Full System Energy Estimation for Sensor Network Gateways

Selim Gurun Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{gurun,ckrintz}@cs.ucsb.edu

ABSTRACT

We present a new energy estimation model for sensor network intermediate gateway nodes (i.e. Crossbow Stargates). Such devices are battery powered and resource constrained and commonly employed as communication, processing, and gateway elements within sensor networks. Understanding and accurately estimating the energy behavior of such devices is key to conserving the battery life of the system.

Our approach considers the system as a whole and couples techniques that estimate energy consumption for a wide range of program activity, including computation, communication, and persistent storage access. We construct our model using empirical data that we collect via hardware performance monitors in the device and novel software performance monitors in the Linux operating system. By integrating information from the hardware, operating system, and program, we are able to accurately characterize the full system energy behavior of the device and its programs.

We extensively evaluate our model and compare its accuracy to that of an extant and similar approach to power estimation for the CPU and and memory subsystem of the Stargate. We find that this prior work, when applied directly to estimate whole system power consumption, is ineffective (introducing error rates of over 50%). Our model achieves an error rate of 3% for computationally bound tasks and of 11% for programs that employ both computation and communication.

1. INTRODUCTION

Wireless sensor networks have gained in popularity recently as a result of their low cost, small size, and their potential for enabling transparent interconnection between the physical world and powerful information systems. Typical installations of these systems consist of a hierarchy of heterogeneous devices that range in capability but are both resource constrained and battery powered. Computationally simple, low-power, sensor elements make physical measurements, perform minor processing, and relay the collected data to more powerful devices. These more powerful devices, i.e., gateways or intermediate nodes, implement significantly more functionality, computational power, and battery capacity than the simpler sensor elements. To program and to facilitate efficient use of sensor systems, we must be able to characterize accurately the power and energy consumption of tasks that execute using these devices.

A significant body of research exists that models and estimates power and energy consumption for a range of sensor network devices [15, 18, 17, 20, 13, 3, 2, 9, 12] and battery technologies [21, 14, 5, 19, 1, 16]. Prior work has provided models for estimating the power consumption of the simpler sensor elements [15, 18, 17] as well as for specific components of the sensor gateways (and similar, battery-powered devices) [**?**, 2, **?**, 12, 20, 13, 9, 3]. However, a key missing piece in this prior work is effective and accurate estimation of *full-system* energy consumption for sensor gateways. In this paper, we present and evaluate such a model for the Crossbow Stargate. The key to our approach is integrated architectural, operating system, and program-level energy behavior characterization.

Extant approaches to estimation of power consumption for gateway-class devices, focus on the primary consumer of battery power: the CPU [3, 2, 12]. Such devices typically implement an energy efficient processor such as the Intel XScale [7]. Prior work has produced very accurate models (e.g., with an average error of 4%) for estimating the power consumption of such processors for a wide range of applications [3]. However, such models are not sufficient for tasks that do not solely perform computation, e.g. sensor network tasks that perform computation, communication, and other I/O. The goal of our work is to model accurately the energy consumption of the entire Stargate device as it executes such tasks.

To enable accurate, full system energy estimation for complex Stargate tasks, we couple the use of hardware performance monitors with operating system (OS) *software* performance monitors. We consider three primary Stargate activities: computation, wireless communication, and persistent storage access. Our approach is empirical in that we use program profile information from a range of programs to develop our model.

Moreover, we employ a two phase estimation process. In the first phase, we filter the profile information to reduce variance in our measurements. Whenever we (or others) measure the performance of tasks that execute on real devices with real OS support, nondeterministic and transient behavior can perturb the measurements. That is, multiple runs of the same program and input produce different results. Such perturbations are caused by a wide range of factors including hardware interrupts, memory and I/O effects, and system calls. To mitigate this problem, we present a novel suite of statistical techniques that filter outliers from the data sets using observations of the clock cycle counts. Our filters use and extend similar techniques used in prior work and help to improve the quality of our full-system energy estimation model – the second phase of our approach.

Our model couples independent linear functions, that we param-

eterize using our filtered data sets, for three primary program activities: computation, communication, and persistent storage access. Our computation model combines measurements from a subset of the available hardware performance monitors (HPMs) on the device. This model is similar to that in prior work [**?**, 3] but is designed to estimate full system energy consumption as opposed to the power consumed by the CPU and memory subsystem in isolation (as in prior work).

