Instruction Level and Operating System Profiling for Energy Exposed Software

Amit Sinha, Nathan Ickes, and Anantha P. Chandrakasan

***Abstract*—Energy conscious software design can significantly improve the energy efficiency of a portable system. A software energy estimation technique using instruction class profiling is presented. The technique is shown to have an estimation error of less than 3% with trivial runtime overhead, based on a set of application programs evaluated on the StrongARM SA-1100 and Hitachi SH-4 microprocessors. A technique to isolate the switching and leakage energy components of software is outlined. The energy overhead of a real-time operating system is also profiled. The overall impact of system-level software energy management is quantified using the MIT μAMPS system as an application example.**

***Index Terms*—Instruction current profiling, low power, software energy, system-level energy management.**

I. Introduction

ENERGY efficient system design is becoming increasingly important with the proliferation of portable, battery-operated appliances such as laptops, personal digital assistants (PDAs), cellular phones, MP3 players, etc. The energy constraints on portable devices are becoming increasingly tight as complexity and performance requirements continue to be pushed by user demand. Incredible computational power is being packed into mobile devices these days. Moore's law has resulted in processor speeds being doubled approximately every 18 mo [1]. There has also been a corresponding increase in power consumption in processors. In fact, microprocessor power consumption has gone up from under 1 W to over 50 W over the last 20 years. As the demand for portable electronics increased, several low-power processors entered the market. Most of the power savings in these low-power processors comes from three sources: 1) smart circuit design, (a good overview of such techniques is present in [2]); 2) throwing away lesser used functionality [3] i.e., architectural trimming; and, 3) voltage scaling and clock gating [16], [25].

It has been shown in separate applications that dedicated hardware implementations can out perform general-purpose microprocessors/digital signal processings (DSPs) by several orders of magnitude in terms of energy consumption [4], [5]. However, dedicated implementations are not always feasible. Application specific integrated circuits (ASICs) are getting increasingly expensive to manufacture and are feasible only when speed and volume constraints are overbearing. Apart from reduction of the prohibitive cost of ASIC solutions, software systems offer significant other advantages over hardwired solutions such as flexibility for changing requirements and standards. The possibility of a field upgrade and lesser time-to-market offered by software solutions has critical economic consequences on the product cycle. The breaking of the $5 threshold for 32-b processors has resulted in an explosion in the use of general-purpose microprocessors and DSPs in high-volume embedded applications [6].

While it is true that maximum power savings are possible through hardware optimizations, it is the software that finally runs on the hardware and smart software design can yield substantial power reduction. The present day software engineer has little or no knowledge of the energy efficiency of his application code. Just as performance tuning tools [7] and feedback compiler optimization techniques have been proposed and used to improve the performance of various applications, it is possible to envision an integrated development environment that provides application energy consumption feedback to the software engineer. We define such a framework as an "energy exposed software" development environment and a conceptual flow within the framework is illustrated in Fig. 1. Energy models for the target hardware platform are embedded in the application development environment along with the energy cost of various operating system and library function calls. When an algorithm is developed in a high-level language and compiled, its energy cost is also profiled along with other standard parameters such as code size. Runtime information is required for energy estimation. Feedback from profiling tools in conjunction with energy models could be used to identify energy intensive code sections for the programmer. The energy efficiency of the program could then be improved iteratively or by smart compiler optimizations. In this paper, we have proposed energy models for software and the operating system based on the StrongARM SA-1100 and Hitachi SH-4 processors. A web based tool that implements these models in the framework of Fig. 1 is also described.

II. Related Work

There has been significant work in the field of software power estimation and optimization. Instruction level power estimation tools have been proposed to compute the energy consumption of software [11]–[13]. The basic idea is to

Manuscript received September 18, 2003; revised February 14, 2002. This work was supported in part by the Defense Advanced Research Project Agency (DARPA) Power-Aware Computing/Communication Program and in part by the Air Force Research Laboratory, Air Force Materiel Command, under Contract F30602-00-2-0551.

A. Sinha was with the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA 02139 USA. He is now with Engim, Inc., Acton, MA 01720 USA (e-mail: sinha@alum.mit.edu).

N. Ickes and A. P. Chandrakasan are with the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: nickes@mtl.mit.edu; anantha@mtl.mit.edu).

Digital Object Identifier 10.1109/TVLSI.2003.819569

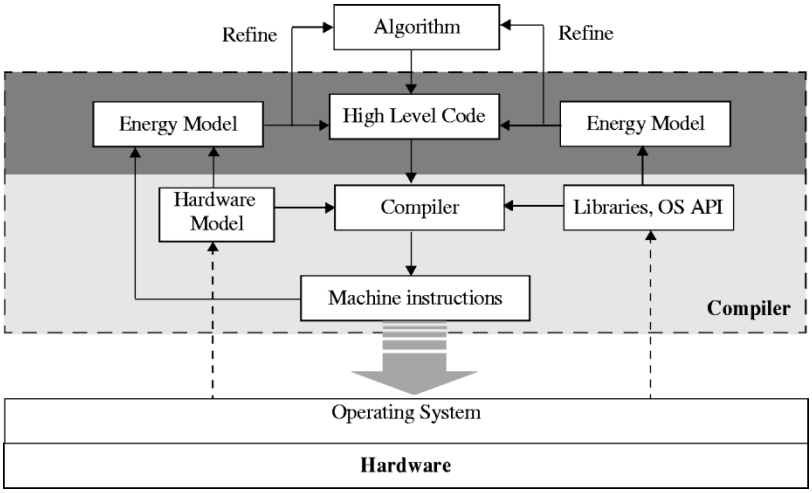

Fig. 1. Energy exposed application development environment.

run each instruction or short sequences of instructions in a loop and measure the average current/power consumption. In [15], the authors perform a microanalysis of instruction level power consumption for a superscalar pipelined processor using dedicated hardware to collect instruction statistics. The primary drawback of estimation techniques based solely on instruction level profiling is that these tools rely on exhaustive characterization of the energy consumption of the entire instruction set architecture (ISA) and interinstruction effects. The estimation model is compute intensive, requiring a complete trace analysis of the program's instructions and is therefore slow. In [14], the authors propose energy estimation using function-level macro-modeling to improve the estimation speed over instruction-level profiling. Their results indicate one-two orders of magnitude speedup based on characterization done on the Fijitsu SPARCLite and SimpleScalar processors.

System level energy management policies have also been widely researched. The authors of [17] have implemented and compared different policies on a laptop computer. Their experiments indicate that policy selection is a tradeoff between power saving, performance, interactivity, and resource requirements. In [26], dynamic power management in a wireless sensor network system is studied. It is shown that fine-grained shutdown can lead to scalable energy-performance tradeoffs. Dynamic voltage and frequency scaling has been proposed and demonstrated as an effective tool for system energy reduction [25], [16].

In this paper, we have outlined a software energy estimation technique that estimates the energy consumption of software with trivial runtime overhead. The significant difference between our technique and others published in literature is two fold. 1) Our technique explicitly computes both switching and leakage energy components of software. Leakage currents are increasing exponentially in digital systems and this distinction is important for optimum policy decisions in standby states of software. 2) Our technique does not require elaborate instruction trace analysis. It is based on a few instruction class statistics that can be collected at runtime with almost zero overhead while maintaining an average energy estimation error of less than 3%. Our estimation technique is verified on the StrongARM SA-1100 and Hitachi SH-4 processors and is described in Section III. We have incorporated our techniques into a web based tool for the research community to explore and test [8], [9]. The tool can just as easily be a part of an integrated development environment (IDE). This tool has been designed as part of the ongoing MIT μAMPS Project, which is developing power-aware techniques for wireless sensor networks [10]. We have used the μAMPS sensor node as an example application. In several embedded systems, a substantial portion of the executing code is comprised of various operating system (OS) function calls. As such it is critical to expose the energy cost of such calls within the IDE. In Section IV we have characterized the energy cost of a popular embedded OS used in the μAMPS application. The OS energy models in conjunction with application code energy estimation have been used to significantly improve the energy efficiency of the μAMPS sensor node. The overall system level energy savings for the μAMPS sensor node have been quantified in Section V.

III. Software Energy Estimation

Our experiments on the StrongARM SA-1100 [18] and Hitachi SH-4 [19] microprocessors, two popular low-power embedded processors, show that the variation in the current consumption across instructions is quite small. A lot of overheads are common across instructions and as a result the overall current consumption of a program is a weak function of the actual instruction stream and, to a first order, depends only on the operating frequency and voltage. Second-order variations do exist but were measured to be less than 7% for a set of benchmark programs. Therefore, a complete instruction level trace analysis is unnecessary and a simple cycle accurate simulation can be used to estimate execution time and basic statistics. We propose a simple fast technique to estimate software energy and estimate second-order variations. Our experiments indicate that our proposed model has accuracy within 3% of actual measurements.

