Presented at the *International Conference on Computer-Aided Design*, San Jose, CA, Nov 1994

Power Analysis of Embedded Software: A First Step Towards Software Power Minimization

Vivek Tiwari Sharad Malik Andrew Wolfe

Dept. of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

Embedded computer systems are characterized by the presence of a dedicated processor and the software that runs on it. Power constraints are increasingly becoming the critical component of the design specification of these systems. At present, however, power analysis tools can only be applied at the lower levels of the design – the circuit or gate level. It is either impractical or impossible to use the lower level tools to estimate the power cost of the software component of the system. This paper describes the first systematic attempt to model this power cost. A power analysis technique is developed that has been applied to two commercial microprocessors – Intel 486DX2 and Fujitsu SPARClite 934. This technique can be employed to evaluate the power cost of embedded software and also be used to search the design space in software power optimization.

1 Introduction

Embedded computer systems are characterized by the presence of a dedicated processor which executes application specific software. Recent years have seen a large growth of such systems. This growth is driven by several factors. The first is an increase in the number of applications as illustrated by the numerous examples of "smart electronics" around us. The second factor leading to their growth is the increasing migration from application specific logic to application specific code running on existing processors. The migration to software programmable solutions can often provide the competitive edge in terms of lower manufacturing costs and shorter time to market. Thus, we are seeing a movement from the logic gate being the basic unit of computation on silicon, to an instruction running on an embedded processor.

A large number of embedded computing applications are power critical, i.e., power constraints form an important part of the design specification. While there has been a significant research effort in power estimation and low power design, there is very little available in the form of design tools to help embedded system designers evaluate their designs in terms of the power metric. At present, power measurement tools are available for only the lower levels of the design - at the circuit level and the gate level. At the least these are very slow and impractical to use to evaluate the power consumption of software, and often cannot even be applied due to lack of availability of circuit and gate level information of the embedded processors. The embedded processors currently used in designs take two possible shapes. The first is "off the shelf" microprocessors or digital signal processors (DSPs). The second is in the form of cores embedded in larger integrated circuits. In the first case, the processor information available to the designer is whatever is made available through data books. In the second case, the designer has logic/timing simulation models to help verify the designs. In neither case is there lower level information available for power analysis.

This paper describes a power analysis technique for embedded software. The goal is to develop and validate an instruction level power model for embedded software. Such a model can then be provided by the processor vendors for both off the shelf processors as well as embedded cores. This can then be used to evaluate embedded software, much as a gate level power model has been used to evaluate logic designs. This is useful in its own right to verify that a design meets its specified power constraints. In addition, it can also be used to search the design space in software power optimization. The technique has so far been applied to two commercial microprocessors – the Intel 486DX2 and the Fujitsu SPARClite 934. This paper uses the former as a basis for illustrating the technique. The application of this technique for the latter is described in a separate reference [4].

2 Experimental Method

While it is recognized that the power consumption of a processor varies from program to program, there is a complete lack of models and tools to analyze this variation. Traditional attempts to model the power consumption in the CPU rely on detailed physical layout of the processor and sophisticated power analysis tools that use the information provided by the layout.

In the case of embedded system design, detailed layout information of the CPU is often not available. Even if it is available, these techniques are expensive and difficult to apply. This is also the reason why the potential for power reduction through modification of software is so far unknown and unexploited. The thrust of our work is to overcome these deficiencies by developing a power estimation methodology based on actual laboratory measurements. Given a measurement setup to measure the current being drawn by the microprocessor, the only other information required can be obtained from the widely available manuals and handbooks specific to that microprocessor.

The main idea is to formulate an instruction level power model for the microprocessor. Given this model and an assembly/machine level program, the power consumption in the program can be efficiently estimated. The specifics of the measurement methodology are described next.

2.1 Power and Energy

The average power consumed by a microprocessor while running a certain program is given by: $P = I \times V_{CC}$, where P is the average power, I is the average current and V_{CC} is the supply voltage. Since power is the rate at which energy is consumed, the energy consumed by a program is given by: $E = P \times T$ where T is the execution time of the program. This in turn is given by: $T = N \times \tau$ where N is the number of clock cycles taken by the program and τ is the clock period.