We couple this computation model with two other novel models that estimate I/O behavior of a sensor network program. Our models estimate the energy consumed for 802.11b wireless network communication and flash memory access. For communication, we employ a similar approach but extend it to use empirical data from novel software performance monitors (SPMs) in the operating system. Our SPMs collect packet counts and bytes transfered to parameterize the linear estimation function for communication. For flash memory, we use a simple model that estimates power consumption for reads and writes large files (>300KB). We parameterize two different models (one for read and one for write) due to the asymmetric energy characteristics of flash access. We use profiles of random, variable size, file access, to parameterize the models.

We evaluate our model using a wide range of programs (different from those that we use to parameterize the different components of the model). The programs perform a number of different activities typical of sensor network tasks for intermediate nodes, e.g, persistent storage access, network communication, and computation. Our estimator achieves an error rate of 3% on average for computation-bound programs and of 11% for tasks that employ both computation and communication. For communication, we find that both packets and bytes are necessary for accurately estimating communication behavior: we find that using the number of bytes transfered alone produces an order of magnitude larger error. When we consider computation and persistent storage access together, we achieve error rates similar to those for programs that we execute using RAM (as opposed to flash memory); moreover, we find that if the OS forces a flush following program execution, our combined model approach correctly approximates the (very different) energy consumption behavior of doing so.

In summary, we contribute with this paper:

- A full-system energy estimation model for the Stargate sensor network gateway device;
- A set of techniques for filtering noisy data sets from repeated executions of the same program/input;
- Individual components that combine to estimate computational behaviors, communication activity, and file I/O via a compact flash device; and
- An empirical evaluation and comparison of our model with an extant approach to CPU-based power estimation. Our results indicate that we must consider the full-system (architecture, operating system, and program) to characterize the behavior of typical sensor network tasks and estimate accurately the full-system energy consumption of the Stargate intermediate node.

In the sections that follow, we present our approach to full-system energy estimation. Our system consists of two key components: a suite of filters to mitigate the effects of spurious events that perturb performance measurements (Section *2.2.1*) and the composite model that we use to estimate energy consumption using these measurements (Section 2). Our model comprises three submodels that characterize computation, communication, and persistent storage activities of programs. In the remainder of the paper, we present our empirical evaluation (Section 3), related work (Section 4), and conclusions (Section 5).

2. ESTIMATING FULL-SYSTEM ENERGY CONSUMPTION

The goal of our work is to estimate accurately the energy behavior of sensor network tasks that employ a typical sensor network gateway (i.e. intermediate) device. We target the Crossbow Stargate device for our work. The Stargate implements a 400MHz Intel XScale CPU, short range 802.11 and long range WAN radio interfaces, and flash memory (that implements potentially gigabytes of storage), among other devices. In addition, the Stargate resources are managed by the Linux operating system.

Full system energy estimation of such devices is critical for sensor networking since their applications employ communication and computation as well as persistent storage and other types of I/O. Such behaviors are not captured by considering only the CPU and memory subsystems as is done in prior work. Our approach to full system energy estimation employs measurements from hardware performance monitors (HPMs) and novel, operating system software performance monitors (SPMs). Prior work has shown that the former are effective to estimate accurately the power consumption of the CPU [12, 8, 2, 3]. We employ these techniques to form a computation model (Section 2.2) which we couple with statistical models for wireless network communication (Section 2.3) and flash file system access (Section 2.4) to estimate the energy consumption of the entire device.

For each model, we use empirical data that we collect (via offline profiling) using a real device and wide range of benchmarks, to develop each model. We first describe our benchmarking methodology and then detail each component of the model in the following subsections.

2.1 Benchmarking Methodology

We employ two benchmark suites for our investigation of accurate estimation of sensor task power consumption for the Stargate. The first suite, to which we refer to as the training set, we use to define our model. The second suite, to which we refer to as the reference set, we use for the empirical evaluation of the accuracy of our model. The suites contain some overlapping applications, however the inputs that we used for the programs are different. The reference set however, contains additional non-overlapping programs. We present the suites and their input sizes in Table 1. The left half of the table is the training set and the right is the reference set. We use the benchmarks above the line to model/evaluate computation and those below for communication. We refer to the former as the NONET group and the latter as the NET group. The benchmarks in boldface we use to model and evaluate our persistent storage model; we refer to this group as FLASH. We execute all programs except the FLASH group (which we execute from the compact flash card) from RAM. The wireless network card is on for all of our experiments regardless of whether we use it or not.