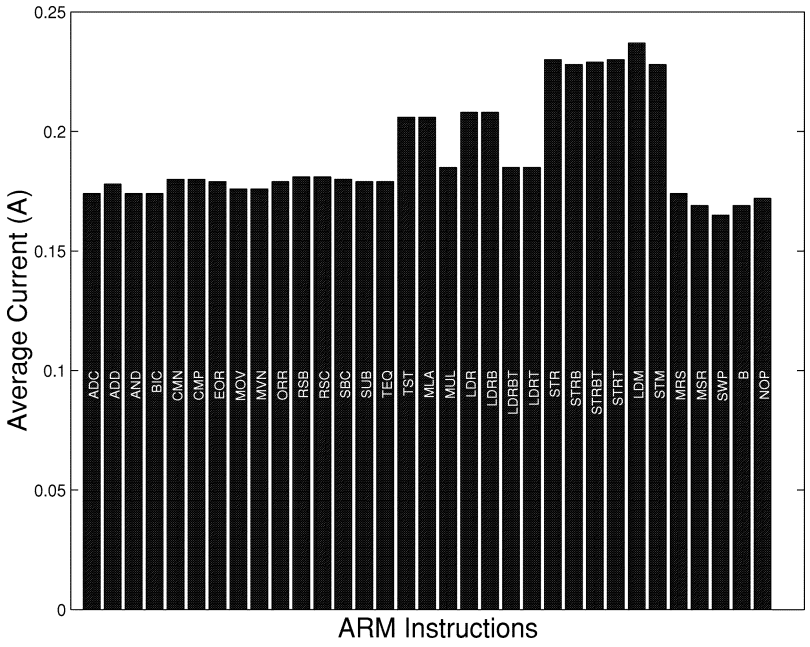

Fig. 2. StrongARM SA-1100 instruction set current consumption.

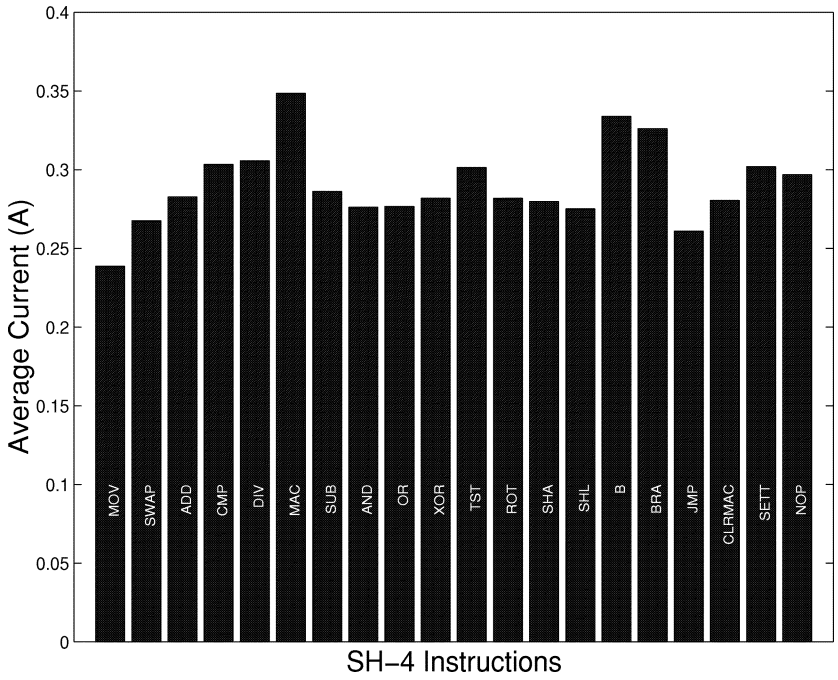

Fig. 3. Hitachi SH-4 instruction set current consumption.

A. Instruction Energy Profiling

Our experimental setup consisted of the Brutus SA-1100 Design Verification Platform, which is essentially the StrongARM SA-1100 microprocessor connected to a PC using a serial link. It can operate from 59–206 MHz, with a corresponding core supply voltage of 0.8–1.5 V. The power supply to the StrongARM core was provided externally through a variable voltage source with the I/O pads running at a fixed supply voltage. The ARM project manager (APM) was used to debug, compile, and execute software on the StrongARM. Current measurements were performed using the ammeter built into the variable power supply. The instruction and data caches were enabled before the programs were executed. To measure the current that is drawn by a subroutine, the subroutine was placed inside a loop with multiple iterations until a stable value of current was measured.

Fig. 2 shows the current consumption of all the instructions of the ARM instruction set on the SA-1100. Each of the 33 current values are themselves the averages of the various addressing modes and inputs in which the instruction can be executed accounting for a total of about 280 data points. The important point to observe is that the current consumption is

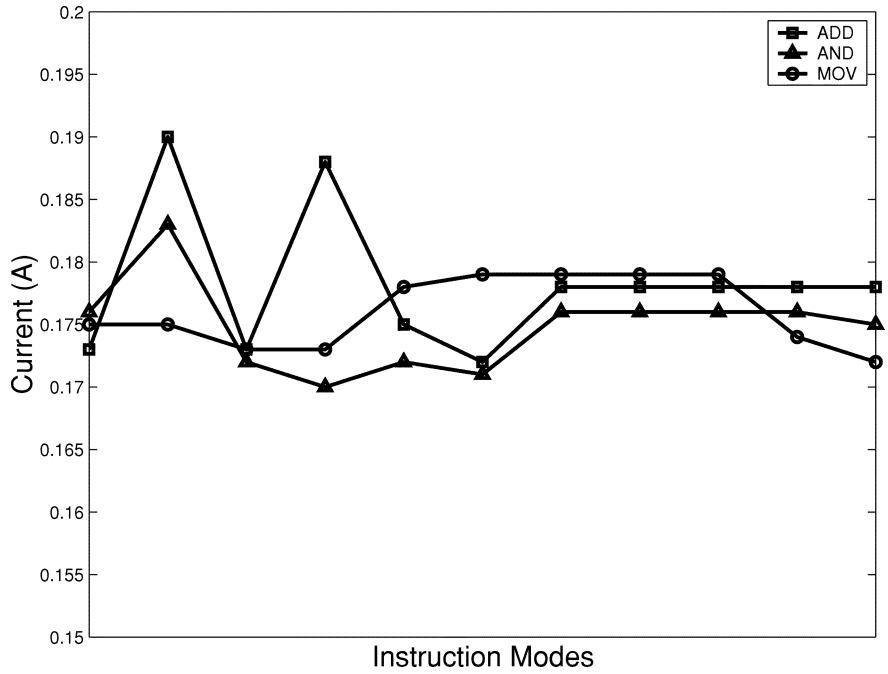

Fig. 4. Current variations within instructions of the StrongARM.

quite uniform. On an average, arithmetic and logical instructions consume 0.178 A, multiplies 0.196 A, loads 0.196 A, stores 0.229 A, while the other instructions consume about 0.170 A. The total variation in current consumption is 0.072 A, which is 38% of the overall average current consumption. Fig. 3 shows the instruction current consumption, measured using a similar setup, for the Hitachi SH-4 processor running at 2.0 V core power supply. The average instruction current is 0.29 A, with a variation of 0.11 A, which once again is 38% of the average. The current variation within an instruction (as a function of addressing modes and data) is even smaller. Fig. 4 shows the current consumption of the StrongARM for three different instructions as a function of various addressing modes and data. In general, we observed that to a first order, the instruction current consumptions are independent of the addressing modes or operands.

Based on the previous discussion, it is reasonable to conclude that the common overheads (such as caches, decode logic, etc.) in contemporary microprocessors are large and overshadow any instruction specific variations. Therefore, estimating software energy consumption with an elaborate instruction trace and interinstruction analysis may not be required for estimating the energy consumption of software to a reasonable accuracy.

B. Software Energy Profiling

1) First-Order Model: While the instruction level current consumption has a variation of about 38%, the variation of the current consumption in programs is much less. Fig. 5 shows the current consumption of six different benchmark programs at different supply voltage and frequency levels in the StrongARM. The maximum current variation is less than 8%. This implies that to a first order, current consumption of a piece of code is independent of the code and depends only on the operating voltage and frequency of the processor. The first-order software energy estimation model is then simply

$$E_{\text{tot}} = V_{dd} I_0(V_{dd}, f) \Delta t \tag{1}$$

where V_{dd} is the supply voltage and Δt is the program execution time.

2) Second-Order Model: While the current variation across programs is quite small in the StrongARM, it might be significant in datapath dominated processors. For example, the current consumption of the multiply instruction in DSPs will be far greater than the current consumption of other instructions. In such cases, a simple model like (1) will have significant error. The following second-order model is proposed.