In common usage, the terms power consumption and energy consumption are often interchanged. However it is important to distinguish between the two when we talk of either of these in the context of programs running on mobile applications. Mobile systems run on the limited energy available in a battery. Therefore the energy consumed by the system or by the software running on it determines the length of the battery life. Energy consumption is thus the focus of attention. We will attempt to maintain a distinction between the two in the rest of the paper. However, in certain cases the term power may be used to refer to energy, in adherence to common usage.

2.2 Current Measurement

For this study, the processor used was a 40MHz Intel 486DX2-S Series CPU. The CPU was part of a mobile personal computer evaluation board with 4MB of DRAM memory. The reason for the choice of this processor was that its board setup allowed the measurement of the CPU and DRAM subsystem current in isolation from the rest of the system. *We would like to emphasize that while the numbers we report here are specific to this processor and board, the methodology used by us in developing the model is widely applicable.* The current was measured through a standard off the shelf, dual-slope integrating digital ammeter.

If a program completes execution in a short time, a current reading cannot be obtained visually. To overcome this, the programs being considered were put in infinite loops and current readings were taken. The current consumption in the CPU will vary in time depending on what instructions are being executed. But since the chosen ammeter averages current over a window of time ($100ms$), if the execution time of the program is much less than the width of this window, a stable reading will be obtained.

The main limitation of this approach is that it will not work for programs with larger execution times since the ammeter may not show a stable reading. However, in this study, the main use of this approach was in determining the current drawn while a particular instruction (instruction sequence) was being executed. A program written with several instances of the targeted instruction (instruction sequence) executing in a loop, has a periodic current waveform which yields a steady reading on the ammeter. This inexpensive approach works very well for this. However the main concepts described in this paper are independent of the actual method used to measure average current. If sophisticated data acquisition based measurement instruments are available, the measurement method can be based on them.

For our setup, V_{CC} was $3.3V$ and τ was $25ns$, corresponding to the 40MHz internal frequency of the CPU. Given these constants, energy is proportional to the average current and number of cycles. Unless otherwise stated, the numbers reported in this paper correspond to average current in mA.

3 Instruction Level Modeling

A modern microprocessor like the 486DX2 is an extremely complex system consisting of several interacting functional blocks. However, this internal complexity is hidden behind a simple interface – its instruction set. Thus to model the energy consumption of this complex system, it seemed intuitive to consider individual instructions. Each instruction involves specific processing across various units of the CPU. This can result in cir-

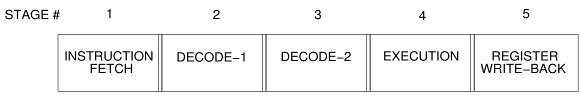

Figure 1: Internal Pipelining in the 486DX2

cuit activity that is characteristic of each instruction and can vary with instructions.

This intuition was the starting point for the empirical study that led to the development of the final instruction-level energy model. Under this model each instruction in the instruction set is assigned a fixed energy cost called the *base energy cost*. The variation in base costs of a given instruction due to different operand and address values is then quantified. The base energy cost of a program is based on the sum of the base energy costs of each executed instruction. However, during the execution of a program, certain inter-instruction effects occur whose energy contribution is not accounted for if only base costs are considered. The first type of inter-instruction effect is the effect of circuit state. The second type is related to resource constraints that can lead to stalls and cache misses. The energy cost of these effects is also modeled and used to obtain the total energy cost of a program.

The instruction-level energy model described here is based on actual measurements and evolved as a result of extensive experimentation. It is comprehensive and provides all the information needed to evaluate programs in terms of their energy costs. The various components of this model are described in the subsections below.

3.1 Base Energy Cost

The base cost for an instruction is determined by constructing a loop with several instances of the same instruction. The average current being drawn is then measured. This current multiplied by the number of cycles taken by each instance of the instruction is proportional to the total energy as described in Section 2.

While this method seems intuitive if the CPU is executing only one instruction at a given time, most modern CPUs, including the 486DX2 are processing more than one instruction at a given time due to pipelining. However, the following discussion shows that the concept of a base energy cost per instruction and its derivation remains unchanged.

