A Self-Optimizing Embedded Microprocessor using a Loop Table for Low Power

Frank Vahid* and Ann Gordon-Ross
Department of Computer Science and Engineering
University of California, Riverside
http://www.cs.ucr.edu/~vahid
{vahid/ann}@cs.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine.

ABSTRACT
We describe an approach for a microprocessor to tune itself to its fixed application to reduce power in an embedded system. We define a basic architecture and methodology supporting a microprocessor self-optimizing mode. We also introduce a loop table as a tunable component, although self-optimization can be done for other tunable components too. We highlight experimental results illustrating good power reductions with no performance penalty.

Keywords
System-on-a-chip, self-optimizing architecture, embedded systems, parameterized architectures, cores, low-power, tuning, platforms.

1. INTRODUCTION
Several billion microprocessors targeted for embedded systems are produced annually. Most of those embedded microprocessors are mass-produced. Mass-produced integrated circuits (IC's) can cost very little per IC, sometimes less than one dollar, due to the amortization of engineering costs over large volumes, high yields from large production runs, and other economy-of-scale factors. Furthermore, mass-produced IC's often justify more design attention and use of advanced IC technology, and thus may provide excellent power and performance compared even with customized IC's produced in smaller volumes. Mass-produced parts also provide immediately available parts and hence faster time-to-market. The trend towards system-on-a-chip (SOC) still includes mass-produced SOC parts, known in that context as programmable platforms [12].

Typically, an embedded microprocessor runs only one application program for its lifetime, in contrast to microprocessors used in other domains. Since low power is so critical in many embedded systems, we would therefore like to customize a microprocessor to its one application in order to reduce power.

IC transistor capacities have continued to increase at a tremendous rate, following Moore's Law [6]. One non-obvious use of the additional transistor capacity is to reduce power in a mass-produced embedded microprocessor by, adding tunable components to the architecture, and extra circuitry that tunes those components to the particular fixed application. Essentially, the microprocessor is self-optimizing. A designer using such a part gets reduced power through some customization while still getting the benefits of a mass-produced IC.

In this paper, we describe a basic architecture and methodology for a self-optimizing microprocessor that tunes itself to an application to reduce power. Such a microprocessor represents an instance of post-fabrication tuning [16], namely tuning done after an IC has been fabricated. We introduce self-profiling circuitry and a designer-controlled self-optimization mode, in which configurable architectural components would be tuned based on an application's profile.

To illustrate self-optimization, we introduce a tunable component called a *loop table*. A loop table is a small memory inside the microprocessor controller, which stores frequently executed loops, thus reducing power-expensive accesses to program memory. On the surface, the loop table is similar to loop-cache approaches, but it differs in how and when its contents are updated. Furthermore, our particular implementation has the desirable feature of not changing cycle-by-cycle behavior. Other tunable components could be used with our self-optimization approach, such as caches with a configurable number of ways [11].

We describe experimental results that show good power and energy reductions on several examples, at the cost of extra gates, and with no change in performance.

2. PROBLEM DESCRIPTION AND RELATED WORK
We wish to develop a mass-producible extended version of a standard embedded microprocessor that can tune its configurable components to a particular application for low power. In doing so, we may use extra transistors, as they are becoming cheaper every year, with the following goals.

First, we desire exact instruction set compatibility, meaning we should avoid adding, deleting, or changing any instructions. Thus, the self-optimizing microprocessor should run any binary designed for the standard microprocessor, and likewise, any binary designed for the self-optimizing processor should run on other versions of the standard microprocessor. Backwards compatibility is a big issue in embedded systems, as is the related-issue of minimizing risk associated with future changes [2].

Second, we would like to avoid modifying or adding any tools in the development tool chain. Embedded system designers tend to have significant investments in high-quality, stable tools. Modifications or additions are not commonly accepted [2].

Third, we would like to preserve cycle-by-cycle behavior. Many embedded system programmers, especially those using microcontrollers, write code that must execute exactly as written to satisfy detailed timing constraints with the external environment.

Tuning a microprocessor to its program has been addressed in previous work from different perspectives: ASIPs, dynamic binary translation, program compression, and caching, which we now discuss. All of the techniques try to improve the execution of frequently-executed program regions.

