$ | $d(C)$ | $d(D)$ | $d(E)$ | $d(ti)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| | regs | ARCH_ | COMPLEX_ | IMM_ | | | | | | |
| *A* | 16 | OFFSET | DIVIDE | HALF | — | 2.28 | 4.16 | 5.52 | 5.95 | 5.77 |
| *B* | 32 | OFFSET | MUL | HALF | 2.28 | — | 2.44 | 4.16 | 5.16 | 4.49 |
| *C* | 32 | REGOFS | SHIFT | NONE | 4.16 | 2.44 | — | 3.31 | 5.11 | 2.85 |
| *D* | 16 | STACK | NONE | NONE | 5.52 | 4.16 | 3.31 | — | 2.24 | 3.79 |
| *E* | 8 | ACC | NONE | HALF | 5.95 | 5.16 | 5.11 | 2.24 | — | 5.02 |

For each of the five benchmark clusters, an optimal architecture is determined and reported, together with the performance benefits and energy and $ED^2P$ savings. The savings are reported as a fraction of original architecture (MSP430). $d()$ represents the weighted Euclidean distance between two architectures. $d(ti)$ is the distance between each architecture and the MSP430. The optimal architectures for each cluster feature two source registers.

powers of 2), so it is best implemented with an architecture with the IMM_NONE setting.

While Figure 9(a) offers a convenient visualization of the optimal architectures on the performance-power graph, we would also like to compare the cluster architectures against the original architecture (the MSP430) and see how similar or dissimilar they are. We define a weighted Euclidean distance $E(A_1, A_2)$ between two architectures as

$$E(A_1, A_2) = \sqrt{\sum_{i=1}^{5} w_i \cdot (C_i(A_1) - C_i(A_2))^2} \qquad (3)$$

where $C_i(A)$ is the value of the $i$-th parameter in the configuration of architecture $A$. We can think of this distance as a number of configuration optimizations necessary to move between architecture $A_1$ and architecture $A_2$, weighted by some constants $w_i$.

The values of $w_i$ are the chip area savings determined for each optimization (just as we computed the energy savings reported in Section 4.2), averaged over all optimizations of one parameter and over all applications within one cluster. The motivation behind this is that some optimizations require more hardware than others, and when determining the distance between architectures, those differences must be taken into account. The weighted Euclidean distance can, therefore, be thought of as the number of optimizations that distinguish a particular architecture from the MSP430.

Plotting the architectures would require a five-dimensional graph, so as a simplification, the architectures have been plotted on a synthetic diagram such that the Euclidean (planar) distances between every pair architectures have been (approximately) preserved. Figure 9(b) shows such a diagram. Note that the axes have no meaning in such a diagram, only distances between the data points. Note that all applications in a cluster have very close execution times. This is due to the fact that a cluster is determined using $ED^2P$ as a metric, which weighs performance much more significantly than consumed energy. Hence we should expect applications within a cluster to have similar (normalized) performance times.

We see from the savings in Table IV that the MSP430 is not an optimal architecture for any of the clusters. However, Figure 9(b) shows that each of the cluster architectures is different, and that some architectures are more similar to MSP430 (and one another) than others. For instance, the architecture for cluster $C$ shares more similarities with MSP430 than architecture $A$. Similarly, architecture for cluster $E$ is different than most other architectures.

## 4.4 Determining Optimal k-core Architectures

We see that while clusters give us more flexibility (we no longer require a separate architecture for each application we run), they also offer slimmer energy savings. This motivates us to examine the relationship between the savings and the number of heterogeneous cores provided in the microprocessor. In this context, we are exploring a class of heterogeneous ISA processors designs [Kahle et al. 2005]. Our approach is distinct from both traditional chip-multiprocessors that execute threads in parallel to increase performance, and Kumar et al. [2003], who proposes single-ISA heterogeneous multicores. Instead, a profiling infrastructure (similar to what we have used in this study) would choose the energy-optimal core for a particular application given and make necessary assembly modifications to run the application on this core. While this application runs, all other datapath cores would be put into a nonstate preserving sleep mode. We assume that the data and instruction memory are shared across all heterogeneous cores; since only one core is active at any time, this does not require significant design complexity.