The 486DX2 CPU has a five-stage pipeline as shown in Figure 1 [1]. Let Ej_{I_k} be the average energy consumed by pipeline stage j, when instruction I_k executes in that stage. Pipeline stages are separated from each other by latches. Thus, if we ignore the effect of circuit state and resource constraints for now, the energy consumption of different stages is independent of each other. Let us assume that in a given cycle, instruction I_1 is being processed by stage 1, I_2 by stage 2, and so on. The total energy consumed by the CPU in that cycle would be: $E_{cycle} = E1_{I_1} + E2_{I_2} + E3_{I_3} + E4_{I_4} + E5_{I_5}$. On the other hand, the total energy consumed by a given instruction I_1, as it moves through the various stages is: $E_{ins} = \sum_j Ej_{I_1}$. This quantity actually refers to the base cost in the sense described above. Our method of forming a loop of instances of instruction I_1, results in $E_{cycle} = E_{ins}$, since in that case, $I_1 = I_2 = I_3 = I_4 = I_5$. The average current in this case is $\sum_j Ej_{I_1}/(V_{CC} \times \tau)$, which is the same as the ammeter reading obtained.

Number	Instruction	Base Cost (mA)	Cycles
1	`NOP`	275.7	1
2	`MOV DX,BX`	302.4	1
3	`MOV DX,[BX]`	428.3	1
4	`MOV DX,[BX][DI]`	409.0	2
5	`MOV [BX],DX`	521.7	1
6	`MOV [BX][DI],DX`	451.7	2
7	`ADD DX,BX`	313.6	1
8	`ADD DX,[BX]`	400.1	2
9	`ADD [BX],DX`	415.7	3
10	`SAL BX,1`	300.8	3
11	`SAL BX,CL`	306.5	3
12	`LEA DX,[BX]`	364.4	1
13	`LEA DX,[BX][DI]`	345.2	2
14	`JMP label`	373.0	3
15	`JZ label`	375.7	3
16	`JZ label`	355.9	1
17	`CMP BX,DX`	298.2	1
18	`CMP [BX],DX`	388.0	2

Table 1: Subset of the Base Cost Table for the 486DX2[2]

Some instructions take multiple cycles in a given pipeline stage. All stages are then stalled. The reasoning applied above, however remains unchanged. The base energy cost of the instruction is just the observed average current value multiplied by the number of cycles taken by the instruction in that stage. For instance, consider a loop of instruction I_1, where I_1 takes m cycles in the 4th stage. Therefore, $E4_{I_1}$ is spread over m cycles. For sake of brevity assume that each stalled stage consumes zero energy. Then the current value observed on the ammeter will be $\sum_j Ej_{I_1}/(V_{CC} \times \tau \times m)$. This quantity multiplied by m yields $\sum_j Ej_{I_1}/(V_{CC} \times \tau)$, the base energy cost of the instruction. m represents the "number of cycles" parameter specified in instruction timing tables in microprocessor manuals.

Table 1 is a sample table of CPU base costs for some 486DX2 instructions. The numbers in Column 3 are the base cost in mA per clock cycle. The overall base energy cost of an instruction is the product of the numbers in Columns 3 and 4 and the constants V_{CC} and τ.

Care should be taken in designing the experiments

[1]All instructions are executed in "Real Mode". All registers contain 0, except in entry 11, where CL contains 1. Entry 15 is a "taken" jump while entry 16 is "fall through". Entries 5, 6 and 9 show *normalized* costs [6].

data	0	0F	0FF	0FFF	0FFFF
No. of 1's	0	4	8	12	16
Base Cost	309.5	305.2	300.1	294.2	288.5

Table 2: Base Costs of `MOV BX, data`

used to determine the base costs. The size of the loop has to be large enough to minimize the effects of the branch statement at the bottom of the loop and small enough not to cause any cache misses. Only the target instructions should execute on the CPU during the experiment and thus system effects like multiple time-sharing applications and frequent interrupts cannot be allowed.