ASIPs (Application-Specific Instruction-set Processors) involve generating a custom instruction-set optimized for a given application, along with a custom tool chain (e.g., compiler). Some such approaches are pre-fabrication [2], with commercial products having recently appeared [4]. Other approaches extend a microprocessor with programmable logic to enable post-fabrication instruction-set customization [9] of mass-produced IC's. In either case, ASIPs obviously require changes to binaries and to tool chains. Somewhat related is a commercial mass-produced IC [15] having a microcontroller extended with programmable logic, where that logic can be used to customize a standard microcontroller's peripherals. Some recent work has focused on tuning a microprocessor without modifying the instruction set [13].

Dynamic binary translation involves on-chip on-the-fly translation of one binary to another binary. A recent commercial processor [8] dynamically profiles a binary to find the most frequently executed code regions, and then caches the corresponding translated binary region, to avoid having to translate that region again, thus saving power. The profiler task runs on the processor itself, so obviously changes cycle-by-cycle behavior.

Some program compression techniques pre-profile a program to detect the most frequently executed regions of code, and compress those regions in program memory to reduce the number of bits transmitted over a power-expensive program memory bus [5]. Those regions are then decompressed in the microprocessor. Such approaches require additional profiling and compression tools, and change the binary.

Caching can also be viewed as tuning to an application. Most caching approaches store the most recently executed blocks in an on-chip fast memory. Trace caches [3] seek to tune further, by caching commonly-executed sequences of blocks, requiring additional trace profiling circuitry. This and other previous caching work focus on improving performance.

Some recent efforts on caching for low power are closely-related to our loop table approach. In many applications, a few small loops account for a large percentage of clock cycles. Since accesses to program memory consume much power, these techniques seek to minimize such accesses by caching those loops in a small, low-power memory. One approach is a filter cache [7], which is an unusually small cache, perhaps just hundreds of bytes, placed closest to the microprocessor in the memory hierarchy. Although the miss rate will be high, the power consumed per hit is so low as to reduce power consumption overall. An extension of this approach [1] involves pre-profiling the program to detect the most frequently-executed loops, having the compiler place those loops in special memory regions, and extending the architecture to detect those regions. Only code from those regions will be placed in the filter cache (renamed the loop cache), eliminating the problem of high miss rate, at the expense of tool chain additions and changes. Another approach, proposed by [10], uses a cache specifically designed for loops. This loop cache is only filled when a short backwards jump instruction is detected, which typically indicates a small loop. Execution stays within the loop cache, using a counter to sequence to subsequent addresses, until the short backwards jump is not taken. Notice that this approach can satisfy all of our goals – no binary modification, no tool modification, and preserved cycle-by-cycle behavior. Thus, we use this as the basis for our loop table – the key difference being that we want to use profiling to ensure that only the most frequently executed loops get cached and that those loops never leave the loop table, thus reducing runtime overhead and hence reducing power even further.

A general discussion of pre- and post-fabrication tuning for embedded microprocessors can be found in [16].

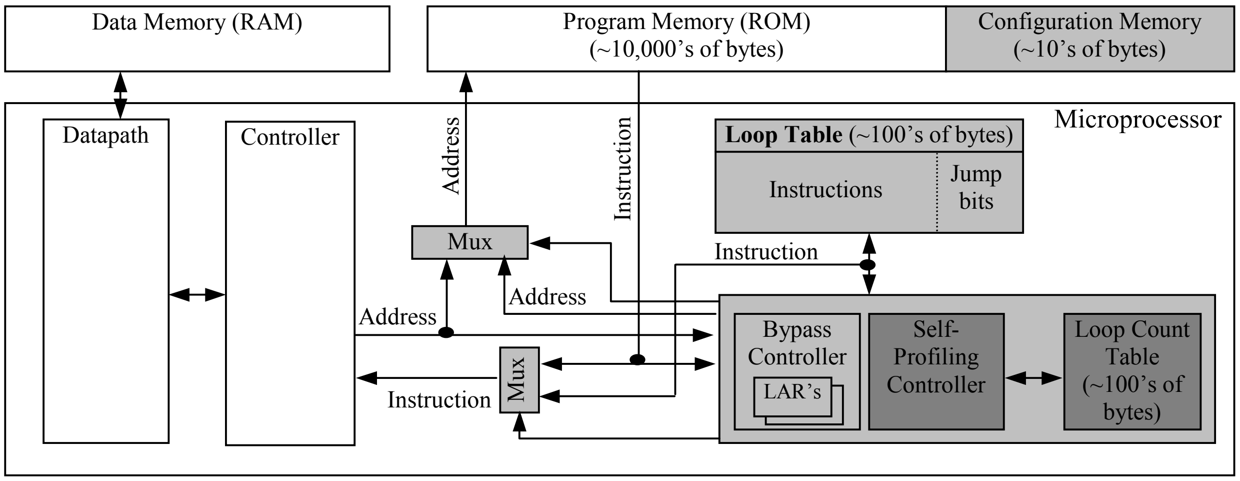
3. ARCHITECTURE AND METHODOLOGY OVERVIEW