So far, we identified two extreme cases of architectures—every benchmark has its own architecture, or there is one architecture—in this case MSP430, and we showed how the idea of clustering can yield a solution in between those two cases. Now let us consider another constraint—let us limit the degree of heterogeneity to application classes.

Figure 10 shows the average $ED^2P$ savings when we are allowed to optimally choose $n$ architectures for the eight benchmarks in each application class. When $n = 1$, we must choose one architecture for all eight benchmarks—the savings will naturally be lowest. When $n = 8$, the case reduces to that examined in Figure 5, and each, application has its own energy-optimal core. We see that after about four architectures, any additional architectures present decreasing marginal benefits. This inflection point might mean that when contrasted with the costs of extra processors, there is an optimal number of processors that ought to be used.

This means that a multicore system where exactly one core is allowed to be turned on will do significantly better than a single-core system. The reason for this is that if one is constrained to pick one core for all applications within a

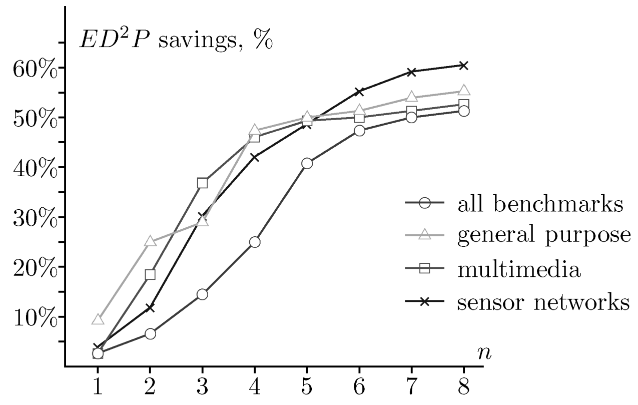


Fig. 10. **Savings (in terms of $ED^2P$) as a function of the number of processors we are allowed to include in the machine.** The architectures are found using a similar maximization problem as one described in Section 4.3. Every data point denotes one additional architecture. Note that for individual application classes, the savings are close to optimal after the number of architectures reaches 5, and when the partition is unrestricted ("clustering" is a special case of this), 7 architectures and more achieve satisfactory savings.

class, the variability in the applications' profiles will translate into high variance of energy and power consumed by the selected architecture. If we provide the microcontroller with a choice of a core to use, the best metric will be used for each application and the high variance will average out.

Note that if all benchmarks are used (i.e., if applications are not separated into classes), the savings are relatively lower than for each individual application class. This makes sense—the commonalities between applications in each application class mean that the optimal architectures are better catered to the common use case, while if we consider all applications, the commonalities disappear. More generally, if we know how similar (or dissimilar) our applications will be, we can determine a priori the number of cores that achieve given energy savings. Figure 11 presents the $ED^2P$ savings as a function of how similar the applications in a set are, and how many cores are included. The coefficient $h$ is a measure of homogeneity of the applications (i.e., the probability that two randomly selected applications will belong to the same application class). Higher values of $h$ indicate that the applications are more similar to one another. The special cases $h = 1.00$ and $h = 0.00$ are illustrated in Figure 12, as the average of the three specific class savings and the savings for all benchmarks, respectively.

The problem with heterogeneous ISA processors, however, is that extra cores require hardware. This cost can be evaluated quantitatively. Assuming that the memory is shared across all cores, we can determine the total (including memory) chip area as a function of $n$. Figure 12 shows this relationship. Since the memory is shared, the area of the chip increases only slightly for each additional core. With four or five additional cores, the total chip area is only 50% greater than the area of a one-core chip. Hence, the energy and performance benefits likely outweigh the cost of four to five cores.