3.1.1 Variations in Base Cost

As Table 1 shows, instructions with differing functionality and different addressing modes can have very different energy costs. This is to be expected since different functional blocks are being affected in different ways by these instructions. Within the same family of instructions, there is variability in base costs depending on the value of operands used. For example, consider the `MOV` *register,immediate* family. Use of different *registers* results in insignificant variation since the register file is probably a symmetric structure. Variation in the *immediate* value, however, leads to measurable variation. As an example, Table 2 shows the variation for `MOV BX,`*immediate*. The costs seem to be almost a linear function of the number of 1's in the binary representation of the immediate data – the more the 1's, the lesser the cost. Similarly, for the `ADD` instruction, the base costs are a function of the two numbers being added. The range of variation in all cases, however, is small. It is observed to be about 14, which corresponds to less than a 5% variation.

For instructions involving memory operands, there is a variation in the base cost depending upon the address of the operand. The variation is of two kinds. The first is due to operands that are mis-aligned [1]. Mis-aligned accesses lead to cycle penalties and thus energy penalties that are added to the base cost. Within aligned accesses there is variation in the base cost depending upon the value of the address. For example, for `MOV DX,[BX]`, the base cost can be greater than the cost shown in Table 1 by about 3.5%. This variation is a function of the number of, and position of, 1's in the binary representation of the address.

Given the operand value and address, exact base costs can be obtained through direct measurements. However, these exact values will be of little use since typically a data or address value can be known only at run-time. Thus, from the point of view of program energy cost estimation, the only alternative is to use average base cost values. This is reasonable given that the variation in base costs is small and thus the discrepancy between the average and actual vlaues will be limited.

3.2 Inter-instruction Effects

When sequences of instructions are considered certain inter-instruction effects come into play, which are not reflected in the cost computed solely from base costs. These effects are discussed below.

3.2.1 Effect of Circuit State

The switching activity in a circuit is a function of the present inputs and the previous state of the circuit. Thus, it can be expected that the actual energy cost of executing an instruction in a program may be different from the instruction's base cost. This is because the previous instruction in the given program and in the program used for base cost determination may be different. For example, consider a loop of the following pair of instructions:

```
XOR BX,1
ADD AX,DX
```

The base costs of the `XOR` and `ADD` instructions are 319.2 and 313.6. The expected base cost of the pair, using the individual base costs would be their average, i.e. 316.4, while the actual current is 323.2. It is greater by 6.8. The reason is that the base costs are determined while executing the same instruction again and again. Thus each instruction executes in what we expect is a context of least change. At least, that is what the observations consistently seem to indicate. When a pair of two different instructions is considered, the context is one of greater change. The cost of a pair of instructions is always greater than the base cost of the pair and the difference is termed as the *circuit state overhead.*

As another example, consider the following sequence of instructions. The base cost and the number of cycles of each instruction is listed alongside:

Number	Instruction	Base Cost	Cycles
1	`MOV CX,1`	309.6	1
2	`ADD AX,BX`	313.6	1
3	`ADD DX,8[BX]`	400.2	2
4	`SAL AX,1`	308.3	3
5	`SAL BX,CL`	306.5	3

The measured cost was 332.8 (avg. current over 10 cycles). Using base costs we get

$$(309.6 + 313.6 + 400.2 \times 2 + 308.3 \times 3 + 306.5 \times 3)/10 = 326.8 \quad (1)$$

The circuit state overhead is thus 6.0.

It is possible to get a closer estimate if we consider the circuit state overhead between each pair of consecutive instructions. This is done as follows. Consider a loop of the targeted pair, e.g., instructions 2 and 3.

The estimated cost for the pair is $(2 \times 400.2 + 313.6 \times 1)/3 = 371.3$, while the measured cost is 374.8. Thus, the circuit state overhead is 3.5. Now the overhead occurs twice in every 3 cycles, once between instructions 2&3, and once between 3&2. Since these two different cases cannot be resolved, let us assume that they are the same. Thus, the overhead each time it occurs would be $3.5 \times \frac{3}{2} = 5.25$. Similarly, the overhead between the pairs 1&2, 3&4, 4&5 and 5&1 is found to be 17.9, 12.25, 3.3 and 17.2 respectively. When these overheads are added to the numerator in Equation 1, we get an estimated cost of 332.38, which is within 0.12% of the measured value.