We developed our architecture and methodology using a standard microcontroller. Microcontrollers represent a well-known class of embedded microprocessors. A microcontroller is an IC possessing a microprocessor with tightly-integrated peripherals, program memory, and data memory. Microcontrollers are typically used in control-dominated applications, having simple architectures oriented towards bit-manipulation, usually excluding features like multipliers, floating-point units, caches, deep pipelines, and branch predictors. Popular microcontrollers include the Intel 8051, and the Motorola 68HC05, 68HC11, and 68HC12.

Figure 1 shows the architecture of a microcontroller's processor core and memories. Peripherals are not shown. The white boxes correspond to the basic architecture; the gray boxes will be discussed later. After an application program has been developed for this microcontroller, the application will be downloaded into the program memory, and does not change for the life of the target product. The program memory is typically some form of programmable ROM, like EPROM, EEPROM, or Flash. Each access to such memory typically consumes much power, since such access uses large, high-capacitance buses. This is especially true if the memory is on a separate chip. High power per access, coupled with the obvious fact that program memory is also accessed very frequently, leads to program memory accesses contributing to a large part of a microcontroller's total power consumption, as can be seen in Figure 5. Thus, reducing program memory accesses is a good focal point for power reduction.

In Figure 1, we have used gray boxes to represent additional hardware for a self-optimizing microcontroller architecture. A *Loop Table* stores copies of frequently-executed loops. A *Bypass Controller* detects addresses, contained in one or more loop address registers (*LAR's*), of loops stored in the loop table. A *Self-Profiling Controller* determines the most frequently-executed loops with the aid of a *Loop Count Table*. These items will all be described further.

Figure 2 illustrates where self-optimization would occur in a design flow. It occurs after fabrication, in a mass-produced IC. It occurs in-system, meaning we need not develop complicated and

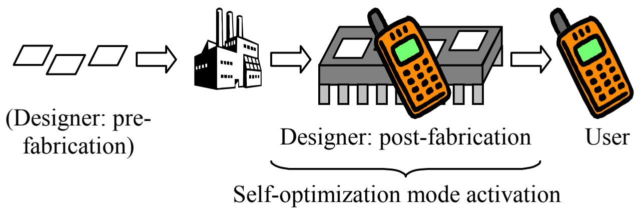


Data Memory (RAM)
Program Memory (ROM) (~10,000's of bytes)
Configuration Memory (~10's of bytes)
Microprocessor
Datapath
Controller
Address
Instruction
Mux
Loop Table (~100's of bytes)
Instructions
Jump bits
Instruction
Address
Address
Instruction
Mux
Bypass Controller
LAR's
Self-Profiling Controller
Loop Count Table (~100's of bytes)

Figure 1: Self-optimizing microcontroller architecture. Items not found in a standard microcontroller are shown in gray.

non-standard simulations. The tuning is done under designer control. Traditional ASIP approaches perform tuning before fabrication, while caching and dynamic binary translation approaches occur during end-use.

Figure 3 illustrates the self-optimization methodology we propose. First, a designer downloads an application program into the microcontroller program memory. Second, the designer resets the IC, and can then execute the program in *normal mode*. Up to that point, the self-optimizing features of the IC are completely invisible. Third, the designer can optionally activate a special *self-optimizing mode*. This mode actually consists of three steps: initialize, profile, and configure. This mode activates the dark-gray components of Figure 1. Here, the program is executing in its real environment, but is also being monitored and profiled. After self-optimization mode is completed, a configuration memory gets written with information on how to configure the architecture to consume less power for the given profile. In our loop table case, the configuration memory would contain the addresses of the most frequently executed loops. Fourth, the designer can again reset the IC, which this time will be configured using the configuration memory. In our case, this means that certain program memory regions will be copied into the loop table. The program executes in normal mode again, but now uses the light-gray components in Figure 1, so should consume less power than before. Finally, a designer may wish to upload the configuration memory to a workstation, and then download that configuration into a large number of identical parts. We now describe the three self-optimization steps in more detail.