Let $C_k, 0 \leq k \leq K-1$ denote the K energy differentiated instruction classes in a processor. Energy differentiated instruction classes are partitions of the instruction set such that the average current consumption of any class is significantly different from that of another class and the current variation between instructions within the same class is small. Class partitions can also be done on the basis of different cycles, e.g., instruction, data access, etc. Let c_k denote the fraction of total cycles in a given program that can be attributed to instructions/cycles in the class C_k, i.e., $\sum c_k = 1$. The proposed second-order current consumption equation is

$$I(V_{dd}, f) = I_0(V_{dd}, f) \sum_{k=0}^{K-1} w_k c_k \tag{2}$$

where w_k are a set of weights. Let $\bar{W}$ represent the vector $(w_0, w_1, \ldots, w_{K-1})^T$. Let $P_n, 0 \leq n \leq N-1$ represent a set of N benchmark programs, C_n denote the cycle fractions vector for program P_n i.e., $(c_0^n, c_1^n, \ldots, c_{K-1}^n)$ and I_n denote

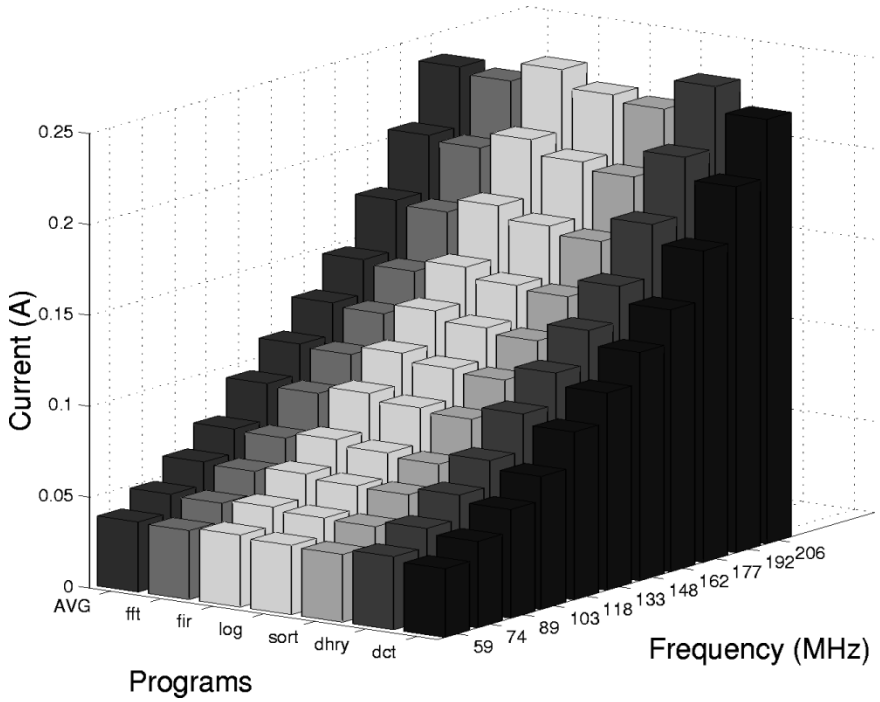

Fig. 5. Program current consumption as a function of operating point.

its average current consumption. Based on (2), we can express the current vector $\bar{I}$ in the following form

$$\bar{I} = I_0(V_{dd}, f)\bar{C}\bar{W} \tag{3}$$

where $\bar{I}$ is the average current $(I_0, I_1, \ldots, I_{N-1})^T$ for the N programs, $\bar{C}$ is a matrix with C_n as the nth row. The weights can be solved for in a least mean square sense using the pseudo-inverse

$$\bar{W} = \frac{1}{I_0(V_{dd}, f)}(\bar{C}^T\bar{C})^{-1}\bar{C}^T\bar{I}. \tag{4}$$

If the instruction classes are a valid energy differentiated partition, the weighting vector $\bar{W}$ will reflect the energy differentiation. The maximum current prediction error will also go down considerably.

On the StrongARM SA-1100, we partitioned the cycles into 4 classes: 1) pure instruction cycles; 2) sequential memory access (accesses which are related to previous ones); 3) nonsequential accesses; and 4) internal cycles. The reader is referred to [29] for mode details. Current measurements were done for six benchmark programs running at all possible frequency and voltage combinations. The weighting vector is shown in Table I. The average current drawn at each operating frequency of the StrongARM is shown in Fig. 6. The StrongARM operates at 11 discrete frequency levels. The minimum operating voltage required is also shown and is almost linear with frequency. In fact, the normalized operating voltage and frequency are approximately related as

$$V_{\text{norm}} = 0.66 f_{\text{norm}} + 0.33 \tag{5}$$

where V_{norm} and f_{norm} are normalized to their respective maximum values.

TABLE I
WEIGHTING FACTORS FOR K = 4 ON THE STRONGARM

Class	Weight	Value
Instruction	w_1	2.1739
Sequential memory access	w_2	0.0311
Non-sequential memory access	w_3	1.2366
Internal cycles	w_4	0.8289

The weighting factors can be interpreted as follows. For programs where instruction cycles and nonsequential memory accesses dominate, the current consumption is higher than the average current at that operating point. Internal cycles and sequential memory access dominated programs will have a lower than average current consumption. The current prediction error with the second-order model can be reduced to less than 2%. The advantage is that this accuracy comes virtually free. No elaborate instruction level profiling is required. Such cycle counts as the ones shown in Table I are easily obtained using simulators/debuggers available with standard processors. Fig. 7 shows the overall improvement of current prediction accuracy on 66 benchmark functions. It can be seen that the current prediction is better in every case (the maximum error of 4.7% using a first order model is reduced to 2.3%). The effectiveness of the current weighting scheme will become more pronounced in processors having a wider variation in average current consumption.

C. Leakage Energy Modeling

With increasing trends toward low-power design, supply voltages are constantly being lowered as an effective way to reduce power consumption. However, to satisfy the ever demanding performance requirements, the threshold voltage is

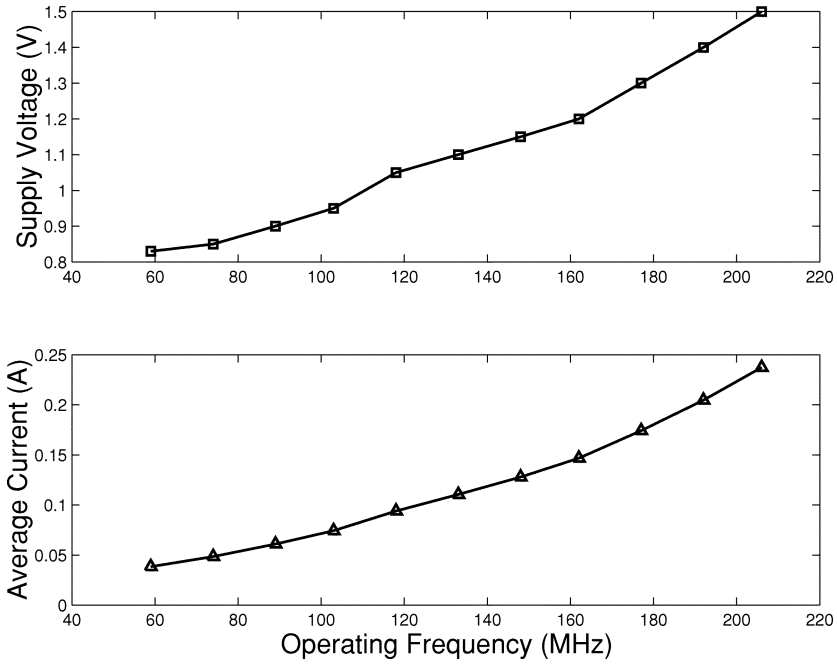

Fig. 6. Average current and supply voltage at each operating frequency of the StrongARM SA-1100.

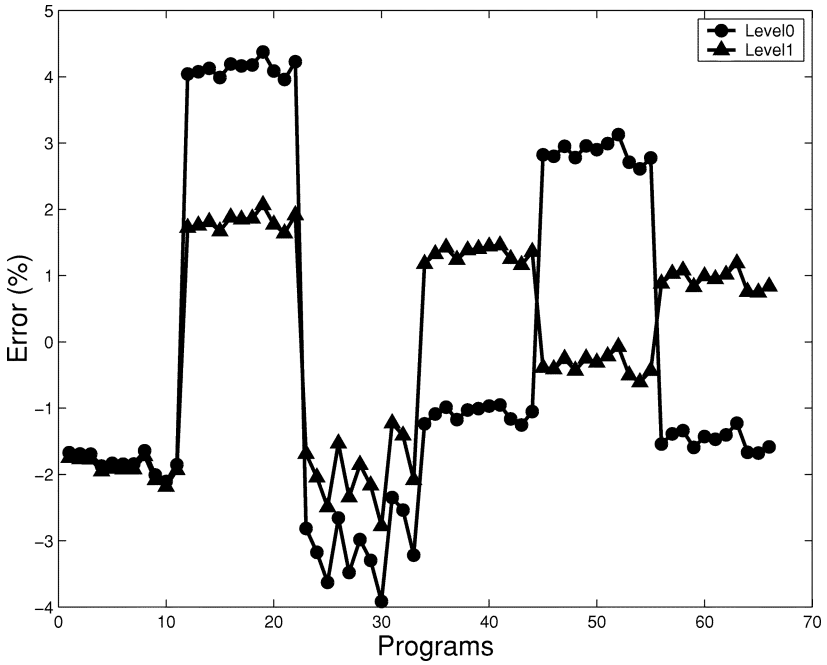

Fig. 7. First- and second-order model prediction errors.

also scaled proportionately to provide sufficient current drive and reduce the propagation delay. As the threshold voltage is lowered, the subthreshold leakage current increases exponentially. Leakage power is already becoming a significant portion of the power budget of contemporary microprocessors. We now outline a technique to separate the leakage current from the switching current for a given program.

