Efficient Architectures through Application Clustering and Architectural Heterogeneity

Lukasz Strozek
Harvard University
strozek@eecs.harvard.edu

David Brooks
Harvard University
dbrooks@eecs.harvard.edu

ABSTRACT

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-constrained application domains such as sensor node and multimedia processing. We implement a process flow that analyzes runtime profiles of applications and combines this information with a model for our architectural design space providing a robust customization engine built upon a fully automated method for determining an efficient architecture (together with appropriate application transformations). By profiling embedded benchmarks from a variety of sensor and multimedia applications, the paper shows the relative energy savings resulting from various architectural optimizations and identifies the number of architectures that achieves near-optimal savings for a group of applications. This paper proposes the use of heterogeneous chip-multiprocessors as a cost-effective approach to capitalize on the potential energy savings provided by application customization while executing a range of applications efficiently.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Design

Keywords

Efficient Custom Architectures, Heterogeneous CMP

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 [14] that are inspired by a variety of existing applications.

Trying to perform well all the time, however, means certain trade-offs. For any architecture, there exist programs for which the machine performance is sub-optimal, and, as architectures become more and more complex, the chance that some other, more specialized architecture outperforms in power-performance efficiency increases. This is not a problem if the architecture is intended to be multi-purpose: 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 certain sensors and are usually deployed with a specific task in mind [9]: they function as a fire detection system that is independent of the main building infrastructure [7], or around volcanoes, measuring seismic activity [18]. 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.

It is well-known that custom hardware can be built to perform a task with significant gains in efficiency. However, designing custom ASICs or VLSI microcontrollers takes time and human capital. For this reason most motes use an off-the-shelf microcontroller (for instance, the TI MSP 430 [16]) which might perform sub-optimally for a given application, but drastically reduces deployment costs and time spent designing the hardware.

This paper proposes a solution that offers similar benefits with a fraction of the cost: it finds an optimal architecture (or a set of architectures comprising a heterogeneous multiprocessor) 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 proposed system determines the optimal architectures for a given application, creates a hardware description language (HDL) model for them, then synthesizes the model to output a chip lay-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'06, October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

out design ready to be fabricated. This paper focuses on the analysis of microcontrollers of similar complexity and market focus as the TI MSP430, MIPS R2000, or ARM7: simple instruction sets implemented on unpipelined or minimally pipelined microarchitectures that are increasingly popular for energy-constrained sensor and media workloads.

The overarching goal of the paper is threefold:

- First, this paper develops a process which takes a run-time profile of an application, synthesizes families of efficient architectures, and performs efficient code transformations to run on these architectures.
- It then relaxes the single-application requirement and partitions 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.
- Finally, given a partition, it determines the optimal number of architectures (comprising the cores in a heterogeneous CMP) by trading off energy benefits against hardware costs.

The rest of the paper 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 paper, and proposes future work. Finally, Section 6 concludes the paper.

2. PROCESS FLOW

The generation of an efficient microprocessor for a particular application can be seen as a search problem. To reduce the search space, we create a *model* encapsulating any microprocessor in just several parameters. Since such a model limits the search space drastically (in our case, to some 3840 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 optimal architecture, the process will therefore accept a user program as input, and determine a set of parameters which yield the most optimal architecture. Two factors will affect the optimal decision:

- The profile of any candidate architecture, determined independently of the user program provided. Since this profile is generated off-line, and only once for every architecture embraced by the model, we will simply look up the pre-computed 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 a different instruction set, it is necessary to convert the user program to a program with identical functionality, but written using a particular instruction set.

Those two sources will 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 will find a *Pareto optimal* family of architectures. To account for possible measurement errors, we will allow the values of the metrics to vary by 5%.

The implementation of the model is a Verilog file which 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 [17] 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.