Our applications come from popular benchmark suites (e.g. MediaBench [**?**] and Java benchmarks: Java-UCSD [**?**], Java-Olden [**?**]) and other similar studies. We plan to make all of the programs and inputs available via our project webpage. We also include a number of applications that perform communication. These include the secure copy protocol (scp) and netpipe [**?**]. For scp, we transfer a 17 MB file. We also include distributed (message passing interface (MPI)) applications: Game of Life (Life) [**?**], Pvnx, Pvkx and Pvkxb [**?**]. MPI is typically employed for distributed computing applications in larger systems. However, these MPI applications above have fair computation requirements that are within the limits of the Stargate CPU.

The characteristics of the MPI applications are analogous to the requirements of sensor network applications. For example in Life,

Training Benchmark Set		Reference Benchmark Set	
Application	Input Size	Application	Input Size
gsmdecode	400 KB	**gsmdecode**	30 KB
gsmencode	4.6 MB	gsmencode	295 KB
jpegdecode	1.6 MB	**jpegdecode**	1.5 MB
jpegencode	11.6 MB	jpegencode	18.6 MB
mpegdecode	34 KB	**mpegdecode**	79 KB
mpegencode	480KB	mpegencode	480 KB
UCSD java	N/A	em3d (Java)	N/A
		bisort (Java)	N/A
		treeadd (Java)	N/A
scp send	17 MB	scp send	1.6MB
scp receive	17 MB	scp receive	1.6MB
netpipe	N/A	game of life (MPI)	N/A
		pvnx (MPI)	N/A
		pvkx (MPI)	N/A
		pvkxb (MPI)	N/A

Table 1: Benchmarks. We use two benchmark sets: Training (left) to parameterize our model and Reference (right) to evaluate the accuracy of our model. We use the benchmarks above the line to model/evaluate computation; those below for communication. We refer to the former as the NONET group and the latter as the NET group. The benchmarks in boldface we use to model/evaluate persistent storage; we refer to this group as FLASH. We execute all programs except the FLASH group (which we execute from the compact flash card) from RAM.

the first processor divides the problem space into subspaces and distributes them to the other processors. Once the other processors complete the execution, they return the results back to the first processor. Then the first processor combines the results, and reiterates the process if necessary. This mechanism is very similar to recent query processing and vehicle tracking architectures for sensor networks. For example in [**?**], the nodes are organized in a tree structure. The root node distributes a query to the network. Each node partially processes the query and returns the results to the parent node. It is the parent node which combines the results. In [**?**], the remote sensor nodes collaborate with a central sensor node for tracking moving vehicles. The remote nodes do partial stream processing and filtering, however, they continuously exchange updates with the central node. The central node produces the results.

We collect performance data from these benchmarks in the form of CPU hardware performance monitors (HPMs). We collect data for each hardware event for five repetitions of the same program. HPMs provide efficient hardware support for profiling CPU-based activities. To monitor I/O activity, we employ novel software performance monitors. We describe these in Sections 2.3 and 2.4. The HPMs that the Stargate supports are shown in Table 2.

We collect the energy consumption data concurrently with HPM data using a 2-channel Agilent 54621A oscilloscope to construct our models and to validate our system. We connect a National Instruments data acquisition board to the exposed terminals of the Stargate to measure the energy consumption of peripheral devices. We collect HPM data using a very light-weight device driver that we have developed. Our monitoring overhead is less than 2%. The HPM driver is coupled with the oscilloscope. It collects HPM counters every 10 million instructions and uses an external output pin to control the oscilloscope and the data acquisition board. Throughout this paper, we use the term interval to refer to a period of 10 million instructions.