1) Principle: The power consumption of a subroutine executing on a microprocessor can be macroscopically represented as

$$P_{\rm tot} = P_{\rm dyn} + P_{\rm stat} = C_L V_{dd}^2 f + V_{dd} I_{\rm leak} \tag{6}$$

where $P_{\rm tot}$ is the total power which is the sum of the static and dynamic components, C_L is the total average capacitance being switched by the executing program, per clock cycle, and f is the operating frequency (assuming that there are no static bias currents in the microprocessor core) [20]. Let us assume that a subroutine takes Δt time to execute. This implies that the energy consumed by a single execution of the subroutine is

$$E_{\rm tot} = P_{\rm tot} \Delta t = C_{\rm tot} V_{dd}^2 + V_{dd} I_{\rm leak} \Delta t \tag{7}$$

where $C_{\rm tot}$ is the total capacitance switched by executing subroutine. Clearly, if the execution time of the subroutine is

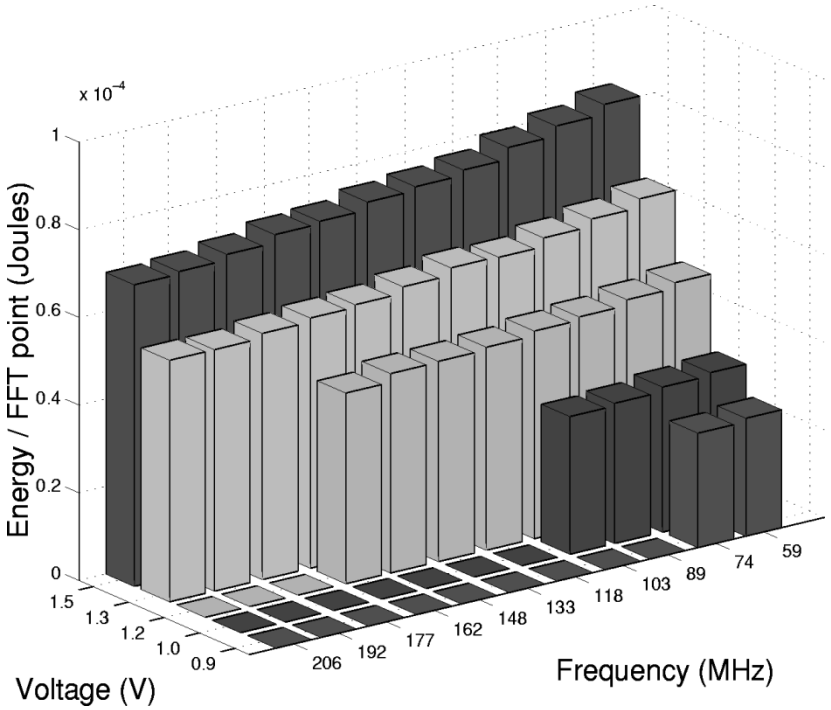

Fig. 8. FFT energy consumption on the StrongARM SA-1100.

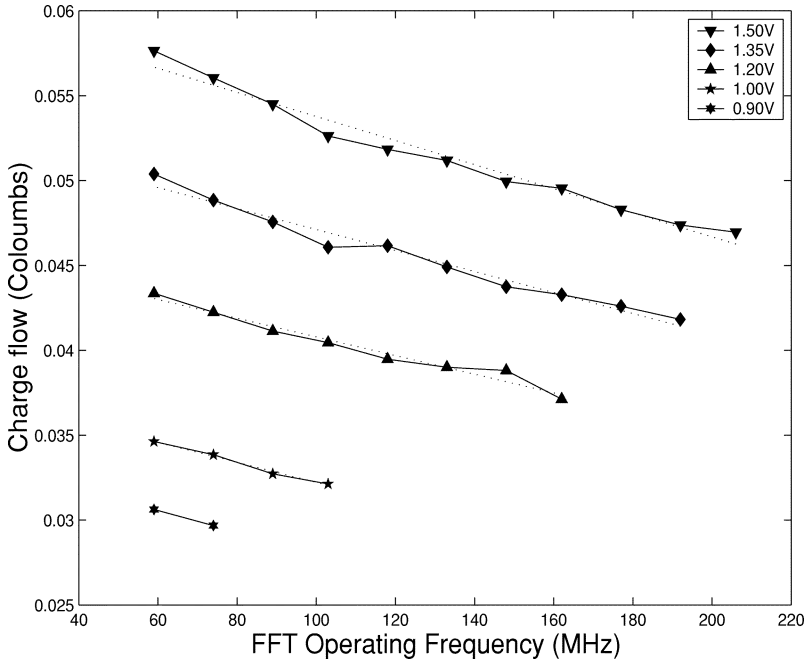

Fig. 9. FFT charge consumption.

changed (by changing the clock frequency), the total switched capacitance $C_{\rm tot}$ remains the same. Essentially, the integrated circuit goes through the same set of transitions except that they occur at a slower rate. Therefore, if we execute the same subroutine at different frequencies, but at the same voltage, and measure the energy consumption we should observe a linear increase with the execution time with the slope being proportional to the amount of leakage.

2) Observations: The subroutine chosen for execution was the decimation-in-time fast Fourier transform (FFT) algorithm because it is a standard, computationally intensive, DSP operation. The execution time for an $N = 1024$ point FFT on the StrongARM is a few tenths of a second. To obtain accurate execution time and stable current readings, the FFT routine was run a few hundred times for each observation. A total of eighty different data points corresponding to different supply

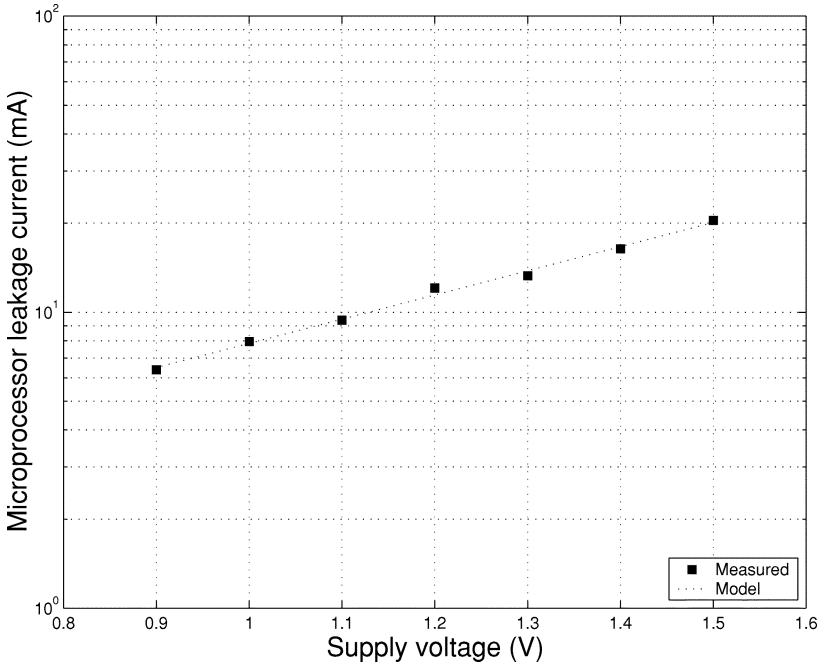

Fig. 10. Leakage current dependence on supply voltage.

voltages between 0.8–1.5 V and operating frequencies between 59–206 MHz were compiled.

Fig. 8 illustrates the implications of (7). When the operating frequency is fixed and the supply voltage is scaled, the energy scales almost quadratically. On the other hand, when the supply voltage is fixed and the frequency is varied the energy consumption decreases linearly with frequency (i.e., increases linearly with the execution time) as predicted by (7). Not all frequency, voltage combinations are possible. For example, the maximum frequency of the StrongARM is 206 MHz and it requires a minimum operating voltage of 1.4 V.

We can measure the leakage current from the slope of the energy characteristics, for constant voltage operation. One way to look at the energy consumption is to measure the amount of charge that flows across a given potential. The charge attributed to the switched capacitance should be independent of the execution time, for a given operating voltage, while the leakage charge should increase linearly with the execution time. Fig. 9 shows the measured charge flow as a function of the operating frequency for a 1024 point FFT. The amount of charge flow is simply the product of the execution time and current drawn. As expected, the total charge consumption decreases, almost linearly, with operating frequency (i.e., increases with execution time) and the slope of the curve, at a given voltage, directly gives the leakage current at that voltage.