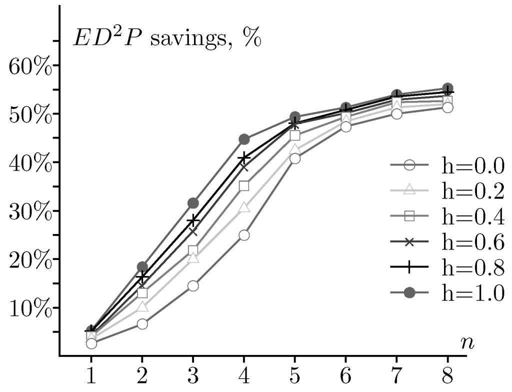


Fig. 11. **Savings in percent as a function of the number of cores and the homogeneity of the applications.** Homogeneity $h$ of the applications is defined as the probability that two randomly selected applications will belong to the same class.

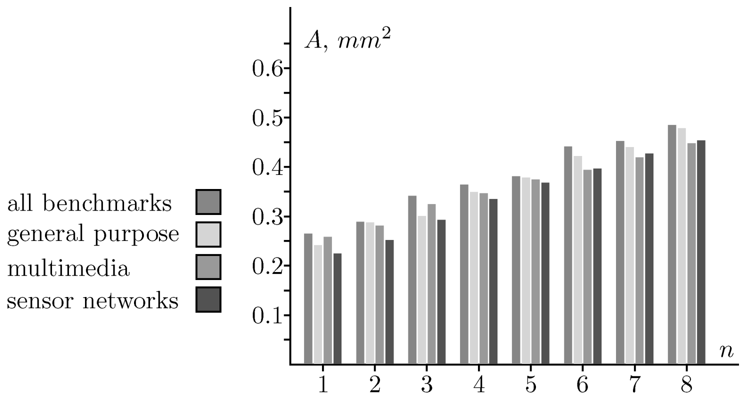


Fig. 12. **Total chip area as a function of the number of processors we are allowed to include in the machine.** Assuming that the memory is shared, each additional core increases the area by a small amount.

## 4.5 Determining Process Validity

Finally, we devise an experiment to justify using assembly-to-assembly translations instead of compiler retargeting. Since the process uses assembly-to-assembly translation as a proxy for actual source translation, it is important to ensure that it introduces no bias in the determination of an optimal architecture. We show that regardless of which architecture the application is originally compiled for, the resulting family of optimal architectures is the same (or nearly the same). By looking at the difference between families, we also suggest that no systematic bias is introduced as a result of the assembly-to-assembly translation.

In this experiment, the Pareto families are determined twice, once for the benchmarks compiled for MIPS, and once again for the same benchmarks compiled for MSP430. Ideally, we would expect the Pareto families to be identical in both cases. Figure 13 summarizes by how much the two families differed, as

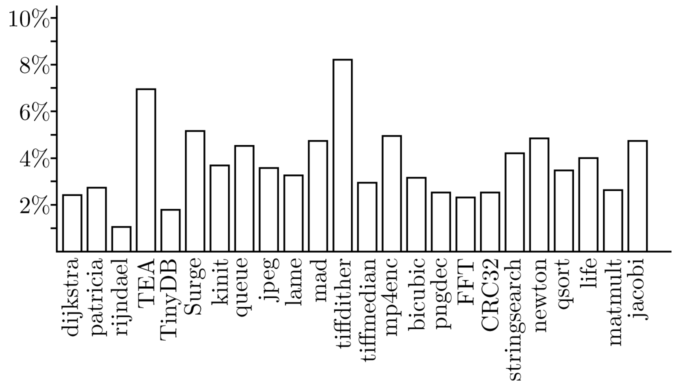


Fig. 13. **The differences in Pareto families for two different targets.** Each benchmark has been compiled for MIPS and again for MSP430. The Pareto optimal families are determined for each case. The height of the bar denotes the fraction of the entire set of architectures that differed between the two cases (i.e., the measure $1 - |\mathscr{F}_1 \cap \mathscr{F}_2| / |\mathscr{F}_1|$)—the ratio of the size of the symmetric difference of the two sets to the average size of the sets.

a fraction of the size of the family. It is clear that the families are nearly identical, with the largest difference close to 8% of the entire set, and most families differing by less than 4%.