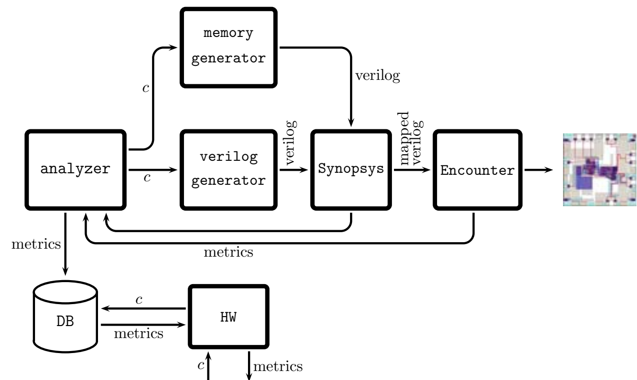

Figure 1: The offline part of the process flow
For each configuration c, the analyzer generates the Verilog model, gathers the metrics (the worst-case delay of the circuit, the chip area, and the static and dynamic power consumption) from Synopsys Design Analyzer and SoC Encounter, and saves them in a database.

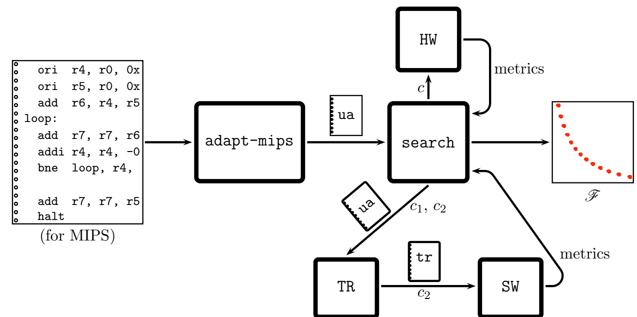

Figure 2: The online part of the process flow
Given a (for example) MIPS assembly file, the adapter first converts it to a file compatible with the universal assembly. For each configuration to test c_2, the search module passes the file to the translator, and then the software profiler. Given the metrics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

Consider the process flow shown in Figures 1 and 2, consisting of two parts: off-line and on-line analysis. Off-line analysis generates the architecture-specific data that can be stored in the database (this data is independent of the appli-

cation being analyzed). First, given a particular set of model parameters, a Verilog model is generated. This model is analyzed by Synopsys Design Analyzer, which synthesizes it and converts 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. 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 [17]. Once Design Analyzer generates the mapped design, Cadence SoC Encounter [3] 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.

On-line 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 (hence, the architecture model should be as expressive as possible to include many existing architectures), but the instruction set can be arbitrary. However, it should be relatively easy to convert that input program to a program that is written in the assembly defined in this paper (called universal assembly) with instructions from the instruction set defined in this paper. 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 users supplies the search module with a few model parameters that must be held fixed. Those parameters, called *irreducible*, will 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 supported by any arbitrary architecture. If it is not supported, it needs to be *translated* (*reduced*) into a program that is. 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 proposed architecture model ensures that a lossless translation is always possible: that is, for any configuration of the *reducible* parameters (all parameters which 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.

2.1 Discussion of Alternative Solutions

The process presented in this paper performs assembly-to-assembly translation of user programs. While such translation results in programs which are correct (programs which 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 different 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 which 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 of all, 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 re-allocation algorithm has been inspired by solutions seen in open-source compilers. Finally, the results presented in this paper are conservative: the paper finds 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 paper 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 with emphasis on register-register addressing. 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 which require a greater number of cycles (for example, the stack architecture might need to address the data memory up to four times).

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 1. The bottom two parameters are irreducible. One should note that the irreducible parameters are fixed throughout the process – the user decides at the beginning of the process what values to set them to. The search only includes reducible parameters, so that the total number of distinct architectures considered is 480.

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.

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).

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

3.2 Instruction Set Architecture and Translations

The instruction set defined in this paper 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 re-allocation algorithm featuring a linear scan algorithm [13]. First, all registers are converted into unique memory-holders (this step is the converse of register allocation – registers are converted into variables). Then, the variables are given their new allocations. Registers `r0` through `r4` are not touched (which allows for jumps to variable addresses such as `jr`); `r5` and `r6` are reserved to handle spill – if a variable cannot be given a register number in the range, it is put in the low address in memory. When the register is used as source, `r5` is used as a temporary register to which the value is loaded from memory.
- 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 which 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 a shift and a nor. This is, obviously, very costly if a large number of immediates are used, but significantly reduces instruction size, and thus the size of instruction memory.
- Finally, converting simpler architectures to more complex architectures requires certain program optimizations. For instance, if a multiplier is added to the architecture, what has previously been achieved with a routine can now be replaced with a simple instruction. The translator locates such instances and simplifies the code by collapsing multiple instructions.