The dotted lines are the linear fits to the experimental data in the minimum mean-square error sense. At this point, it is worthwhile to mention that the term "leakage current" has been used in an approximate sense. Truly speaking, what we are measuring is the total static current in the processor, which is the sum of leakage and bias currents. However, in the SA-1100 core, the bias currents are small and most of the static currents can be attributed to leakage. This assertion is further supported by the fact that the static current we measure has an exponential behavior.

From the BSIM2 MOS transistor model [21], the subthreshold current in a MOSFET is given by

$$I_{\text{sub}} = Ae^{\frac{(V_G - V_S - V_{TH0} - \gamma' V_S - \eta V_{DS})}{n' V_T}} \left(1 - e^{-\frac{V_{DS}}{V_T}}\right) \tag{8}$$

where

$$A = \mu_0 C_{\text{ox}} \frac{W_{\text{eff}}}{L_{\text{eff}}} V_T^2 e^{1.8} \tag{9}$$

and V_T is the thermal voltage, V_{TH0} is the zero bias threshold voltage, γ' is the linearized body effect coefficient, η is the drain induced barrier lowering (DIBL) coefficient and V_G, V_S, and V_{DS} are the usual gate, source, and drain-source voltages, respectively. The important point to observe is that the subthreshold leakage current scales exponentially with the drain-source voltage. The leakage current at different operating voltages was measured as described earlier, and is plotted in Fig. 10. The overall microprocessor leakage current scales exponentially with the supply voltage. Based on these measurements the following model for the overall leakage current is proposed for the microprocessor core

$$I_{\text{leak}} = I_0 e^{\frac{V_{dd}}{n V_T}} \tag{10}$$

where $I_0 = 1.196$ mA and $n = 21.26$ for the StrongARM SA-1100.

3) Explanation of Exponential Behavior: The exponential dependence of the leakage current on the supply voltage can be attributed to the DIBL effect. Consider the stack of nMOS devices shown in Fig. 11. Equation (8) suggests that for a single transistor, the leakage current should scale exponentially with $V_{DS} = V_{DD}$ because of the DIBL effect. However since the

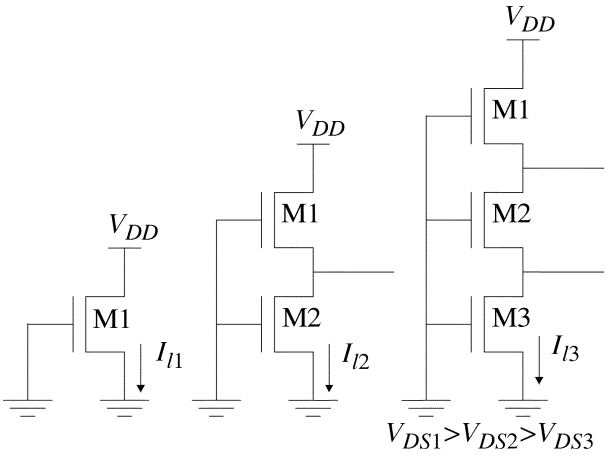

Fig. 11. Effect of transistor stacking.

ratio V_{DS}/V_T is larger than 2, the term inside the brackets of (8) is almost 1. It has been shown in [22] that this approximation is also true for a stack of two transistors. With three or more transistors, the ratio V_{DS}/V_T for at least the lowest transistor becomes comparable to or even less than 1. Therefore, the term inside the bracket of (8) cannot be neglected for such cases. The leakage current progressively decreases as the number of transistors in the stack increases and for a stack of more than three transistors the leakage current is small and can be neglected. It has further been shown in [22] that the ratio of the leakage currents for the three cases shown in Fig. 11 can be written as

$$I_{l1} : I_{l2} : I_{l3} = 1.8e^{\frac{\eta V_{dd}}{n' V_T}} : 1.8 : 1. \tag{11}$$

Therefore, the leakage current of a MOS network can be expressed as a function of a single MOS transistor by accounting for the signal probabilities at various nodes and using the result of (11). If the number of stacked devices is more than three, the leakage current contribution from that portion of the circuit is negligible. If there are three transistors stacked such that two of them are "OFF" and one is "ON" then the leakage analysis is the same as the stack of two "OFF" transistors. For parallel transistors, the leakage current is simply the sum of individual transistor leakages. A similar argument holds for pMOS devices. Since, the leakage current of a single MOS transistor scales exponentially with V_{DD}, using the above argument, we can conclude that the total microprocessor leakage current also scales exponentially with the supply voltage.

4) Separation of Current Components: Table II compares the measured leakage current with the values predicted by (10). The maximum percentage error measured was less than 6% over the entire operating voltage range of the StrongARM which suggests a fairly robust model.

Based on the leakage model described by (10), the static and dynamic components of the microprocessor current consumption can be separated. The standby current of the StrongARM in the "idle" mode at 1.5 V is about 40 mA. This is not just the leakage current but also has the switching current due to the circuits that are still being clocked. On the other hand, using the technique just described, the exact leakage component of the current consumed can be extracted and its exponential dependence on supply voltage can be shown.

TABLE II
LEAKAGE CURRENT MEASUREMENTS

V_{DD} (V)	I_{leak} (mA) Measured	Model	Error (%)
1.50	20.41	20.10	1.50
1.40	16.35	16.65	-1.84
1.30	13.26	13.80	-4.04
1.20	12.07	11.43	5.27
1.10	9.39	9.47	-0.87
1.00	7.96	7.85	1.40
0.90	6.39	6.53	-1.70

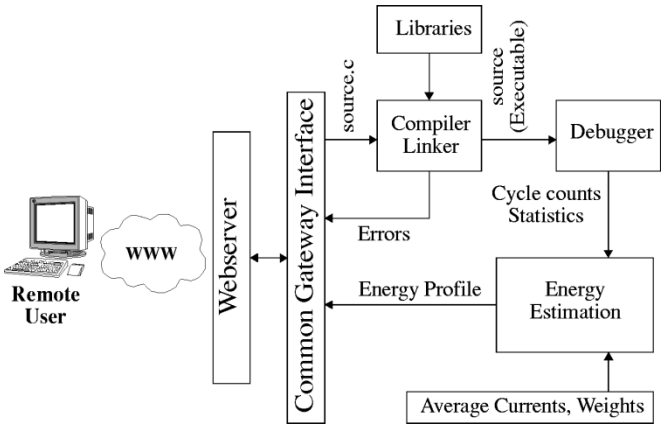

Fig. 12. JouleTrack block diagram.

D. JouleTrack

The estimation techniques described in Section II were implemented in a web based tool called JouleTrack and could just as easily be incorporated into a standard IDE. The tool is available online [8]. The broad approach in the tool is summarized in Fig. 12. The user uploads his *C* source code. The webserver uses common gateway interface (CGI) scripts to create a temporary workspace for the user. The uploaded program is compiled and linked with any standard *C* libraries, if required. The user also specifies any command line arguments that the program might need along with a target operating frequency. Compiler optimization options are also available. The user can choose the simulation level. Three levels are supported in the current version of JouleTrack—the first two implement the first- and second-order models described in Section III-B, while the third level also incorporates the leakage model described in Section III-C. Compile/link time errors are reported back by the CGI to the user. If no errors exist the program is executed on an ARM simulator which produces the program outputs (which the user can view), assembly listing (which can also be viewed) as well as runtime statistics like execution time, cycle counts, etc. These statistics are fed into an estimation engine which computes the energy profile and charts the various energy components using the techniques prescribed.

IV. OPERATING SYSTEM ENERGY PROFILING

Using the techniques outlined in Section III it is possible to develop a framework for exposing the energy consumption of

software to the application developer. In many embedded systems, a significant portion of tasks are executed by various operating system functions. As such, it is important to benchmark the energy consumption of these functions in order to have an estimate of the overhead energy cost or running an operating system. Just as software energy consumption depends strongly on the type of processor, memory configuration, etc., the energy overhead of various operating system functions depends completely on the type of OS and target hardware. In this section, we describe the results and technique used to benchmark the embedded OS that runs on the μAMPS sensor node. This OS is based on the Red Hat eCos [23] operating system and was ported to the sensor node.