This example illustrates that by determining costs of pairs of instructions, it is possible to improve upon the results of the estimation obtained with base costs alone. However, extensive experiments with pairs of instructions revealed that the circuit state overhead has a limited range - between 5.0 and 30.0 and most frequently occurred in the vicinity of 15.0. This motivates an efficient yet fairly accurate way to account for the circuit state overhead. Calculate the average current for the program using the base costs. Then, add 15.0 to it, to account for circuit state overhead.

A specific manifestation of the effect of circuit state is the effect of switching that occurs on address and data lines. Our experiments revealed that the overall impact of this effect was small. For data reads from the cache, greater switching of the address values led to at most a 2% increase in the the energy cost while for data writes (which go to the cache and the memory bus), the overhead due to greater switching was less than 4%.

The limited variation in the circuit state overhead is contrary to popular belief. In fact, a recent work [3], talks about scheduling instructions to reduce this overhead. But as our experiments reveal, the methods described in this work will not have much impact for the 486DX2. The probable explanation for the limited variation in circuit state overhead is that a major part of the circuit activity in a complex processor like the 486DX2, is common to all instructions, e.g., instruction pre-fetch, pipeline control, clocks etc. While the circuit state may cause significant variation within certain modules, its impact on the overall energy cost is swamped by the much greater common cost. However, we would not like to rule out the impact of circuit state overhead for all processors. It may well be that it is a significant part of the energy consumption in DSPs and processors with complex power management features. An investigation of this issue is the subject of our future study.

3.2.2 Effect of Resource Constraints

Resource constraints in the CPU can lead to stalls e.g. pipeline stalls and write buffer stalls [1, 2]. These can be considered as another kind of inter-instruction effect. They cause an increase in the number of cycles needed to execute a sequence of instructions. For example, a sequence of 120 `MOV DX,[BX]` instructions takes about 164 cycles to execute, instead of 120 due to pre-fetch buffer stalls. While determining the base cost of instructions, it is important to avoid stalls, since they represent a condition that ought not to be reflected in the base cost. Thus, for `MOV DX,[BX]` a sequence consisting of 3 `MOV` instructions followed by a `NOP` is used since there are no stalls during its execution [2]. Knowing the cost of the `NOP` and the measured value for the sequence, the base cost of the `MOV` is determined.

The energy cost of each kind of stall is experimentally determined through experiments that isolate the particular kind of stall. For example, an average cost of 250 per stall cycle was determined for the prefetch buffer stall.

To account for the energy cost of the above stalls during program cost estimation, the number of stall cycles has to be multiplied by the experimentally determined stall energy cost. This product is then added to the base cost of the program. The number of stall cycles is estimated through a traversal of the program code.

3.2.3 Effect of Cache Misses

Another inter-instruction effect is the effect of cache misses. The instruction timings listed in manuals give the cycle count assuming a cache hit. For a cache miss, a certain cycle penalty has to be added to the instruction execution time. Along the same lines, the base costs for instructions with memory operands are determined in the context of cache hits. A cache miss will lead to extra cycles being consumed, which leads to an energy penalty. For experimentation purposes, a cache miss scenario is created by accessing memory addresses in an appropriate order. An average energy penalty of 216 per miss cycle has been experimentally obtained. This has to be multiplied by the average number of miss penalty cycles to get the average energy penalty for one miss. The average penalty multiplied by the cache miss rate is added to the base cost estimate to account for the cache misses during execution of a program.

4 Estimation Framework

In this section we describe a framework for energy estimation of programs using the instruction level power model outlined in the previous section. We start by illustrating this estimation process for the program shown in Table 3. The program has three basic blocks as shown in the figure.[2] The average current per cycle and the number of cycles for each instruction are given along

[2] A basic block is defined as a contiguous section of code with exactly one entry and exit point.