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 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) which 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: thirty-two registers, Register-offset Load-Store architecture type, a shifter, multiplier and a divider included, three source registers included, instructions with half-width immediate values included, 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

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

Table 1: Model parameters and their values
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.

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: sixteen registers, Indirect Load-store architecture type, no complex arithmetic circuits, no three source registers, instructions with half-width immediate values included, 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 paper 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 the claim of this paper.

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, this paper picks an ED^2P utility metric and applies it to the family, thus determining the architecture which 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 maximized the total utility (in terms of ED^2P) of all applications in the cluster.

Finally, the paper attempts to determine whether using heterogeneous multi-cores is advantageous by looking at the relationship between the number of optimal architectures allowed for an entire class of benchmarks and the performance benefits. The paper finds 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 [8], a freely available embedded benchmark suite; RAW [2], a suite for general purpose computing; standalone applications and portions of the TinyOS kernel and user program code [9]. Table 2 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 paper use the MSP430 as a reference architecture, though for validation purposes, the benchmarks have also been compiled for MIPS. Three application classes are identified: sensor network applications, multimedia applications and general purpose applications.

To determine a Pareto curve, we use the following metrics: machine performance (measured in microseconds), *i.e.* the time it takes a particular machine to execute a particular benchmark (or part of a benchmark) and 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. 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

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 Algorithm
	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

Table 2: Benchmarks used
Four classes of applications are identified: sensor network, multimedia, general purpose. All benchmarks come from four sources: MiBench, TinyOS, RAW and standalone applications.

(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 will yield an upper bound on how much savings can be achieved through application-specific architectures. Figures 5 (a) through (c) 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 Pareto optimal family of architectures is determined with a 5% threshold (essentially, if changing any metric by 5% puts an architecture on the Pareto curve, this architecture is included in the resulting family). For reference, the architecture that corresponds to MSP430 is emphasized on the diagram.

For simplicity, we use the the ED^2P metric to constrain ourselves to one optimal architecture as opposed to the entire family of architectures. For that architecture, in each of the three examples, we offer a decomposition of the total energy savings by considering each architectural optimization, *i.e.* 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 this purpose, a list has been made of all configuration changes necessary to convert the MSP430 into 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.

In general, we find that the application path length dominates clock frequency in determining application performance. While power dissipation can fluctuate by 20% - 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 3. 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 75–85%.

Figure 4 presents the code size for the optimal architectures. We reiterate that these results are derived with an optimization function purely driven by energy considerations. While the energy cost of increased code size is modeled by including estimates for a larger instruction memory, we did not optimize our designs for chip area, although it would be trivial to include this in our optimization framework. Because of this, code blowup 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, which means that the total code size has increased. 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% [16].

In reality, most high-volume computing architectures will need to execute more than one application. In such cases architectures which may be optimal for one application will be far from optimal 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 which run

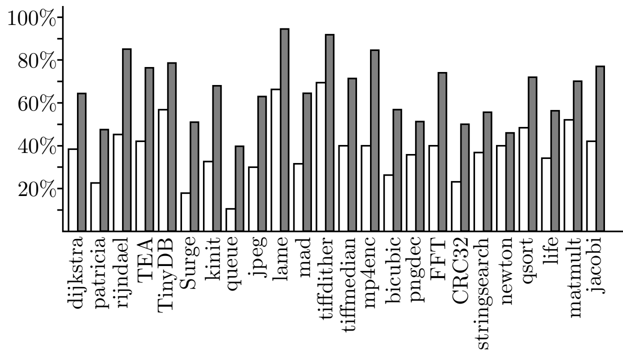

Figure 3: Energy and ED^2P savings relative to MSP430.
For each benchmark, an optimal architecture is determined and the energy (white bars) and ED^2P (gray bars) savings are identified.

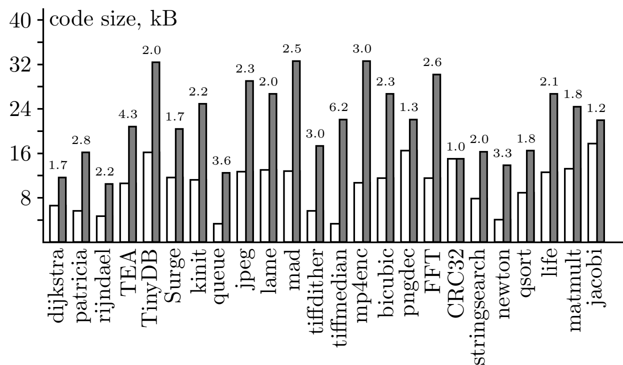

Figure 4: Code size relative to the MSP430.
For each benchmark, the initial code size (white bars) and the code size for the optimal architecture (gray bars) are reported.

