437

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

Vivek Tiwari, Sharad Malik, and Andrew Wolfe

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. This can help in verifying if a design meets its specified power constraints. Further, it can also be used to search the design space in software power optimization. Examples with power reduction of up to 40%, obtained by rewriting code using the information provided by the instruction level power model, illustrate the potential of this idea.

I. 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. A notable example is automobile electronics where embedded processors control each aspect of the efficiency, comfort and safety of the new generation of cars. The second factor leading to their growth is the increasing migration from application specific logic to application specific code running on existing processors. This in turn is driven by two distinct forces. The first is the increasing cost of setting up and maintaining a fabrication line. At over a billion dollars for a new line, the only components that make this affordable are high volume parts such as processors, memories and possibly FPGA's. Application specific logic is getting increasingly expensive to manufacture and is the solution only when speed constraints rule out programmable alternatives. The second force comes from the application houses, which are facing increased pressures to reduce the time to market as well as

to have predictable schedules. Both of these can be better met with software programmable solutions made possible by embedded systems. 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. This has led to a significant research effort in power estimation and low power design. However, 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 to a limited extent at the logic level. At the least these are very slow and impractical to use to evaluate the power consumption of embedded 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 (DSP's). The second is in the form of embedded cores which can be incorporated in a larger silicon chip along with program/data memory and other dedicated logic. In the first case, the processor information available to the designer is whatever the manufacturer cares to make 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 of this research is to present a methodology for developing and validating an instruction level power model for any given processor. 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. 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 [9]. The ability to evaluate software in terms of the power metric helps in verifying if a design meets its specified power constraints. In addition, it can also be used to search the design space in software power optimization. Examples with power reduction of up to 40% on the 486DX2, obtained by rewriting code using the information provided by

the instruction level power model, illustrate the potential of this idea.

II. EXPERIMENTAL METHOD

The power consumption in microprocessors has been a subject of intense study lately. Attempts to model the power consumption in processors often adopt a "bottom-up" approach. Using detailed physical layouts and sophisticated power analysis tools, isolated power models are built for each of the internal modules of the processor. The total power consumption of the processor is then estimated using these individual models. No systematic attempt, however, has been made to relate the power consumption of the processor to the software that executes on it. Thus, while it is generally 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. This is also the reason why the potential for power reduction through modification of software is so far unknown and unexploited. The goal of our work is to overcome these deficiencies by developing a methodology that would provide a means for analyzing the power consumption of a processor as it executes a given program. We want to provide a method that makes it possible to talk about the "power/energy cost of a given program on a given processor." This would make it possible to very accurately evaluate the power cost of the programmable part of an embedded system.

We propose the following hypothesis that forms the basis for meeting the above goal: *By measuring the current drawn by the processor as it repeatedly executes certain instructions or certain short instruction sequences, it is possible to obtain most of the information that is needed to evaluate the power cost of a program for that processor.*

The intuition that guides this hypothesis is as follows: Modern microprocessors are extremely complex systems consisting of several interacting functional blocks. However, this internal complexity is hidden behind a simple interface—the instruction set. Thus to model the energy consumption of this complex system, it seems intuitive to consider individual instructions. Further, each instruction involves specific processing across various units of the processor. This can result in circuit activity that is characteristic of each instruction and can vary with instructions. If a given instruction is executed repeatedly, then the power consumed by the processor can be thought of as the power cost of that instruction. In a given program, certain inter-instruction effects also occur, such as the effect of circuit state, pipeline stalls and cache misses. Repeatedly executing certain instruction sequences during which these effects occur may provide a way to isolate the power cost of these effects. Thus the sum of the power costs of the each instruction that is executed in a program enhanced by the power cost of the inter-instruction effects can be an estimate for the power cost of the program.

The above hypothesis, however, is of no use until it is validated. We have empirically validated the hypothesis for two commercial microprocessors using actual physical measurements of the current drawn by them. The validation of the

hypothesis, and based on it, the derivation of the parameters of an instruction level power model for the Intel 486DX2, is the subject of the next few sections.

Given that the above hypothesis has been validated for two processors using physical measurements, there is an alternative way for deriving the parameters of the instruction level power model. Instead of physically measuring the current drawn by the CPU, it can be estimated using accurate, simulation based power analysis tools. The execution of the given instruction/instruction sequence is simulated on lower level (circuit or layout) models of the CPU, and the power analysis tool provides an estimate of the current drawn. The advantage of this method is that since detailed internal information of the CPU is available, it may be possible to relate the power cost of the instructions to the micro-architecture of the CPU. This could provide cues to the CPU designer for optimizing the designs for low power.