(Designer: pre-fabrication)
Designer: post-fabrication
User
Self-optimization mode activation

Figure 2: Self-optimization mode is in-system but under designer control.

4. SELF-OPTIMIZING MODE

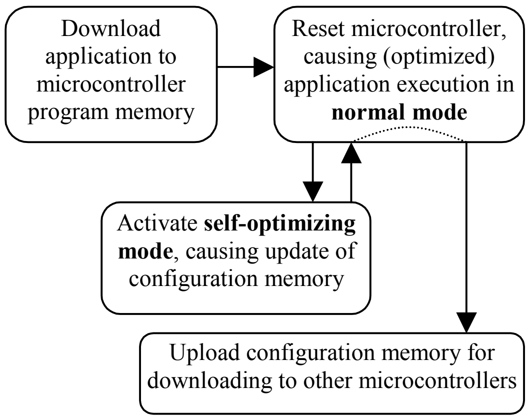
4.1 Initializing

The first question we faced was determining how to activate self-optimization. One option is to include an extra input pin on the IC. A second is software activation by setting of a bit in a special register. A third is to use a special combination of values on existing pins including the reset pin. Similar methods can be used for the chip to indicate completion of self-optimization. We currently use the first option.

Once activated, initializing for our loop table consists of traversing the program memory to detect all possible loops, and placing the starting address of each loop in a loop count table. Loops can be detected by finding relatively short backwards jumps in the program memory. Filling the loop count table is handled by the self-profiling controller in Figure 1. Initialization is only done once in self-optimization mode.

4.2 Profiling

Next, the self-profiler executes the application in program memory, monitors the program memory address bus, and updates



Download application to microcontroller program memory
Reset microcontroller, causing (optimized) application execution in normal mode
Activate self-optimizing mode, causing update of configuration memory
Upload configuration memory for downloading to other microcontrollers

Figure 3: Self-optimization methodology.

the loop count table to reflect the number of times each loop is executed. Note that the application runs in its real environment, so this profiling information can be quite accurate.

We must therefore maintain a table of loop addresses and associated count values in the loop count memory. However, profiling cannot change the cycle-by-cycle behavior of the application's execution. Thus, when a loop address is detected, its corresponding count value must be looked up and incremented within the time that it takes to execute the instruction. This can be hard. Storing the counts as a linked list in a memory will not work, because finding the record with the current loop address will take many cycles.

One solution is to use a loop count table that is the same size as the program memory. Thus, the current loop address indexes directly to a memory location storing the count value for that address. This is obviously wasteful in terms of memory, since only a small fraction of program memory addresses correspond to loops. For example, an application might have 64,000 words of program memory, but only 100 loops.

We will instead need to use a much smaller memory, having perhaps a few hundred words. We investigated two possible options for the loop count table. One is to make it fully-associative, where each location in the memory consists of both a key (the loop address) and data (the count value). As with any fully-associative memory, an access consists of simultaneously comparing the input address with all keys. If the microprocessor executes instructions in only a few cycles, then an increment feature could be added to each word of this memory. Otherwise, if more cycles are available (as is often the case in microcontrollers), then the count value can be read, incremented, and written back to the memory location. The drawback of fully-associative memories is the many gates used for comparison logic.

A second option for the loop count table is a hardware hash table. We can use a simple hashing function to map program memory addresses to loop count table addresses. Again, since the ratio of program memory addresses to loops is large, we should be able to design a simple hash function, using the address bits, that minimizes conflicts. If enough cycles exist per instruction, we can even tolerate some conflicts, since we could use multiple words per location, and/or simply overflow to the next location on a conflict. We could even create an N-way set-associative cache in order to tolerate some conflicts without loss of cycles. Again, an auto-increment feature could be incorporated into the loop count table if we only have a few cycles per instruction. We currently use a fully-associative memory.

The location of the loop count table in the overall architecture is shown in Figure 1. Note that in any loop count table with fewer locations than program memory, we run the risk of not being able to accommodate all loop addresses. This will not result in incorrect functionality, but merely yield less-than-optimal results, which is acceptable. Instead, we could perform several rounds of such loop counting on subsets of all the loop addresses.