Finally, we performed full synthesis of selected architectures and compared them to the full synthesis of the original architecture. The energy, power, and chip area estimates provided by Synopsys were within 7% of our model's estimates.

## 4.6 Further Optimizations

Some optimizations are possible that decrease the running time of the process while still yielding optimal architectures. It is possible to prune the search space of all architectures by evaluating a set of simple heuristics on the profile of the application. Moreover, one can use the energy, execution time, and area estimates provided by Synopsys as a good proxy for the more accurate data from SoC Encounter.

The motivating idea behind using Synopsys as a proxy for the chip layout data (instead of Place and Route) is that the metrics provided by Place and Route (chip area, energy, running time) are needed only for the ordering of all architectures from most to least efficient (the actual value of the metric is not very helpful until the user supplies his utility curve). If the ordering given by SoC Encounter is similar to the ordering given by Synopsys, we can use Synopsys data as a proxy for the metric values. Ideally, if the two orderings are identical, then the resulting Pareto optimal curves will also be identical. More specifically, the Pareto curve is preserved under monotonic metric transformations.

Finally, we noted that the exhaustive search, even if it included only several hundred architectures, was not a particularly efficient way to determine efficient architectures. We devised a number of optimizations, which reduced the running time of the process significantly while providing optimal or near-optimal results.

It is clear that some architectures will definitely not lie on the Pareto optimal curve even before running the simulation. For instance, a program that includes hundreds of multiplications and divisions had better be implemented in an architecture that includes the division and multiplication instructions. Similarly, a program that features a large number of indirect loads should be implemented in an architecture that allows indirect loads. In such cases, the search module can prune the search space initially by simulating the original program in its original architecture and looking for hints that would help it fix any parameters. Fixing one parameter reduces the search space by a factor of between 2 and 4. This reduces the running time of the process by about the same factor. We developed heuristics linking the frequency of certain operations with a particular configuration option. For example, the inclusion of the multiplication unit is forced if a "large" number of multiplies is used. The same applies to divides, shifts by nonconstant amounts, nonsequential memory accesses, and immediate instructions. Those heuristics prune the search space by a factor of 2.6 while still producing families that overlap by over 97%.

## 5. RELATED WORK

The question of customizing the architectures to particular applications is not a new one, and existing solutions can be divided into three categories: those leveraging FPGAs, those proposing full-custom solutions, and those generating a specialized architecture from a model.

The latter is particularly relevant to this article. For example, Nazhandali et al. [2005] attempt to optimize the energy usage of sensor network processors that all share the same ISA, but differ in the data path width, the memory architecture (Harvard versus Von Neumann), and the supported addressing modes. Similarly, Thumb [ARM Corporation] provides some features of an application-specific ISA by varying bit-width [Krishnaswamy and Gupta 2003]. Moreover, Cheng and Tyson [2005] propose to replace fixed instruction and register decoding hardware in embedded CPUs with support for programmable decoders allowing tailoring of the instruction set to reduce code size and instruction cache energy. Additionally, Fisher et al. [2000] propose an entire technology platform where the instruction issue width and the various arithmetic units have been parameterized.

The PICO project [Kathail et al. 2002] is worth pointing out as it is similar to this article in the goals and the approach. However, there are some important differences between the two:

—PICO focuses mainly on VLIW architectures [Aditya et al. 1999], hence the parameter space that the authors look at is different from the space of our architectures, except the register file size and the type of functional units.