However, in the case of embedded system design, detailed layout information of the CPU is often not available to the designer of the system. Even if it is available, the simulation based tools and techniques are expensive and difficult to apply. A methodology based on laboratory measurements, like the one described below, is inexpensive and practical, and often may be the only option available. Given a 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 specifics of the measurement methodology are described next.

A. 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, as has been done in the above discussion. However it is important to distinguish between the two 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 terms 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.

B. Current Measurement

For this study, the processor used was a 40 MHz Intel 486DX2-S Series CPU. The CPU was part of a mobile personal computer evaluation board with 4 MB 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. Execution time of programs was measured through detection of specific bus states using a logic analyzer.

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 (100 ms), 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 3.3 V and τ was 25 ns, corresponding to the 40 MHz internal frequency of the CPU. Thus, if the average current for an instruction sequence is I A, and the number of cycles it takes to execute is N, the energy cost of the sequence is given by: $E = I \times V_{CC} \times N \times \tau$, which equals: $(8.25 \times 10^{-8} \times I \times N)$ J. Throughout the rest of the paper, in order to specify the energy cost of an instruction (instruction sequence), the average current will be specified. The number of cycles will either be explicitly specified, or will be clear from the context.

III. INSTRUCTION LEVEL MODELING

Based on the hypothesis described in Section II, an instruction level energy model has been developed and validated for the 486DX2. 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.
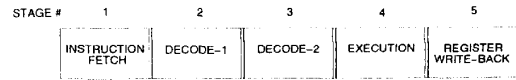

Fig. 1. Internal pipelining in the 486DX2

The instruction-level energy model described here is based on actual measurements and evolved as a result of extensive experimentation. The various components of this model are described in the subsections below.

A. 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 II.

While this method seems intuitive if the CPU executes only one instruction at a given time, most modern CPU's, including the 486DX2, process 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 Fig. 1 [6]. 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: $E_{\text{cycle}} = E1_{I_1} + E2_{I_2} + E3_{I_3} + E4_{I_4} + E5_{I_5}$. On the other hand, the total energy consumed by a given instruction I_1, as it moves through the various stages is: $E_{\text{ins}} = \sum_j Ej_{I_1}$. 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 $E_{\text{cycle}} = E_{\text{ins}}$, 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.

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. Energy consumption in any of the stalled stages can be considered as a part of $E4_{I_1}$. 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 I
Subset of the Base Cost Table for the 486DX2[1]

Number	Instruction	Current (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 II
Base Costs of MOV BX, Data

data	0	0F	0FF	0FFF	0FFFF
No. of 1's	0	4	8	12	16
Current(mA)	309.5	305.2	300.1	294.2	288.5

Table I is a sample table of CPU base costs for some 486DX2 instructions. The numbers in Column 3 are the observed average current values. 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. A rigorous confidence interval was not determined for the current measurement apparatus. However, it was observed that repeated runs of an experiment at different times resulted in only a very small variation in the observed average current values. The variation was in the range of ± 1 mA.

Care should be taken in designing the experiments used to determine the base costs. The loops that are used to determine the base costs of instructions have to satisfy certain size constraints. As more of the target instructions are put in the loop, the impact of the branch statement at the bottom of the loop is minimized. The observed current value thus converges with increasing loop size. Thus, the loop size should be large enough in order to obtain the converged value. Very large loops, on the other hand, may cause cache misses, which are undesirable during determination of base costs. A loop size of 120, which satisfies both the above constraints, was chosen. Only the target instructions should execute on the CPU during experiments, and thus system effects like multiple time-sharing applications and interrupts cannot be allowed during the experiments.

Variations in Base Cost: As Table I 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

[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 [11].

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. For example, 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. Table II illustrates this through some sample values. 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 mA, 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 misaligned [6]. 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 I 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 values will be limited.

B. 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.

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 mA and 313.6 mA. The expected base cost of the pair, using the individual base costs would be their average, i.e., 316.4

TABLE III
An Example Instruction Sequence

Number	Instruction	Current(mA)	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

mA, while the actual cost is 323.2 mA. It is greater by 6.8 mA. 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 sequence of instructions shown in Table III. The current cost and the number of cycles of each instruction is listed alongside. The measured cost for this sequence is 332.8 mA (avg. current over 10 cycles). Using base costs we get:

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

The circuit state overhead is thus 6.0 mA.