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 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 which maximizes the utility function (ED^2P) for a particular benchmark. Given the set of all benchmarks, we want to partition them 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

$$\alpha \sum_{C_i} |C_i|^{\tau} + \beta \sum_{C_i} \sum_{B_{ij} \in C_i} U(A_i, B_{ij}) \qquad (1)$$

where C_i is the i-th cluster, B_{ij} is a benchmark included in the i-th cluster, A_i is the cluster architecture and U is the utility function. τ, α and β are determined empirically.

Figure 6(a) shows the optimal architecture for each benchmark, the partition of architectures into clusters, and the optimal architecture for each cluster. In addition to it, Table 3 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.

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 which 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 8 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 powers of two), and so it is best implemented with an architecture with the `IMM_NONE` setting.

While figure 6(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

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

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 weighed 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 6(b) shows such a diagram – it is important to note that the axes have no meaning in such a diagram, only distances between the data points.

We see from the savings in Table 3 that the MSP430 is not an optimal architecture for any of the clusters. However, figure 6(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, architecture for cluster C shares more similarities with MSP430 than architecture A. Similarly, architecture for cluster E is different than most other architectures.

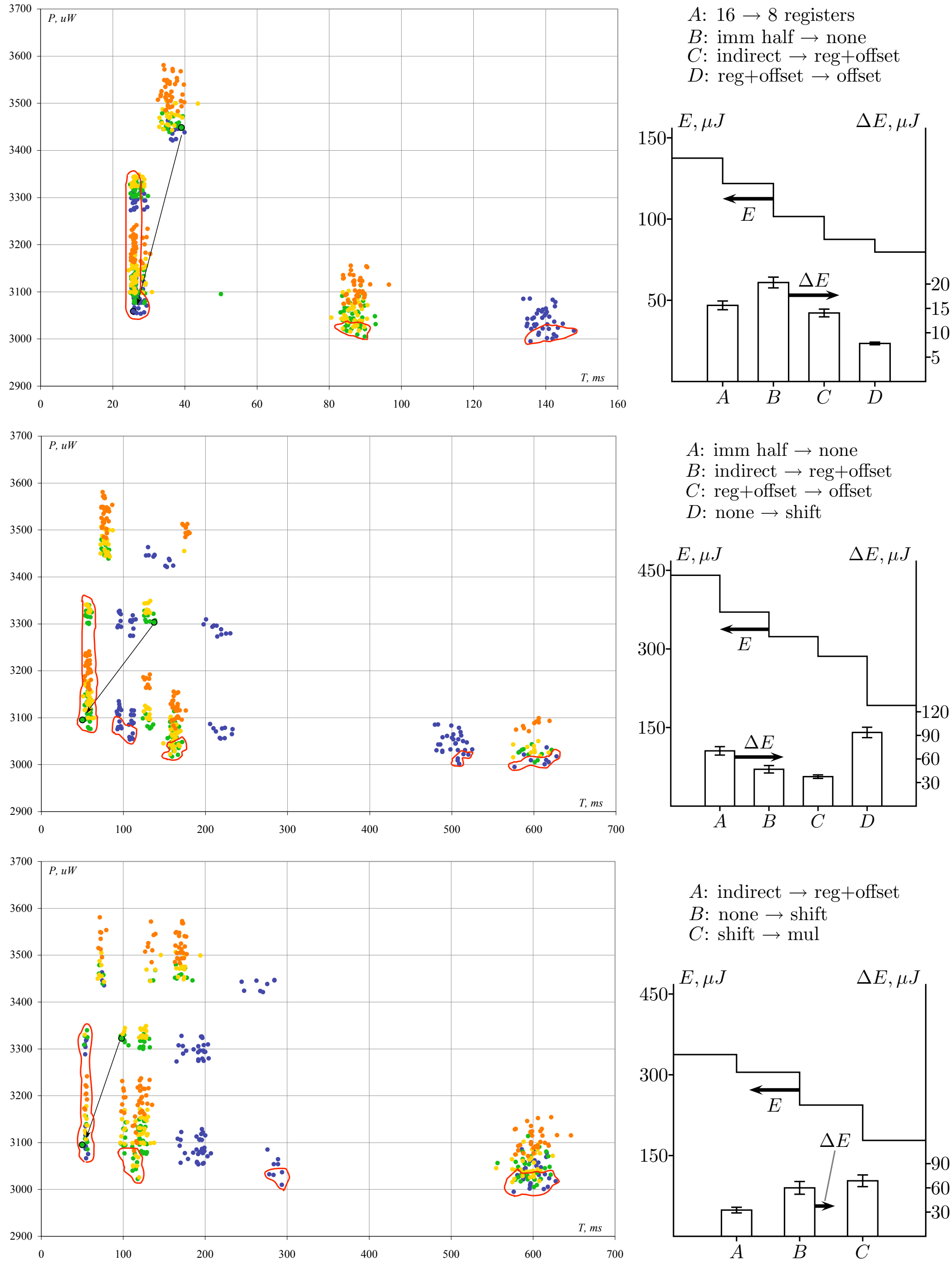

Figure 5: Pareto efficient families of architectures.
The figures on the left show all the architectures plotted on the performance-power graph for (a) fft, (b) lame and (c) rijndael. In this experiment, the running times of the benchmarks have not been normalized. Different colors for data points correspond to different number of registers. The circled data points are the Pareto efficient 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). The figures on the right show savings broken down by particular optimizations.