A. Sensor Hardware Overview

The current version of the μAMPS sensor node is based on the StrongARM SA-1110 processor and has 1 MB of onboard SRAM and flash memory. The board runs at a nominal battery (single-lithium primary-cell) power supply of 4.0 V. The onboard power supply circuits generate a 3.3 V supply for all digital circuits. A separate analog power supply is also generated to isolate the digital power supply noise from the analog circuits. The 3.3 V digital power supply also powers the I/O pads of the StrongARM SA-1110 processor. However, the core power supply is generated through a dc/dc converter circuit that can regulate the power supply from 0.925 V to a maximum of 2.0 V with a conversion efficiency of about 85%. The radio module is on a similar sized board and consists of a dual power 2.4 GHz radio for 10 m and 100 m ranges. The 16 bit bus interface connector will allow the radio module to be stacked onto the processor board. In addition, the connector allows a different sensor board (e.g., a seismic sensor) to be stacked as well. The processor board also has an RS-232 and a USB connector for remote debugging and connecting to a basestation. The board features a built in acoustic sensor (a microphone, some opamps, and A/D circuit) that talk to the StrongARM processor using the synchronous serial port (SSP). The opamp gains are programmable and processor controlled. An envelop detect mechanism has also been incorporated into the sensor circuit, which bypasses the A/D circuit and wakes up the processor when the signal energy crosses a certain programmable threshold. Using this feature can significantly reduce the power consumption in the sense mode and allows for truly even driven computation. The sensor processor board is depicted in Fig. 13.

B. OS Overview

The eCos OS is designed to be completely scalable across platforms as well as within a given platform. Essentially, source level configuration allows the user to add or remove packages from a source repository based on system requirements. For example, the user might choose to remove math libraries and the resulting kernel will be leaner. The core eCos system consists of a number of different components such as the kernel, the *C* library, an infrastructure package, etc. Each of these provides a large number of configuration options, allowing application developers to build a system that matches the requirements of their particular application. To manage the potential complexity of multiple components and lots of configuration options, eCos

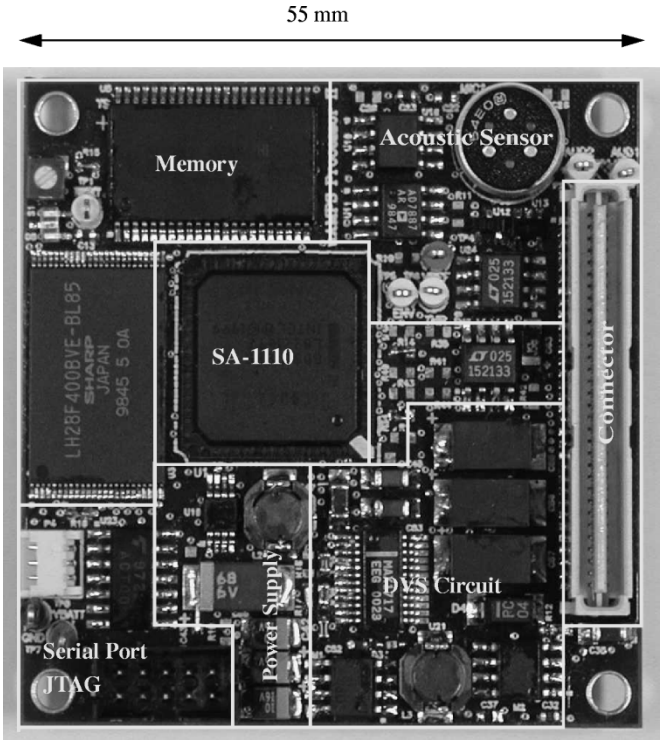

Fig. 13. μ AMPS processor board.

has a component framework: a collection of tools specifically designed to support configuring multiple components. Furthermore, this component framework is extensible, allowing additional components to be added to the system at any time. The eCos component description language (CDL) lets the configuration tools check for consistency in a given configuration and point out any dependencies that have not been satisfied. The eCos OS was ported to our sensor node and a software power management layer was added to exploit the shutdown and dynamic voltage and frequency scaling hooks available.

C. Energy Profiling of Kernel Functions

Profiling the energy cost of real-time kernel functions is nontrivial because it is impossible to exhaustively characterize all possible operating scenarios. Most functions are designed to be interruptible and measurements can at best be typical. To minimize error in timing measurements a hardware timer, which drives the real-time clock, was used. This timer can be read with microsecond resolution. For each measurement, the operation was repeated a number of times. A time stamp was obtained directly before and after each operation was performed. Data gathered for the entire set of operations was then analyzed, generating average, maximum, and minimum values. The sample variance was also calculated. The cost of obtaining real-time clock timer values was also measured, and was subtracted from all timing measurements. The average energy cost was computed by multiplying the average current consumption with the average estimated execution time.

Most kernel functions can be measured separately. In each case, a reasonable number of iterations were performed. Where the test case involved a kernel object, for example creating a task, each iteration was performed on a different object. A set

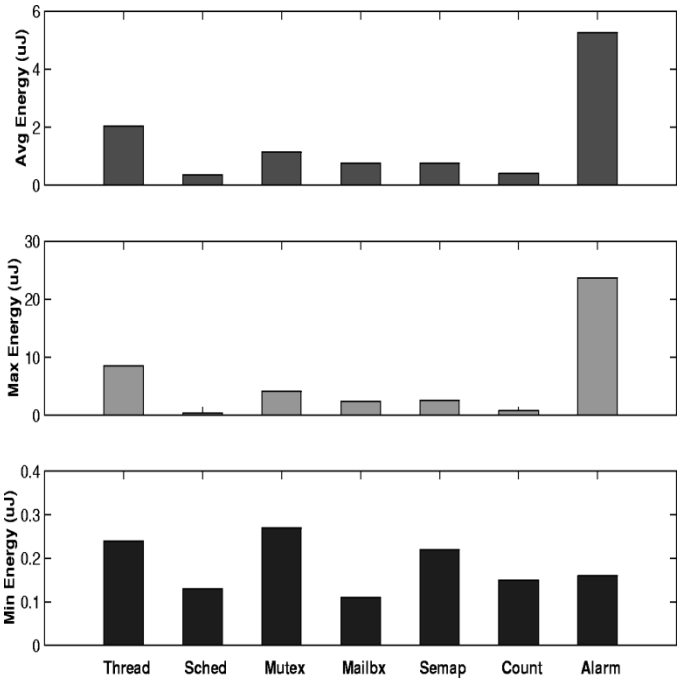

Fig. 14. Average, maximum, and minimum energy consumption of kernel function calls (grouped by category).

of tests which measured the interactions between multiple tasks and certain kernel primitives were also tested. Most functions were tested in such a way as to determine the variations introduced by varying numbers of objects in the system. For example, the mailbox test measured the cost of a peek operation when the mailbox was empty, had a single item, and had multiple items present. In this way, any effects of the state of the object or how many items it contained could be determined.

Fig. 14 shows the average, maximum and minimum system energy cost of various kernel functions grouped by category. The measurements have been made on a nominal system running at 59 MHz. The system power supply was 4.0 V. Every function category consists of 10–20 different individual OS functions. Each function was tested a number of times under different conditions. Although it is impossible to completely characterize a real-time operating system (since latencies and therefore energy consumption depends heavily on interrupts, workloads etc.), Fig. 14 gives an estimate of the energy cost of various kernel API calls under nominal conditions. It can be seen that most kernel functions cost a few μ Joules of energy (corresponding to a few tens of μm s execution time). Scheduling operations are the most efficient, while thread manipulation and alarm functions are more expensive in comparison. A more detailed characterization is available in [24].

V. System Level Software Power Management

Exposing the energy consumption of the application and OS to the developer can enable important design refinements that ultimately enhance the energy efficiency of the system. Substantial energy savings can obtained from software if it exploits all the power management hooks available in the hardware. For example, the μAMPS sensor node features dynamic voltage (and frequency), scaling (DVS) that allows the processor operating voltage, and frequency to be reduced when workload levels are low. This leads to quadratic energy savings [16], [25]. In addition, selective portions of the node can be shutoff to reduce the power consumption [26], [27]. The power aware software environment provides a power-management API along with the power characterization of various idle and active modes. In this section, we quantify the efficacy of system level software power management, enabled by exposing the energy consumption of the system to software, on the sensor node.

A. Active Mode Power Savings

Fig. 15 shows the power consumption of the sensor node in the fully active state (all modules on) as a function of the operating frequency of the SA-1110. The figure shows the power consumption using both DVS and just frequency scaling (which is done at a fixed core voltage of 1.65 V). The system power supply was 4.0 V. In the active mode, DVS is the primary source of power management. When running at the maximum operating voltage and operating frequency, the power consumption of the system is almost 1 W. Active power management using DVS results in about 53% system wide power savings.

An important energy performance tradeoff exists based on the fact that minimum energy consumption results when the processing rate variation is minimum [28]. Minimum processing rate variation in turn implies a larger maximum performance hit. Fig. 16 plots the relative battery life improvement as a function of the variance in workload. Each workload profile had a Gaussian distribution with a fixed average workload. It can be seen that with a fixed average workload the energy efficiency (and therefore battery lifetime) degrades with increased workload variance. Exposing such phenomena to the application developer is important. Judicious use of the OS scheduler can reduce preemptions and thread suspensions thereby minimizing the workload variance on the processor.