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 $(2 \times 400.2 + 313.6 \times 1)/3 = 371.3$ mA, while the measured cost is 374.8 mA. Thus, the circuit state overhead is 3.5 mA. 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 $3.5 \times \frac{3}{2} = 5.25$ mA. Similarly, the overhead between the pairs 1 and 2, 3 and 4, 4 and 5, and 5 and 1 is found to be 17.9, 12.25, 3.3, and 17.2 mA, respectively. When these overheads are added to the numerator in (1), we get an estimated cost of 332.38 mA, 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 mA and 30.0 mA and most frequently occurred in the vicinity of 15.0 mA. 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 mA 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 back-to-back data reads from the cache, greater switching of the address values led to at most a 3% increase in the energy cost. For back-to-back data writes (which go to both the cache and the memory bus since the cache is write-through), the impact of greater switching of the address values was less than 5%.

The limited variation in the circuit state overhead is contrary to popular belief. In fact, a recent work [8], 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 the case that it is a significant part of the energy consumption in processors like RISC's (Reduced Instruction Set Computers) DSP's, and processors with complex power management features. An investigation of this issue is the subject of our future study.

Effect of Resource Constraints: Resource constraints in the CPU can lead to stalls e.g. pipeline stalls and write buffer stalls [6], [7]. 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 prefetch 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 [7]. 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 mA for stall cycles was determined for the prefetch buffer stall.

It has been observed that the cost of stalls can show some variation depending upon the instructions involved in the stall. Through extensive experimentation it may be possible to subdivide each stall type into specific cases and to assign a cost to each case. However, in general, the use of a single average cost value for each stall type suffices.

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.

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 penalty of 216 mA for cache miss cycles 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.

IV. 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 IV. The program has three basic blocks as shown in the figure[2]. The average current and the number of cycles for each instruction are provided in two separate columns. For each basic block, the two columns are multiplied and the products are summed up over all instructions in the basic block. This yields a value that is proportional to the base energy cost of one instance of the basic block. The values are 1713.4, 4709.8, and 2017.9, for B1, B2, and B3, respectively. 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 the base cost of 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 an average current of 369.1 mA. Adding the circuit state overhead offset value of 15.0 mA we get 384.0 mA. The actual measured average current is 385.0 mA. 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 cache 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. It was observed that the main reasons for the discrepancy in the estimated and actual cost are as follows: First, for instructions that require operands, the operand values and addresses are often not known until runtime. Thus, average base costs may have to be used instead of exact costs. Second, the circuit state overhead for pairs of consecutive instructions in the program may differ from the default value used. Third,

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

TABLE IV
ILLUSTRATION OF THE ESTIMATION PROCESS

Program	Current(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

the penalty due to stalls and cache misses is difficult to predict statically. As discussed in Section III, the first two effects are limited in their impact on the overall cost. The inability to predict the penalty due to stalls and cache misses, on the other hand, can potentially have a greater impact on the accuracy of the estimate. However, for programs with no stalls and cache misses, the maximum difference between the estimated and the measured cost was less than 3% of the measured cost.

A. Overall Flow

The overall flow of the estimation procedure is shown in Fig. 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
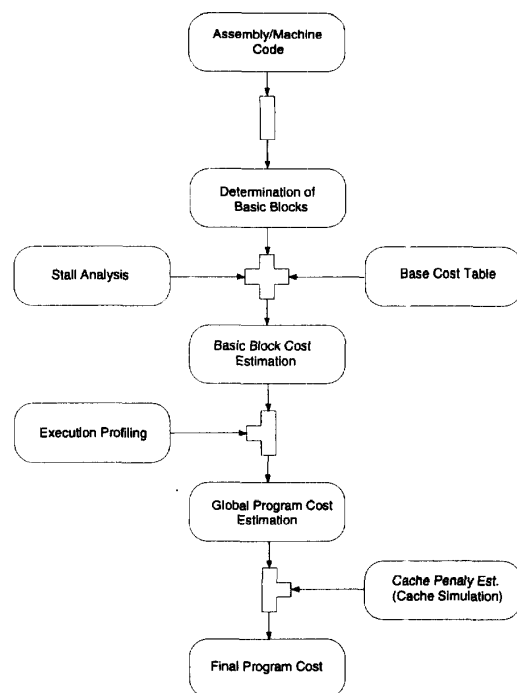
Fig. 2. Software energy consumption estimation methodology.

computation needs an estimate of the miss ratio, which is obtained through a cache simulator.