Cl.	Benchmarks	perf	E	ED^2P
A	rijdael, TEA, lame, mp4enc, tiffdither, tiffmedian, CRC32	14.5%	17.4%	39.6%
B	dijkstra, patricia, bicubic, jacobi, fft, newton, jpeg, pngdec, mad	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_	src regs	IMM_						
A	16	OFFSET	DIVIDE	2	HALF	–	2.28	4.16	5.52	5.95	5.77
B	32	OFFSET	MUL	2	HALF	2.28	–	2.44	4.16	5.16	4.49
C	32	REGOFS	SHIFT	2	NONE	4.16	2.44	–	3.31	5.11	2.85
D	16	STACK	NONE	2	NONE	5.52	4.16	3.31	–	2.24	3.79
E	8	ACC	NONE	2	HALF	5.95	5.16	5.11	2.24	–	5.02

Table 3: Optimal architectures for each benchmark class.
For each of the three benchmark classes, 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.

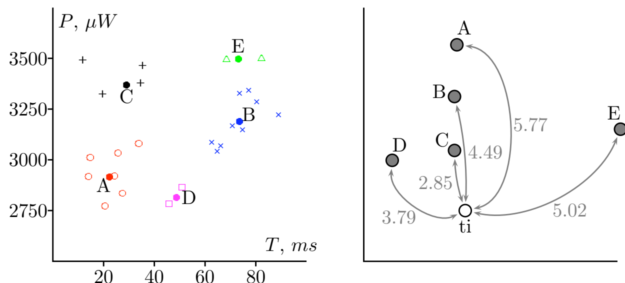

Figure 6: Benchmark clusters and optimal cluster architectures.
(a) For each benchmark, an architecture which maximizes the utility function is plotted on the performance-power graph. Benchmarks are combined in clusters and an optimal architecture for each cluster (also marked on the graph with a solid 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 3. (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 weighed number of optimization changes that need to be made to the MSP430 in order to arrive at the given architecture.

4.4 Determining within-class Optimal 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 chip multiprocessor* designs [11]. Our approach is distinct from traditional chip-multiprocessors which execute threads in parallel to increase performance. Instead, a profiling infrastructure (similar to what we have used in this study) would choose the energy-optimal core for a particular application given those available in the microprocessor 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 non-state 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 will 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. Still, the savings obtained through clustering are significant enough to make the idea of a heterogeneous system attractive. Now let us consider another constraint – let us limit the multiprocessing to application classes.

Figure 7(a) shows the average ED^2P savings when we are allowed to optimally choose n architectures for the 8 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 3 and each application has its own energy-optimal core. We see that after about 4 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.