—PICO does not seem to be integrated into the VHDL-synthesis flow as tightly as our process. Specifically, rather than include power or cycle time information in their analysis, PICO focuses mainly on chip area.

—PICO's focus is not application clustering or heterogenous core analysis, which is a major motivation of our article. However, PICO's work is based on a superset ISA, of which this work is a generalization.

Similarly, Sheldon et al. [2006] constrains itself to the FPGA space and does not consider heterogeneous multicore processors. Finally Kumar et al. [2006] is similar to this article in that it considers heterogeneous multicore systems but it focuses primarily on single-ISA systems.

Related work exists that focuses on optimizing the search for an optimal architecture. Spacewalker [Snider], for example, extends the work of Kathail et al. [2002] but it also focuses on VLIW architectures. The article's main contribution are the characteristics of the "walk"—the search for an optimal architecture, rather than the choice of the model parameters or the multicore systems. Sherwood et al. [2004] and Padmanabhan et al. [2006] use linear programming to define the search more rigorously.

Furthermore, Fisher et al. [1996] design a system that uses an efficient compiler to generate code for a customizable architecture and shows a great variation in the running time and chip area. While closest in its objective to this article, Fisher et al. [1996] constrain itself primarily to image processing algorithms to show the drawbacks to specialized hardware. Finally, Clark and Tang [2002] attempt to customize instruction sets of mobile and embedded applications using retargetable compilers: If the source code of an application is known, it suffices to modify the compiler to allow a variation in the target architectures.

Two additional article complement our work. Clark et al. [2005] focus on designing custom instructions and seamlessly integrating it into existing instruction sets. Those instructions are supported by hardware accelerators and increase the performance of microcontrollers that take advantage of the added custom instructions. This work can easily be used to extend the results of this article. After an efficient instruction set is determined, hardware-accelerated custom instructions can be attached on top of the base architecture, yielding even higher energy savings or performance gains.

Chen et al. [1993] suggest doing optimizations using profile-based transformations of the program. Two categories of profile information (control flow and memory dependence) are identified and used to reduce the execution time. The findings of Chen et al. [1993] could be used to improve the transformations we introduced in Section 3.2.

The contributions of Nazhandali et al. [2005] and Fisher et al. [2000] are important, but compared to those findings, we:

—consider a wide range of applications, with workloads varying from sensor network to multimedia and general purpose ones.

—look at a wider rang e of architectural issues, such as architecture type (Load-store versus Stack versus Accumulator), the number of registers, and path length, found to be critical in the performance analysis of the architectures.

—find that heterogeneous ISA processors offer sizeable energy and performance savings.

Finally, Tensilica allows the programmer to extend the ISA through custom instructions. Commonly executed instructions are grouped in clusters, and new hybrid instructions are added to the instruction set. This article differs from Tensilica by allowing for simplification of the baseline architecture, as well as varying fundamental structures of the architecture (such as switching between stack and accumulator architecture). Moreover, we find compromise architectures by clustering applications and quantifying the energy-efficiency benefit of heterogeneous processors for these embedded applications.

In terms of research on heterogeneous multiprocessors, previous work primarily focused on very high-performance designs and primarily same-ISA heterogeneous processors [Kumar et al. 2003]. We show that it is possible to apply principles of heterogeneity to embedded architectures with varying ISAs.

## 6. CONCLUSIONS AND FUTURE WORK

This article presents a simple and efficient way to provide energy and time savings by customizing the architecture on which a particular application should be run. By determining a Pareto optimal family of architectures, the designer has a choice of multiple architectures that execute an application optimally. Automating this design process can reduce design effort drastically. We show that the energy savings, when compared with existing microcontrollers, such as MIPS or TI MSP 430, are substantial. The relative energy savings, broken down into optimization classes, can help designers better fine-tune their custom designs.