Program	Base Cost(mA)	Cycles
; Block B1		
main:		
mov bp,sp	285.0	1
sub sp,4	309.0	1
mov dx,0	309.8	1
mov word ptr -4[bp],0	404.8	2
;Block B2		
L2:		
mov si,word ptr -4[bp]	433.4	1
add si,si	309.0	1
add si,si	309.0	1
mov bx,dx	285.0	1
mov cx,word ptr _a[si]	433.4	1
add bx,cx	309.0	1
mov si,word ptr _b[si]	433.4	1
add bx,si	309.0	1
mov dx,bx	285.0	1
mov di,word ptr -4[bp]	433.4	1
inc di, 1	297.0	1
mov word ptr -4[bp],di	560.1	1
cmp di,4	313.1	1
jl L2	405.7(356.9)	3(1)
;Block B3		
L1:		
mov word ptr _sum,dx	521.7	1
mov sp,bp	285.0	1
jmp main	403.8	3

Table 3: Illustration of the Estimation Process

with each instruction. With these numbers the base cost of the basic block B1 is 1713.4, B2 is 4709.8, and B3 is 2017.9. B1 is executed once, B2 4 times and B3 once. The `jmp main` statement has been inserted to put the program in an infinite loop. Cost of the `jl L2` statement is not included in the cost of B2 since its cost is different depending on whether the jump is taken or not. It is taken 3 times and not taken once. Multiplying each basic block by the number of times it is executed and adding the cost of the unconditional jump `jl L2`, we get a number proportional to the total energy cost of the program. Dividing it by the estimated number of cycles (72) gives us the cost per cycle (average current) of 369.1. Adding the circuit state overhead offset value of 15.0 we get 384.0. The actual measured average current is 385.0. This program does not have any stalls and thus no further additions to the estimated cost are required. If in the real execution of this program, some cold-start misses are expected, their energy overhead will have to be added.

To validate the estimation model described in the previous section, experiments were conducted with several programs. A close correspondence between the estimated and measured cost was obtained. The estimated cost was typically within 3% of the measured cost.

4.1 Overall Flow

The overall flow of the estimation procedure is shown in Figure 2. Given an assembly or machine level program, it is first split up into basic blocks. The base cost of each instance of the basic block is determined by adding up the base costs of the instructions in the block. These costs are provided in a base cost table. The energy overhead due to pipeline, write buffer and other stalls is estimated for each basic block and added to the basic block cost. Next, the number of times each basic block is executed has to be determined. This depends on the path that the program follows and is dynamic, run-time information that is obtained from a program profiler. Given this information, each basic block is multiplied by the number of times it will be executed. The circuit-state overhead is added to the overall sum at this stage, or alternatively, it could have been determined for each basic block using a table of energy costs for pairs of instructions. An estimated cache penalty is added to get the final estimate. The cache penalty overhead computation needs an estimate of the miss ratio, which is obtained through a cache simulator.

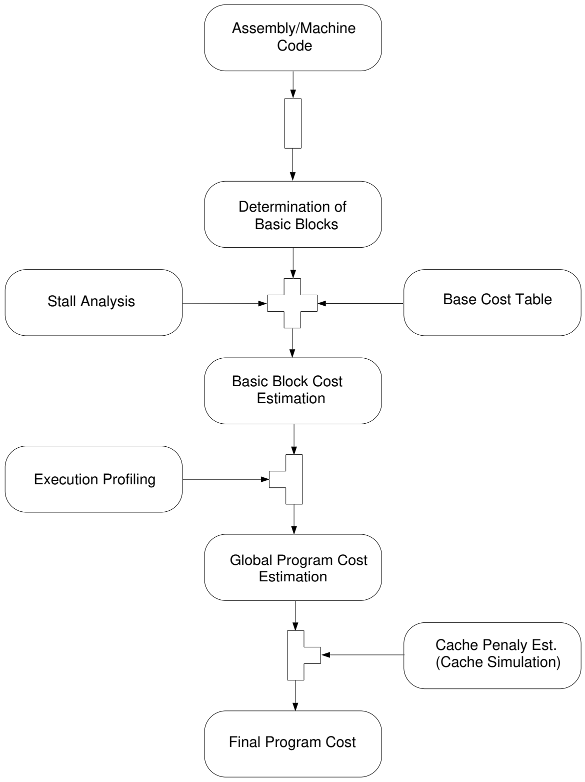

Figure 2: Software Energy Consumption Estimation Methodology

5 Memory System Modeling

The energy consumption in the memory system is also a function of the software being executed. The salient observations regarding the DRAM system current on our experimental setup are briefly described here. Details are provided in a separate reference [6].