V. 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 [11].

The DRAM system draws constant current when no memory access is taking place. This current value was determined to be 77.0 mA or 5.3 mA, depending on whether page mode was active or not.[3] 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, when both the previous and the present access map to the same page, i.e., for a page hit, the cost is 122.8 mA (for 3 cycles including 1 wait state). For a page miss, the cost is 247.8 mA (for 6 cycles including 4 wait states). 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, $(X + (77.0 \times n + 5.3 \times m) \times 8.25 \times 10^{-8})J$. 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

[3] "Page mode active" refers to the condition when the row address has been latched and the Row Address Strobe (RAS) signal is active.

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.

VI. 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. The growing trend towards tight energy budgets necessitates identification and exploration of every possible source of energy reduction, forcing us to examine the design of energy efficient software.

The energy formula described in Section II-A shows that the energy cost of a program is proportional to the product of the average current and the running time of the program. Thus, the value of this product has to be reduced in order to reduce the energy cost. This section examines some alternatives for this, in the context of the 486DX2, using the results of the instruction level analysis that was described earlier.

A. Instruction Reordering

A recent work [8] presents a technique for scheduling instructions on an experimental RISC processor in such a way that the switching on the control path is minimized. In terms of the energy formula, this technique is trying to reduce the average current for the program through instruction reordering.

Our experiments based on actual energy measurements on the 486DX2, however, reveal that this technique does not translate into very significant overall energy reduction. This technique is trying to reduce what we termed as the circuit state overhead. As we saw earlier, this quantity is bounded by a small range and does not show a great amount of variation. In fact, it was observed that different reorderings of several sequences of instructions showed a variation of only up to 2% in their current cost. It can be concluded that this technique is not very effective for the 486DX2. However, its effectiveness on other architectures and processors should be investigated further.

B. Generation of Energy Efficient Code

While reordering of a given set of instructions in a piece of code may have only a limited impact on the energy cost, the actual choice of instructions in the generated code can significantly affect the cost. As a specific example, an inspection of the energy costs of 486DX2 instructions reveals that instructions with memory operands have very high average current compared to instructions with register operands. Instructions using only register operands cost in the vicinity of 300 mA. Memory reads that hit the cache cost upwards of 430

TABLE V
RESULTS OF ENERGY OPTIMIZATION OF SORT AND CIRCLE

Program	hlcc.asm	hht1.asm	hht2.asm	hht3.asm
Avg. Current (mA)	525.7	534.2	507.6	486.6
Execution Time (μsec)	11.02	9.37	8.73	7.07
Energy ($10^{-6}J$)	19.12	16.52	14.62	11.35
Program	clcc.asm	cht1.asm	cht2.asm	cht3.asm
Avg. Current (mA)	530.2	527.9	516.3	514.8
Execution Time (μsec)	7.18	5.88	5.08	4.93
Energy ($10^{-6}J$)	12.56	10.24	8.65	8.37

mA. Memory writes cost upwards of 530 mA and also incur a memory system current cost since the cache is write-through. Thus, reduction in the number of memory operands can lead to a reduction in average current.

The reduction in energy cost, i.e., in the product of average current and running time would be greater still, since use of memory operands incurs more cycles. For example, ADD DX, [BX] takes two cycles, even in the case of a cache hit, while ADD DX, BX takes just one cycle. Potential pipeline stalls, misaligned accesses, and cache misses, further add to the running time. Reduction in number of memory operands can be achieved by adopting suitable code generation policies, e.g. saving the least amount of context during function calls. However, the most effective way of reducing memory operands is through better utilization of registers. This entails techniques akin to optimal global register allocation of temporaries and frequently used variables [1] [2].

The impact of the above ideas on the energy cost of programs is illustrated here using examples. The first program considered is a *heapsort* program in C, called "sort" [3]. hlcc.asm is the assembly code for this program generated by lcc, an ANSI C compiler [4]. The sum of the observed average CPU and memory currents is given in the table above. The program execution times and overall energy costs are also reported. lcc is a general purpose compiler and while it produces good code, it leaves room for further improvement of running time. Hand tuning of the code for shorter running time (hht1) leads to a 15% reduction in running time. The average current goes up a little since of all the instructions that were eliminated, a greater proportion had lower average currents. However, due to the reduction in running time, the overall energy cost goes down by 13.5%. So far only temporary variables had been allocated to registers. In hht2, 3 local variables are allocated to registers and the appropriate memory operands are replaced by register operands. Even though redundant instructions are not removed, there is a 5% reduction in current and a 7% reduction in running time. In hht3, 2 more local variables are allocated to registers and all redundant instructions are removed. Compared to hlcc, hht3 has 40.6% lower energy consumption. Results for another program derived from the circle program [5] are also shown in Table V. Significant energy reduction, about 33%, are observed for this program too.