B. Idle Mode Power Savings

Table III shows the measured power consumption of the sensor node in various modes of operation. The sensor node can be classified as a processor power dominated architecture. The radio module follows the processor in power requirement (estimated at about 70 mA at 3.3 V). DVS can reduce system power consumption by 53%. Shutting down each of the components (analog power supply, radio module, and the processor itself) results in another 54% power savings, i.e., idle power management accounts for about 97% of system wide power savings. The overall power savings attributed to various power management hooks has been shown in Fig. 17. Exposing these power characterizations to the application layer facilitates energy efficient design.

The energy savings in the field depend significantly on processing rate requirements and event statistics. To estimate the energy savings from active mode power consumption, we would need an estimate of the workload variation in the system. If we assume that the average workload requirement was 50% with slow variations, the estimated energy savings is about 30%. Idle mode energy savings, on the other hand, can be significant. If we assume that the operational duty cycle is 1%, the estimated

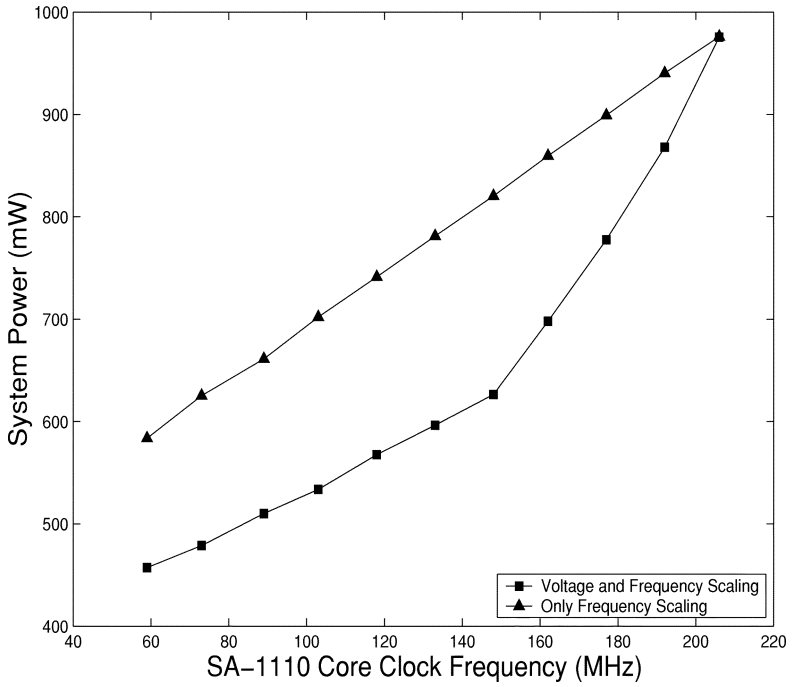

Fig. 15. System level power savings from active power management using DVS.

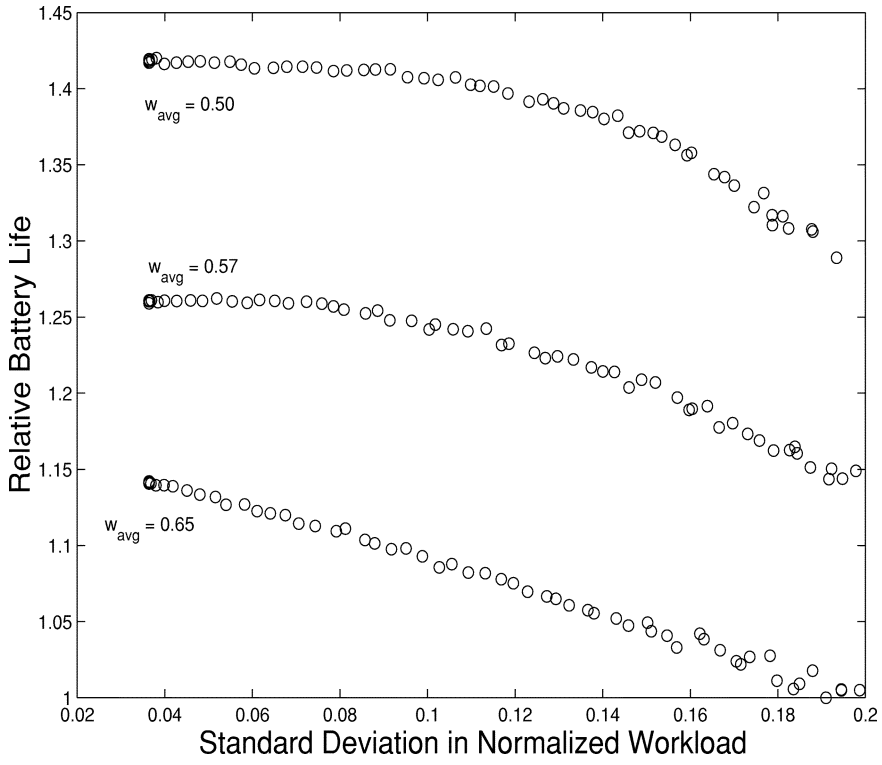

Fig. 16. Degradation in DVS savings with increase in workload variance.

energy savings is about 96%. This implies that sensor node battery life can be improved by a factor of over 27 (i.e., a node that lasts for a day with no power management will now last for almost a month)! With a 10% duty cycle, the battery life improvement is by a factor of about 10. Fig. 18 shows the factor by which battery life of the sensor node can be enhanced by using power management techniques as a function of the workload and duty cycle requirement. The important point to observe is that an energy exposed application development environment makes such information available to the software engineer who can then make important design refinements that enhance the system energy efficiency.

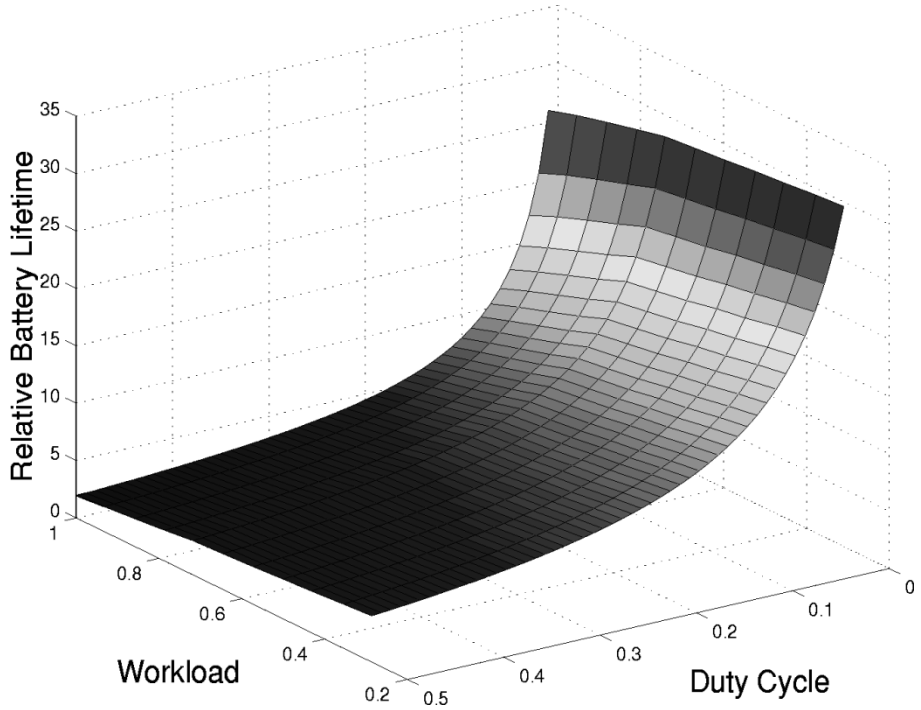

Fig. 18. Battery life improvement in the sensor node compared to a node with no power management as a function of duty cycle and active workload.

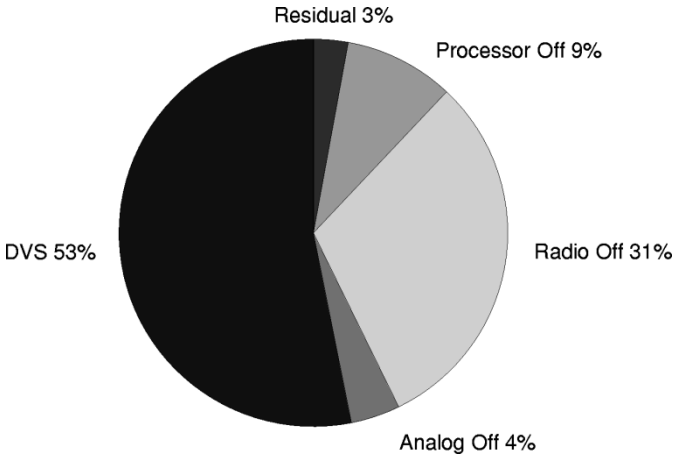

Fig. 17. System level power savings distribution.