The DRAM system draws constant current when no memory access is taking place. This current value was determined to be $77.0mA$ or $5.3mA$, depending on whether page mode was active or not. Greater current is drawn during a memory access. The exact value of this current depends on the address of the present and previous memory access. For example, for writes, the cost of a page hit is 122.8 (for 3 cycles) and that of a miss is 247.8 (for 6 cycles). For page hits, a smaller variation was observed depending on the number of bits that change from the previous address to the present.

Let X be the sum of the energy costs of each individual memory access. Let n and m be the number of memory idle cycles during which the page mode is active and inactive, respectively. The total memory system energy cost is given by $X + 77.0 \times n + 5.3 \times m$. As discussed above, the quantity X depends on the location and sequence of memory accesses made by the program. Along with n and m, this is dynamic, run-time information, which can only be loosely estimated by static analysis. Thus, modeling of memory system energy consumption is difficult if only static analysis is used. However, as the above discussion shows, analysis of this consumption is feasible. This is significant, given that for systems with tight energy budgets, it is important to understand all sources of energy consumption.

6 Software Power Optimization

In recent years, there has been a spurt of research activity targeted at reducing the energy consumption in systems. This research, however, has by and large not recognized the potential energy savings achievable through optimization of software. This was mainly due to the lack of practical techniques for analyzing the energy consumption of programs. This deficiency has been alleviated by the measurement and estimation methodology described in the previous sections. This methodology makes it possible to compare and evaluate programs in terms of their energy consumption and also to study the effect of compilation on the energy consumption of programs.

Using the results of this work, several possible avenues for energy reduction through code restructing and compilation have been studied [5]. Examples with energy reduction of up to 40% on the 486DX2 based system, obtained by rewriting code, demonstrate the potential of these ideas. These ideas will be pursued further as part of the research in the area of software power optimization.

7 Analysis of SPARClite 934

The previous sections describe the application of the power analysis methodology for the 486DX2, a CISC processor. To verify the general applicability of this methodology, it was decided to apply the methodology to a processor with a different architectural style. The Fujitsu SPARClite 934, a RISC processor targeted for embedded applications was chosen for this purpose. A power analysis of this processor has been performed using the measurement and experimentation techniques described in the previous sections. The basic model of a base energy cost per instruction, enhanced by the inter-instruction effects remains valid for this processor, though the actual costs differ in value. The details of this analysis are described in a separate reference [4].

8 Summary and Future Work

This paper presents a methodology for analyzing the energy consumption of embedded software. It is based on an instruction level model that quantifies the energy cost of individual instructions and of the various inter-instruction effects. The motivation for the analysis methodology is three-fold. It provides insights into the energy consumption in processors. It can be used to help verify if an embedded design meets its energy constraints and it can also be used to guide the design of embedded software such that it meets these constraints.

The methodology has so far been applied to two commercial processors, a CISC and a RISC. Future work will extend this to other architecture styles, to characterize and contrast their energy consumption models. DSPs, superscalar processors and processors with internal power management will be considered.

9 Acknowledgements

We would like to thank Deo Singh, Suresh Rajgopal and Tom Rossi of Intel Corp. for providing us with the 486DX2 evaluation board, Mike Tien-Chien Lee, Masahiro Fujita and Dinesh Maheshwari of Fujitsu for helping make the SPARClite analysis possible and Dan Markham and Pete Derosa of Princeton Univ. for assistance in collecting the experimental data.

References

[1] Intel Corp. *i486 Microprocessor, Hardware Reference Manual*, 1990.

[2] Intel Corp. *Intel486 Microprocessor Family, Programmer's Reference Manual*, 1992.

[3] C. L. Su, C. Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *IEEE COMPCON*, Feb. 1994.

[4] V. Tiwari, T.C. Lee, M. Fujita, and D. Maheshwari. Power analysis of the SPARClite MB86934. Technical Report FLA-CAD-94-01, Fujitsu Labs of America, August 1994.

[5] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the 1994 Symposium on Low Power Electronics*, October 1994.

[6] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of the Intel 486DX2. Technical Report CE-M94-5, Princeton Univ., Dept. of Elect. Eng., June 1994.