Profiling must be done for sufficient time, but not for so long as to cause overflow in the count values of the profile table.

Profiling will consume extra power while executing. However, we again stress that the profiling circuit only runs during self-optimization mode; during normal mode, this circuit is shut down completely and does not contribute to power consumption. We point out that the IC must be designed to tolerate this extra power during self-optimization mode. We also point out that the extra circuitry could indirectly increase wire capacitance of the microprocessor core by lengthening some wires, but this increase should be far outweighed by the decrease from fewer program memory accesses.

4.3 Configuring

After profiling, we want to determine how to best tune the architecture to the given profile. In our case, we want to put frequently executed loops in the loop table. Actually, during configuration, we just want to store the addresses of those loops in a non-volatile memory, so that when the IC is reset in the future, we can then load the appropriate loops into the loop table. We want to store this information in non-volatile memory because we do not want to have to run self-optimization every time we power up the microcontroller. Once self-optimization is done, the IC should consume less power for the remainder of its lifetime, as long as the same program resides in program memory.

We achieve this goal by keeping the configuration memory in a reserved region at the bottom of program memory. This assumes we are using in-system programmable ROM, like EEPROM or Flash, and that enough extra words exist in that memory. A particular location of this region is a flag; if the flag is set, then this means that the region includes valid configuration information. If not set, it means self-optimization has not yet been run. A nice feature of putting the configuration memory in program memory is that, whenever a new program is downloaded, the flag automatically gets reset. This reset is desirable because configuration information for a previous program is not necessarily valid for a new program. The drawbacks are that there must be extra words available for a configuration information region, and this particular region must be independently in-system programmable.

Alternatively, we could use extra memory for the configuration memory, but this requires extra non-volatile memory hardware, and must include a method of resetting the configuration flag when the program memory is updated.

5. NORMAL MODE

In normal mode, the microcontroller executes its application. If no previous self-optimization was performed, this execution will be identical to that on a microcontroller without our extensions. If self-optimization was performed and hence the configuration memory updated, execution should consume less power. We now describe the behavior of a microcontroller reset, and execution without and with prior self-optimization.

5.1 Reset

During reset caused by power up and/or assertion of the external reset pin found on all microcontrollers, the microcontroller checks the flag in configuration memory. If set, the microcontroller reads the loop start addresses from configuration memory, copies the corresponding loops from program memory into the loop table, and stores the loop addresses in the bypass controller's loop address registers. It also sets a flag (flip-flop) in the bypass controller indicating self-optimization configurations are activated. These tasks are in addition to other reset tasks. This may or may not increase the cycles needed for reset, but the

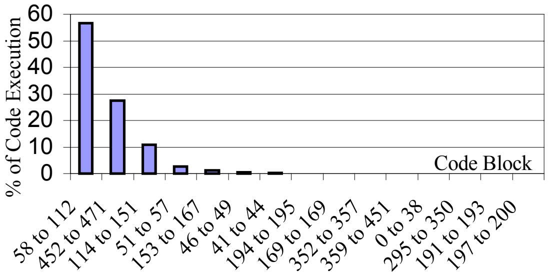


Figure 4: A few loops often comprise most of an application's execution time, as in the matrix multiply example.

precise number of cycles needed for reset is not an issue in most cases, and is distinct from the earlier requirement of cycle-by-cycle accuracy during execution.

5.2 Executing from the Program Memory

If the configuration flag is set, the microcontroller compares each address, generated by the controller and destined for program memory, with the loop address registers. If a match is not found, or the configuration flag is not set, then that address proceeds to program memory and instructions are fetched as usual.

5.3 Executing from the Loop Table

If the configuration flag is set and an address destined for program memory matches with an address register, program memory is not accessed, and program memory address lines are held constant to reduce switching activity and hence power; in fact, program memory can even be shut down at this time. Instead, the first instruction of the corresponding loop is fetched from the loop table immediately, providing a seamless transition to a loop table mode of operation. Once in loop table mode, subsequent addresses are immediately translated into loop table accesses, until we transition out of this mode.