We also present a way of determining clusters of applications that execute efficiently on similar architectures. It often happens that most applications deployed to a sensor network belong to one cluster (e.g., they are all network routing algorithms). In such cases, replacing the off-the-shelf microcontroller with a custom-made one can provide significant energy savings. We attempt to explain those differences by analyzing the applications themselves: Certain classes of applications tend to prefer particular classes of architectures.

Most importantly, if the application clusters cannot be determined (if the applications are not known a priori), we find that even for diverse application classes, it is possible to find a small number of architectures that together achieve energy and performance savings within 15% of the optimal. Together with the fact that extra cores cost little (compared with the original core and memory subsystem), this article points out that heterogeneous ISA processors can be effectively used in embedded systems.

We show that there is a clear trade-off between energy consumption and chip area: When the architectures are area-constrained, the total energy savings are reduced from their optimal values. The savings are nonlinear: As the savings converge to their optimal values, the code bloat increases significantly. Interestingly, if the code of the translated applications is not allowed to grow in the average, we can still achieve energy savings over the base architecture.

One area for improvement lies in the Verilog model implementation. The model should be further optimized and pipelined. While it is currently possible to tell how many clock cycles a pipelined design would take to execute a particular program, without actually pipelining it, nothing conclusive can be said about the optimal model. Specifically, pipelining introduces overhead (due to the interlocks and extra routing logic), which is difficult to estimate.

Relaxing the design of the register file can also yield interesting results: Mixed-width registers or banked register files may provide better-performing architectures. However, doing this is no easy feat: Variations in register file design yield a much more complicated space, which quickly becomes difficult to do searches on.

It is important to note that while the configurations found in this article are more efficient than the MSP430, the analysis presented here requires a priori knowledge of the application workloads and different workloads might yield different architecture. As a general purpose microprocessor, the MSP430 makes a good compromise solution. However, significant energy and performance benefits can be gained by exploiting the application variations through customized microcontrollers and heterogeneous ISA processor systems. Therefore, we propose that heterogeneous processors can be used to take advantage of this in future embedded systems.

## ACKNOWLEDGMENTS

Thanks to Glenn Holloway, Mark Hempstead, Gu-Yeon Wei, and the anonymous reviewers for their invaluable comments.

## REFERENCES

Aditya, S., Rau, B. R., and Kathail, V. 1999. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings of the 12th International Symposium on System Synthesis (ISSS'99)*. IEEE, Los Alamitos, CA107.

ARM Corporation. Thumb ISA. http://www.arm.com/products/CPUs/ARM7TDMI.html.

Babb, J., Frank, M., Lee, V., Waingold, E., Barua, R., Taylor, M., Kim, J., Devabhaktuni, S., and Agarwal, A. 1997. The raw benchmark suite: computation structures for general purpose computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA, 161–171.

Bergamaschi, R. A., O'Connor, R. A., Stok, L., Moricz, M. Z., Prakash, S., Kuehlmann, A., and Rao, D. S. 1995. High-level synthesis in an industrial environment. *IBM J. Res. Dev. 39*, 1–2, 131–148.

Brooks, D., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J., Zyuban, V., Gupta, M., and Cook, P. W. 2000. Power-aware microarchitecture: design and modeling challenges for the next generation microprocessors. *IEEE Micro 20*, 6, 26–44.

Budd, G. and Milne, G. 1996. ARM7100—a high integration, low power microcontroller for pda applications. In *Proceedings of the 41st IEEE Computer Conference*. IEEE, Los Alamitos, CA, 182.

Cadence Incorporate. SoC Encounter. http://www.cadence.com/products/digital_ic/soc_encounter/.

Chen, W. Y., Mahlke, S., Warter, N., and Hank, R. 1993. Using profile information to assist advanced compiler optimization and scheduling. In *Advances in Languages and Compilers for Parallel Processing*, U. Banerjee et al Ed.

Cheng, A. C. and Tyson, G. S. 2005. An energy efficient instruction set synthesis framework for low power embedded system designs. *IEEE Trans. Comput. 54*, 6, 698–713.

Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., and Flautner, K. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, 272–283.

Clark, N. and Tang, W. 2002. Automatically generating custom instruction set extensions. In *Proceedings of the 1st Workshop on Application-Specific Processors*. ACM, New York, 94–101.

Fisher, J. A., Faraboschi, P., and Desoli, G. 1996. Custom-fit processors: Letting applications define architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO-29)*. IEEE, Los Alamitos, CA, 324–335.

Fisher, J. A., Homewood, F., Brown, G., Desoli, G., and Faraboschi, P. 2000. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA'00)*. IEEE, Los Alamitos, CA, 203–213.

FOK, C.-L., ROMAN, G.-C., AND LU, C. 2005. Mobile agent middleware for sensor networks: an application case study. In *Proceedings of the International Symposium on Information Processing in Sensor Networks*. ACM, New York.

FUDENBERG, D. AND TIROLE, J. 1983. *Game Theory*. MIT Press, Cambridge, MA.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*. IEEE, Los Alamitos, CA, 3–14.

HILL, J. L. 2003. System architecture for wireless sensor networks. Ph.D. thesis, University of California, Berkeley.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*. IEEE, Los Alamitos, CA, 1098–1102.

KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Dev. 49*, 4/5, 589–604.

KATHAIL, V., ADITYA, S., SCHREIBER, R., RAU, B. R., CRONQUIST, D. C., AND SIVARAMAN, M. 2002. Pico: automatically designing custom computers. *IEEE Comput. 35,* 9, 39–47.

KRISHNASWAMY, A. AND GUPTA, R. 2003. Mixed-width instruction sets. *Comm. ACM 46*, 8, 47–52.

KUMAR, R., TULLSEN, D., AND JOUPPI, N. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. IEEE, Los Alamitos, CA, 23–32.

KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. 2003. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-36).* IEEE, Los Alamitos, CA, 81–92.

MIPS TECHNOLOGIES. Mips 4k processor core family user's manual. http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorCores/4KFamily/MD00016-2B-4K-SUM-01.17.pdf.

NAZHANDALI, L., ZHAI, B., OLSON, J., REEVES, A., MINUTH, M., HELFAND, R., PANT, S., AUSTIN, T., AND BLAAUW, D. 2005. Energy optimization of subthreshold-voltage sensor network processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05).* IEEE, Los Alamitos, CA, 197–207.

PADMANABHAN, S., CYTRON, R. K., CHAMBERLAIN, R. D., AND LOCKWOOD, J. W. 2006. Automatic application-specific microarchitecture reconfiguration. In *Proceedings of the Reconfigurable Architectures Workshop (RAW'06).* IEEE, Los Alamitos, CA.

POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 21*, 5, 895–913.

SHELDON, D., KUMAR, R., VAHID, F., LYSECKY, R., AND TULLSEN, D. 2006. Application-specific customization of parameterized FPGA soft-core processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'06)*. ACM, New York, 261–268.

SHERWOOD, T., OSKIN, M., AND CALDER, B. 2004. Balancing design options with Sherpa. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, 57–68.

SNIDER, G. 2001. Spacewalker: Automated design space exploration for embedded computer systems. Tech. rep.t HPL-2001-220, HP Laboratories, Palo Alto, CA.

TENSILICA. Xtensa LX Processor. http://www.tensilica.com/products/xtensa_LX.htm.

TEXAS INSTRUMENTS. 2006. TI MSP430 user guide. http://www.ti.com/litv/pdf/slau049f.

UMC FARADAY. UMC Faraday 0.13$\mu$m libraries. http://freelibrary.faraday-tech.com/ips/013library.html.

WERNER-ALLEN, G., JOHNSON, J., RUIZ, M., LEES, J., AND WELSH, M. 2005. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the 2nd European Workshop on Sensor Networks.* IEEE, Los Alamitos, CA, 108–120.

Received March 2007; revised April 2008, September 2008; accepted November 2008