The specific optimizations used in the above examples were prompted by the results of the instruction level analysis of the 486DX2. They are discussed in greater detail in [10]. In general, the ideas used for energy efficient code for one processor may not hold for another. An instruction level analysis, using the methodology described earlier, should therefore be performed for each processor under consideration. That methodology provides a way for assigning energy costs to instructions. The idea behind energy driven code generation is to select instructions using these costs, such that the overall energy cost of a program is minimized. An investigation of this issue for different architectural styles will be pursued further as part of research in the area of software power optimization.

VII. 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 [9].

VIII. 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 threefold. 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. Initial attempts at code re-writing demonstrate significant power reductions—justifying the motivation for such a power analysis technique.

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. Finally, we hope to use this analysis in automatic techniques for the reduction of power consumption in embedded software.

of AT&T Bell Labs and D. Hanson of Princeton University for the 486 code generator.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*. Reading, MA: Addison Wesley, 1988.

[2] M. Benitez and J. Davidson, "A retargetable integrated code improver," *Tech. Rep. CS-93-64*, Univ. of Virginia, Dept of Computer Sci., Nov. 1993.

[3] Press *et al.*, *Numerical Recipes in C*. Cambridge, MA: Cambridge Univ., 1988.

[4] C. W. Fraser and D. R. Hanson, "A retargetable compiler for ANSI C," *SIGPLAN Notices*, pp. 29–43, Oct., 1991.

[5] R. Gupta, "Co-synthesis of hardware and software for digital embedded systems," Ph.D. dissertation, Dept. of Electrical Eng., Stanford University, CA, 1993.

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

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

[8] 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.

[9] V. Tiwari, T. C. Lee, M. Fujita, and D. Maheshwari, "Power analysis of the SPARClite MB86934," *Tech. Rep. FLA-CAD-94-01*, Fujitsu Labs of America, Aug. 1994.

[10] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. 1994 Symp. Low Power Electron.*, Oct. 1994.

[11] V. Tiwari, S. Malik, A. Wolfe, "Power analysis of the Intel 486DX2," *Tech. Rep. CE-M94-5*, Princeton Univ., Dept. of Elect. Eng., June, 1994.

Vivek Tiwari received the B. Tech degree in computer science and engineering from the Indian Institute of Technology, New Delhi, India in 1991. Currently he is working towards the Ph.D. degree in the Department of Electrical Engineering, Princeton University, Princeton, NJ.

His research interests are in the areas of computer aided design of VLSI and embedded systems and in microprocessor architecture. The focus of his current research is on tools and techniques for power estimation and low power design. He has held summer positions at NEC Research Labs, Intel Corporation, and Fujitsu Labs of America, in 1992, 1993, and 1994, respectively, where he worked on the above topics.

Mr. Tiwari received the IBM Graduate Fellowship award in 1993 and 1994.

Sharad Malik received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, India in 1985 and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley in 1987 and 1990 respectively.

Currently he is an Assistant Professor with the Department of Electrical Engineering, Princeton University. His current research interests are in the synthesis and verification of digital systems.

Dr. Malik has received the President of India's Gold Medal for academic excellence (1985), the IBM Faculty Development Award (1991), an NSF Research Initiation Award (1992), a Best Paper Award at the IEEE International Conference on Computer Design (1992), the Princeton University Engineering Council Excellence in Teaching Award (1993, 1994), the Walter C. Johnson Prize for Teaching Excellence (1993), Princeton University Rheinstein Faculty Award (1994), and the NSF Young Investigator Award (1994).
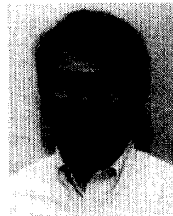
Andrew Wolfe received the B.S.E.E. from the Johns Hopkins University in 1985 and the M.S. and Ph.D. degrees from Carnegie Mellon University in 1987 and 1992, respectively. His doctoral dissertation introduced a new model for instruction-level parallel processor architecture called XIMD.

He was a Semiconductor Research Corporation Fellow from 1986 to 1991. He has also worked as a processor designer at ESL/TRW in Sunnyvale, CA and as a product design consultant for numerous companies. Since 1991, he has been an Assistant Professor at Princeton University. His current research interests include embedded systems architectures and design tools, instruction-level parallelism, and video-signal processors.

Dr. Wolfe has served as General Chair of Micro-26 and as Program Chair of Micro-24 as well as serving on the technical committees of several ICCD and Micro conferences. He has presented tutorials on instruction-level parallelism at ASPLOS V and ISCA 20 as well as a tutorial on embedded systems at ICCD '93.