The transition out of loop table mode must be seamless as well. If the controller for the loop table were just to examine the addresses of read operations to determine if they were in the loop's address range, extra cycles would be required for the comparison. To avoid extra cycles, we keep two extra bits in the loop table to quickly detect departures out of the loop. There are two ways for execution to leave the loop table: either the jump at the end that returns to the beginning of the loop is not taken, or a jump is encountered in the loop that jumps out of the loop. During self-optimization, the code to be placed in the loop table is examined to determine which instructions may take execution out of the loop. For each instruction, two extra bits are included in the loop table. 00 means this instruction cannot exit the loop – it is either not a jump, or it jumps to a location inside the loop (and is not the last instruction). 10 means the instruction is a jump that exits the loop if *not* taken (meaning it is the last instruction). 11 means the instruction is a jump that exits the loop if taken. An exit causes subsequent instructions to immediately be fetched from program memory.

5.4 Experimental Results

We performed experiments to validate the effectiveness of our architecture and methodology. We used an existing synthesizable VHDL model of an 8051 microcontroller as a starting point [16]. We modified this model with the architectural extensions described in this paper. The initial version supports only a single-loop in the loop table. All experiments were performed using Synopsys synthesis, simulation and power analysis tools [14]. The synthesis tool takes a VHDL register-transfer-level representation of the microcontroller and outputs a gate-level netlist. To account for high capacitance of large buses in deep submicron technologies, the long bus wires going to program memory (in a special chip region or even separate chip supporting a programmable ROM technology) have a capacitance 100 times greater than a typical short on-chip wire, while the data memory (RAM) bus has a capacitance 10 times greater. Simulation of an application running on the microcontroller is done to obtain a count of the switching on each net. Total power consumption is computed using the standard equation for dynamic CMOS power, $\frac{1}{2}CV^2f$, for each net, summed over all nets, where C is a net's capacitance, V is voltage, and f is the switching frequency as computing during simulation. This is all done using standard Synopsys tools.

We experimented with three examples. *Ex1* is a program that computes checksums. *Ex2* computes the greatest common divisor of two numbers. *Ex3* performs matrix multiplication. All assembly program code was generated from C source by the Keil C compiler, and we made no modifications to the assembly code. Each of these examples has the common embedded system property of spending much time in small loops. *Ex1* spends 97% of its time in 36% of the program code. *Ex2* spends 77% of its time in 31% of the code. *Ex3* spends 57% of its time in 11% of the code (A), and another 28% of its time in 4% of the code (B) as seen in Figure 4. Upon investigation, we noticed that A is a loop that at one location calls B; B is a block that executes and jumps back to A. Thus, B could be inlined into A to create a single loop, which would have yielded even better power savings than our current single-loop loop table could provide. This suggests the need for tuning not just the architecture to the application, but also the application to the architecture, an area of future work.

Figure 5 provides power results for these three examples, on the 8051 without any extensions (*bef*), and on the 8051 with extensions and after self-optimization (*aft*), with power data broken down by microprocessor subsystem. Overall power is significantly reduced in all three examples, as desired, averaging nearly 34% total power reduction. Reductions will be even greater when we allow more than one loop in the loop table. Notice that power related to ROM access decreases by an average of 50%, with additional power due to the loop table and additional control logic being small. The percentages depend greatly on the IC process technology; as feature sizes decrease, the capacitance of large buses compared with internal nets increases, making the loop table approach more attractive as features continue to shrink.

Performance was not changed, as desired. No clock cycles were added or removed during normal or self-optimization operation. The clock cycle length was not modified either. Note that since performance did not change, the power reductions also imply energy reductions.

Though not shown in the figure, we also measured the power after architectural extensions were made but before self-optimization. There was no noticeable increase. Furthermore, we measured the power during self-optimization. The average power increase during this temporary mode was only about 5%.

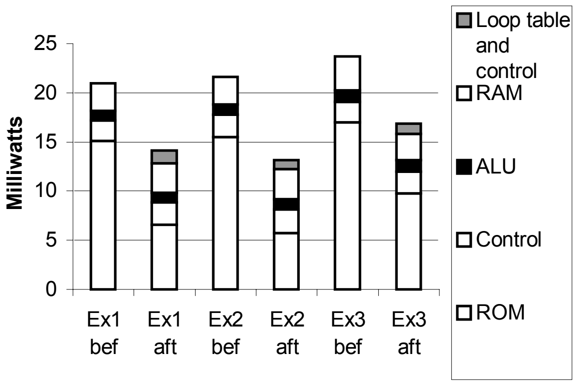


Figure 5: Power consumption for the examples.