TABLE III
MEASURED POWER CONSUMPTION IN VARIOUS OPERATINGMODES OF THE SENSOR

	Processor		Radio	Analog	Power
	DVS	Mode			(mW)
Active Mode	max	run	on	on	975.6
	min	run	on	on	457.2
Idle Mode	min	run	on	off	417.2
	min	run	off	off	117.2
	-	sleep	off	off	28.0

VI. CONCLUSION

Based on experiments done on the StrongARM SA-1100 and Hitachi SH-4, we conclude that the common overheads presented across instructions in general purpose microprocessors result in the variation in current consumption of different instructions being small. The variation in current consumption of programs is even smaller. Therefore, to a first order, we can assume that processor current consumption depends only on operating frequency and supply voltage and is independent of the executing program. A second-order model that uses energy differentiated instruction classes/cycles is also proposed and it was shown that the resulting estimation error was reduced to less than 3%. A technique for separating the leakage and switching energy components of software is also outlined. An important part of such an energy exposed software development framework involves characterizing the embedded operating system over which the applications execute. Energy profiling was done for the eCos real-time embedded operating system. We conclude that the operating system function calls can have a substantial energy overhead and exposing the energy cost of these functions can improve the energy efficiency of the application. Using the μAMPS system as an example, we have demonstrated that an energy exposed software technique can substantially enhance the system energy efficiency.

ACKNOWLEDGMENT

The authors would like to thank R. Min for his help with the DVS circuit and T. Furrer for the initial work of porting the eCOS operating system to StrongARM.

REFERENCES

[1] Moore's Law [Online]. Available: http://www.intel.com/intel/museum/25anniv/hof/moore.-htm

[2] A. Chandrakasan and R. Brodersen, *Low-Power CMOS Design*. Piscataway, NJ: IEEE Press, 1998.

[3] S. Santhanam *et al.*, "A low-cost, 300-MHz, RISC CPU with attached media processor," *IEEE J. Solid-State Circuits*, vol. 33, pp. 1829–1839, Nov. 1998.

[4] T. Xanthopoulos and A. Chandrakasan, "A low-power DCT core using adaptive bit-width and arithmetic activity exploiting signal correlations and quantizations," presented at the *Proc. Symp. VLSI Circuits*, June 1999.

[5] J. Goodman, A. P. Dancy, and A. P. Chandrakasan, "An energy/security scalable encryption processor using an embedded variable voltage DC/DC converter," *IEEE J. Solid-State Circuits*, vol. 33, pp. 1799–1809, Nov. 1998.

[6] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proc. DAC 1999*, New Orleans, LA, pp. 151–156.

[7] Intel VTune Performance Analyzer [Online]. Available: http://developer.intel.com/software-/products/vtune/index.htm

[8] JouleTrack—A Web Based Tool for Software Energy Profiling [Online]. Available: http://www-mtl.mit.edu/research/anantha/jouletrack/JouleTrack/

[9] A. Sinha and A. Chandrakasan, "JouleTrack—A web based tool for software energy profiling," presented at the Proc. 38th Design Automation Conf., Las Vegas, NV, June 2001.

[10] The MIT μAMPS Project [Online]. Available: http://www-mtl.mit.edu/research/icsystems-/uamps/

[11] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 437–445, Dec. 1994.

[12] J. T. Russell and M. F. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," in *Proc. Int. Conf. Computer Design*, 1998, pp. 328–333.

[13] H. Mehta, R. M. Owens, and M. J. Irwin, "Instruction level power profiling," in *Proc. ICASSP*, 1996, pp. 3326–3329.

[14] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "High-level software energy macro-modeling," in *Proc. Design Automation Conf.*, 2001, pp. 605–610.

[15] H. Cheng-Ta, C. Lung-Sheng, and M. Pedram, "Microprocessor power analysis by labeled simulation," in *Proc. Design Automation Test Conf., Europe*, 2001, pp. 182–189.

[16] T. Simunic *et al.*, "Dynamic voltage scaling and power management for portable systems," in *Proc. Design Automation Conf.*, 2001, pp. 524–529.

[17] L. Yung-Hsiang and G. Micheli, "Comparing system level power management policies," *IEEE Des. Test Comput.*, vol. 18, pp. 10–19, Mar.–Apr. 2001.

[18] Intel StrongARM Processors [Online]. Available: http://developer.intel.com/design/strong-/sa1100.htm

[19] Hitachi SuperH Processors [Online]. Available: http://www.superh.com/

[20] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[21] J. Sheu *et al.*, "BSIM: Berkeley short-channel IGFET model for MOS transistors," *IEEE J. Solid-State Circuits*, vol. SC-22, pp. 558–566, Aug. 1987.

[22] R. X. Gu and M. I. Elmasry, "Power dissipation analysis and optimization of deep submicron CMOS digital circuits," *IEEE J. Solid State Circuits*, vol. 31, pp. 707–713, May 1996.

[23] The eCos Operating System [Online]. Available: http://www.redhat.com/ecos

[24] A. Sinha, "Energy Efficient Operating Systems and Software," Ph.D. dissertation, Massachusetts Inst. Technology, Cambridge, July 2001.

[25] V. Gutnik and A. P. Chandrakasan, "An embedded power supply for low-power DSP," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 425–435, Dec. 1997.

[26] A. Sinha and A. Chandrakasan, "Dynamic power management in wireless sensor networks," *IEEE Des. Test Comput.*, pp. 62–75, Mar.–Apr. 2001.

[27] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan, "An architecture for a power-aware distributed microsensor node," presented at the *Proc. IEEE Workshop Signal Processing Systems (SiPS)*, Lafayette, LA, Oct. 2000.

[28] A. Sinha and A. Chandrakasan, "Dynamic voltage scheduling using adaptive filtering of workload traces," presented at the *Proc. 14th Int. Conf. VLSI Design*, Bangalore, Karnataka, India, Jan. 2001.

[29] *Intel StrongARM SA-1100 Microprocessor Developer's Manual*.

Amit Sinha received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, in 1998, and the S.M. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1999 and 2001, respectively.

He is a Cofounder and System Architect at Engim, Acton, MA, a company focused on developing high-performance wideband communication systems. He has worked on several projects at Texas Instruments, Intel and Hewlett Packard. His research interests include wireless communications and low-power systems and software.

Dr. Sinha was awarded the President of India Gold Medal for graduating *summa cum laude* from the Indian Institute of Technology, New Delhi. He is currently serving on the technical program committee for DATE, and has been a reviewer for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, VLSI Design, DAC, and ISLPED.

Nathan Ickes received the B.S. and M.Eng. in electrical engineering and computer science, from the Massachusetts Institute of Technology (MIT), Cambridge, in 2001 and 2002 respectively. He is currently pursuing the Ph.D. degree at MIT.

His research interests include microsensor systems, power-efficient processors, and power-aware operating systems.

Anantha P. Chandrakasan received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1989, 1990, and 1994, respectively.

He is a Cofounder and Chief Scientist at Engim, Inc., Acton, MA, a company focused on high-performance wireless communications. Since September 1994, he has been at the Massachusetts Institute of Technology, Cambridge, and is currently a Professor of Electrical Engineering and Computer Science. He held the Analog Devices Career Development Chair from 1994 to 1997. His research interests include the ultra low-power implementation of custom and programmable digital signal processors, distributed wireless sensors, ultra wideband radios, and emerging technologies. He is a coauthor of *Low-Power Digital CMOS Design* (Norwell, MA: Kluwer, 1995) and *Digital Integrated Circuits* (Englewood Cliffs, NJ: Prentice-Hall, 2002). He is a coeditor of *Low-Power CMOS Design* (Piscataway, NJ: IEEE Press, 2001) and *Design of High-Performance Microprocessor Circuits* (Piscataway, NJ: IEEE Press, 2001).

Dr. Chandrakasan received the Best Paper Award in 1993 for IEEE Communications Society's Best Tutorial Paper Award, the IEEE Electron Devices Society's 1997 Paul Rappaport Award for the Best Paper in an EDS publication, and the 1999 Design Automation Conference Design Contest Award. He received the National Science Foundation Career Development Award in 1995, the IBM Faculty Development Award in 1995, and the National Semiconductor Faculty Development Award in 1996 and 1997. He has served on the technical program committee of various conferences including International Solid-State Circuits Conference (ISSCC), VLSI Circuits Symposium, DAC, and International Symposium on Low-Power Electronics and Design (ISLPED). He has served as a technical program cochair for the 1997 ISLPED, VLSI Design'98, and the 1998 IEEE Workshop on Signal Processing Systems, and as a general cochair of the 1998 ISLPED. He was the Signal Processing Subcommittee Chair for ISSCC from 1999 to 2001, the Program Vice-Chair for ISSCC 2002, the Technical Program Chair for ISSCC, 2003. He was an Associate Editor for the IEEE TRANSACTION ON SOLID-STATE CIRCUITS from 1998 to 2001. He served as an Elected Member of the Design and Implementation of Signal Processing Systems (DISPS) Technical Committee of the Signal Processing Society and serves on the SSCS AdCOM. He is the Technology Directions Chair for ISSCC 2004.