The problem with heterogeneous CMPs, 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 7(b) 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 4-5 cores.

4.5 Determining Process Validity

Finally, an experiment is devised to justify using assembly-to-assembly translations instead of compiler retargetting. 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. It suffices to show that regardless of which architecture the application is originally compiled for, the resulting family of optimal architectures is the same (or

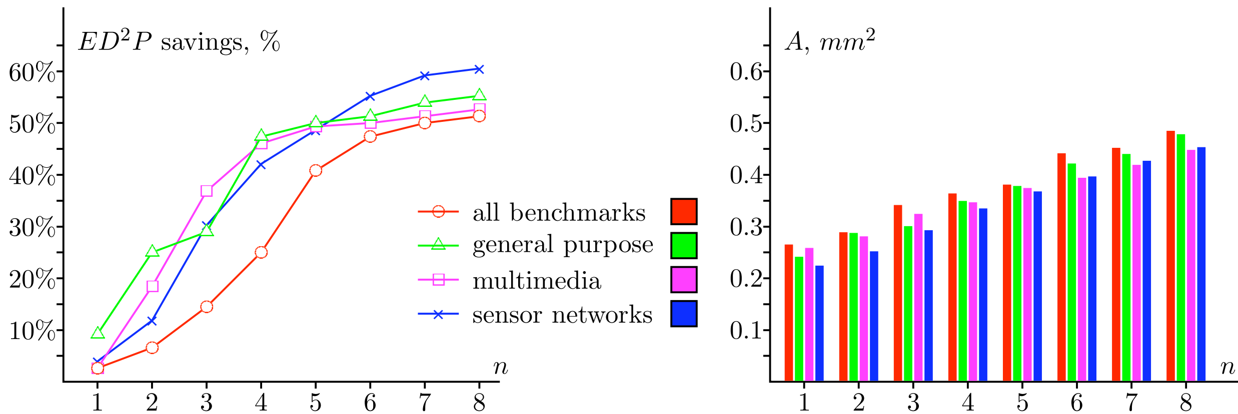

Figure 7: Savings (in terms of ED^2P) as a function of the number of processors we are allowed to include in the machine and total chip area.
(a) The architectures are found using a similar maximization problem as one described in 4.3. Every data point denotes one additional architecture. Note that for 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. (b) Assuming that the memory is shared, each additional core increases the area by a small amount.

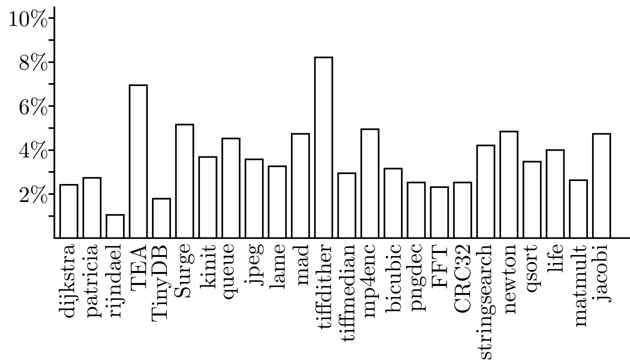

Figure 8: The differences in Pareto families for two different targets.
Each benchmark has been compiled for MIPS and again for MSP430. The Pareto optimal families are determined for each case. The height of the bar denotes the fraction of the entire set of architectures that differed between the two 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.

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 8 summarizes by how much the two families differed, as 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%.

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 paper. For example, [12] attempts 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 [1] makes attempts at application-specific ISA by varying bit-width [10]; moreover [6] proposes an entire technology platform where the instruction issue width and the various arithmetic units have been parameterized.

Furthermore, [5] designs a system which 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 paper, [5] constrains itself primarily to image processing algorithms to show the drawbacks to specialized hardware. Finally, [4] attempts to customize instruction sets of mobile and embedded applications using retargettable compilers: if the source code of an application is known, it suffices to modify the compiler to allow a variation in the target architectures.

The contributions of [12] and [6] are important, but compared to those findings, this paper:

- Considers a wide range of applications, with workloads varying from sensor network to multimedia and general purpose ones
- Considers path length, found to be critical in the performance analysis of the architectures
- Looks at a wider range of architectural issues such as architecture type (Load-store versus Stack versus Accumulator) and the number of registers.
- Finds that heterogeneous CMPs can offer significant energy and performance savings

Finally, [15] 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 paper differs from [15] 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, this paper finds compromise architectures by clustering applications and quantifying the energy-efficiency benefit of heterogeneous CMPs for these embedded applications.