Size data is provided in Table 1. We see that the loop count table adds the most to the size, and the loop table adds the next largest amount. We are currently working on reducing the sizes of the memories. We also note several items. First, transistor budgets are very large and still growing [6], representing a very different design situation from a few years ago. Second, the loop count table and profiler transistors are completely idle during normal operation. Third, if we are willing to accept an increase in cycle-by-cycle behavior during self-optimization, then the loop count table can be reduced to a much smaller memory. Fourth, the microcontroller version we used is as small as they come. Other versions have more program and data memory, and other microcontrollers have more complex logic, but the self-optimization extensions would not increase for those, meaning the percentage size increase would be much less. Finally, we can envision that IC's having only the loop table and by-pass logic, but not the self-profiling logic and loop count table, could be mass-produced as smaller and lower cost product-oriented parts, with the complete self-optimizing architecture serving as prototype-oriented parts [16].

6. CONCLUSIONS

Pre-fabricated mass-produced microcontrollers are extremely popular due to low-cost and short time-to-market. Previously, they could not be tuned to a specific application to reduce power consumption. We introduced a methodology and architecture that makes use of today's abundance of transistors to enable such tuning in a mass-produced part. The key is a special self-optimization mode that profiles the application and saves tuning configuration information. We illustrated our approach by introducing a loop-table as a tunable component, and showed that self-optimization using a loop table reduced power significantly in several examples. We plan to experiment with larger benchmarks, as well as develop additional tunable components to achieve greater power reductions.

Table 1: Size breakdown in gates

Subsystem	Original	Extended
Controller	3,391	3,767
ALU	2,100	2,100
Decoder	586	586
RAM	17,312	17,312
ROM (8kbytes)	11,000	11,000
Select logic		132
Loop Count Table		33,595
Loop Table		16,740
Self-Profiler/Bypass		7,188
Total:	34,389	92,420

7. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under grants CCR-9811164 and CCR-9876006.

8. REFERENCES

[1] Bellas, N.; Hajj, I.; Polychronopoulos, C.; Stamoulis, G. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. International Conference on Computer Design, pp. 378-383, 1999.

[2] Fisher, J.A.. Customized Instruction-Sets for Embedded Processors. Design Automation Conference, pp. 253-257, 1999.

[3] Friendly, D., S. Patel, Y. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. ACM/IEEE International Symposium on Microarchitecture, 1998.

[4] Gonzalez, R.E. Xtensa: A Configurable and Extensible Processor. IEEE Micro, pp. 60-70, 2000. Also see Tensillica Corp., http://www.tensillica.com.

[5] Ishihara, T., H. Yasuura. A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors. Design Automation and Test in Europe, March 2000.

[6] Kiefendorff, K.. Transistor Budgets Go Ballistic. Microprocessor Report, Volume 12, Number 10, August 1998, pp. 34-43.

[7] Kin, J., M. Gupta, W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. International Symposium on Microarchitecture, pp. 184-193, December 1997.

[8] Klaiber, A.. The Technology Behind Crusoe Processors. Transmeta Corporation White Paper, January 2000.

[9] Kucukcakar, K. An ASIP Design Methodology for Embedded Systems. Int. Workshop on Hardware/Software Codesign, pp. 17-21, 1999.

[10] Lee, L. H., B. Moyer, J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops. International Symposium On Low Power Electronics and Design, 1999.

[11] Malik, A., B. Moyer, D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. Int. Symposium on Low Power Electronics and Design, pp. 241-243, 2000.

[12] Semiconductor Industry Association. International Technology Roadmap for Semiconductors: 1999 edition. Austin, TX:International SEMATECH, 1999.

[13] Stitt, G., F. Vahid, T. Givargis and R. Lysecky. A First Step Towards an Architecture Tuning Methodology for Llow Power. International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2000.

[14] Synopsys Inc., http://www.synopsys.com.

[15] Triscend Corporation, http://www.triscend.com.

[16] Vahid, F. and T. Givargis. Platform Tuning for Embedded Systems Design. IEEE Computer, Vol. 34, No. 3, pp. 112-114, March 2001. Also see The UCR Dalton Project, http://www.cs.ucr.edu/~dalton.