In terms of research on heterogeneous CMPs, previous work primarily focused on very high performance designs and primarily same-ISA heterogeneous CMPs [11]. This paper shows that it is possible to apply principles of heterogeneity to embedded architectures with varying ISAs.

6. CONCLUSIONS AND FUTURE WORK

This paper achieves three goals. First, it 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 user has a choice of multiple architectures which execute an application optimally. Automating this design process can reduce design effort drastically. The paper shows 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.

The paper also shows that there exist clusters of applications which execute efficiently on similar architectures. It often happens that the applications executed within a sensor network belong to one cluster (for example, they are all network routing algorithms). In such cases, replacing the off-the-shelf microcontroller with a custom-made microcontroller can provide significant energy savings. The paper attempts to explain those differences by analyzing the applications themselves. This high-level description can be helpful to architecture designers in avoiding performance pitfalls. Certain classes of applications tend to prefer particular classes of architectures.

Finally, if the application clusters cannot be determined (if the applications are not known *a priori*), the paper finds that even for diverse application classes it is possible to find a small number of architectures which together achieve a near-optimal (within 15% of the optimal) energy and performance savings. Coupled with the fact that extra cores cost little (compared with the original core and memory subsystem), the results of this paper suggest that heterogeneous multiprocessors can be effectively used in embedded systems.

The paper also shows evidence of a difference between the workloads of sensor network applications, media applications and general purpose applications. As such, different classes of applications require different microcontrollers in order to fully utilize the potential of a microcontroller.

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.

It is important to note that while the configurations found in this paper 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 CMP systems.

Acknowledgments

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.

7. REFERENCES

[1] ARM Corporation. *Thumb ISA* http://www.arm.com/products/CPUs/ARM7TDMI.html

[2] J. Babb et al. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 1997.

[3] Cadence Inc. SoC Encounter. http://www.cadence.com/products/digital_ic/soc_encounter/

[4] N. Clark, W. Tang and S. Mahlke. Automatically Generating Custom Instruction Set Extensions. *First Annual Workshop on Application-Specific Processors*, 2002, pp. 94–101.

[5] J. A. Fisher, P. Faraboschi and G. Desoli. Custom-fit processors: Letting applications define architectures. In *Proceedings of Microarchitecture*, Dec. 2–4, 1996, pp. 324–335.

[6] J. A. Fisher, F. Homewood, G. Brown, G. Desoli and P. Faraboschi. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of International Symposium on Computer Architecture*, 2000, p. 203.

[7] C.-L. Fok, G.-C. Roman and C. Lu. Mobile agent middleware for sensor networks: an application case study. In *Proceedings of International Conference on Information Processing in Sensor Networks*, Apr 2005.

[8] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth IEEE Workshop Workload Characterization*, pp. 10–12, Dec 2001.

[9] J. L. Hill. System architecture for wireless sensor networks. PhD thesis, *University of California, Berkeley*, 2003.

[10] A. Krishnaswamy, R. Gupta. Mixed-width instruction sets. In *Communications of the ACM*, Vol. 46, No. 8, 2003.

[11] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *36th International Symposium on Microarchitecture, MICRO-36*, Dec 2003.

[12] L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin and D. Blaauw. Energy Optimization of Subthreshold-Voltage Sensor Network Processors. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[13] M. Poletto and V. Sarkar. Linear Scan Register Allocation. In *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, Sep 1999, pp. 895–913.

[14] Standard Performance Evaluation Corporation. *The Integer SPEC95 Benchmarks.* http://www.spec.org/cpu95/CINT95/

[15] Tensilica. Xtensa LX Processor. http://www.tensilica.com/products/xtensa_LX.htm

[16] Texas Instruments. *TI MSP430 Product Brochure.* http://focus.ti.com/lit/ml/slab034k/slab034k.pdf

[17] UMC Faraday 0.13μm Libraries. http://freelibrary.faraday-tech.com/ips/013library.html.

[18] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of European Workshop on Sensor Networks*, Jan 2005.