2.2 Computation Model

The computation model estimates the power consumption of tasks that execute within the boundaries of CPU. Our model employs 6

Event	Description
0x0	Instruction cache miss requires fetch from external memory.
0x1*	Instruction cache cannot deliver an instruction.
0x2*	Stall due to a data dependency.
0x3	Instruction TLB miss.
0x4	Data TLB Miss
0x5	Branch instruction executed, branch may or may not have changed program flow.
0x6	Branch mispredicted
0x7	Instructions executed
0x8*	Stall because the data cache buffers are full.
0x9	Stall because the data cache buffers are full.
0xa	Data cache access, not including Cache Operations.
0xb	Data cache miss, not including Cache Operations.
0xc	Data cache write-back. This event occurs once for each 1/2 line (four words) that are written back from the cache.
0xd	PC Modified

Table 2: HPM events in Intel PXA-255. The flagged events count the number of cycles the event condition persists.

parameters: Core clock cycles (x_1), instruction cache misses (x_2), instructions not delivered (x_3), data stalls (x_4), instruction TLB misses (x_5) and data TLB misses (x_6). We employ these events because of their close relationship program power consumption and because their use has been shown to be effective for power estimation of the CPU by prior researchers [**?**, 2, 3]. We have experimented with data cache access miss events and found that they did not contribute to improvements in the accuracy of our model.

Since the Stargate processor is only able to monitor two events at once, we must execute the same program many times to monitor the different events and to collect consistent measurements for the same events. The measurement data can differ across runs (of the same program/input) as a result of hardware state or operating system events. To mitigate the impact of these perturbations, we employ a set of statistical techniques that filter outlying data sets (execution profiles) – those that differ significantly from the rest.

2.2.1 Filtering Runtime Variability from Repeated Runs of the Same Program

A phenomenon that impacts the quality of program profiles generated using real devices with real applications and operating systems, is the noise that is introduced into the data due to asynchronous, transient, or otherwise uncommon and spurious hardware and software activities [**?**, 10]. Hardware interrupts, memory and I/O effects, system calls, virtual memory paging, and OS scheduling decisions, among other external factors, result profile variations of different magnitudes.

For applications such as phase tracking, applying advanced techniques that can recognize and eliminate these variations can soften their adverse side-effects [**?**, 10, 11]. However, in a modeling study such as ours, the accuracy of the model is strictly bounded by the quality of the collected data, i.e. the observations. Hence, our estimation system first filters the profile data prior to model construction (training benchmarks) or model evaluation (reference benchmarks). As a result, we are monitoring and estimating the behavior of a program without these perturbations (however infrequent). As part of future, work, we are considering how to handle them independently to improve the accuracy of our estimation system.

Each of our filters use the CPU clock cycle counter to analyze data set variance. We execute each program repeatedly (25 times in our experiments) using the same input, recording the cycle counter value for each interval (10 million instructions) of execution. We refer to the vector of intervals that each execution produces as CCNT. The k_{th} entry in CCNT is the clock cycle count for interval k.

We distinguish individual runs using a subscript on CCNT, e.g., $CCNT_4$ is the 4^{th} run of the program. We use n to mean the total number of runs that we profile as part of a single experiment (e.g., 25). When we refer to a particular interval within a particular run, we subscript CCNT with two variables ($CCNT_{i,k}$) to indicate the k_{th} interval in the i_{th} execution. For example, $CCNT_{4,2}$ is the number of clock cycles recorded in second interval of fourth run.

We introduce three novel filters that identify runs that are significantly different from the other runs using the CCNT vectors. Our first filter evaluates *correlation coefficients* (using Pearson's correlation [**?**]). Given that there are n CCNT vectors (for the n executions) we compute the correlation coefficient between all pairs of vectors. We store the result in a $n \times n$ square matrix. This matrix, to which we refer to as the *correlation matrix*, is lower triangular as the correlation operation is commutative.

Our algorithm makes multiple passes over this matrix. In the first pass, it flags any entry that is below a certain threshold (0.9 in our case). In the subsequent passes, it removes the column that has the highest number of flagged entries, together with the corresponding row. The algorithm continues removing one column and one row at a time until there are no flagged entries.

The correlation filter is similar to that used in prior work [**?**, **?**] and identifies many anomalies. We find though that the correlation filter is unable to capture a number of outliers. Figure 1 depicts such cases. The graphs show CPU clock cycles over time for five repetitions ($p1$ - $p5$) of a hypothetical program. In the left graph, $p2$ uses 33% more CPU cycles than $p1$. The correlation filter however, cannot recognize the difference since the clock cycle counts of $p1$ and $p2$ are fully correlated. Our comparison of total clock cycles is not helpful either, because the value is the same for all three programs. We can use the correlation between total execution time to eliminate p1; however, the correlation filter will still consider $p2$ and $p3$ to be the same.

To capture the differences between $p1$, $p2$, and $p3$, we employ an area filter. This filter first transforms the CCNT variable for each execution i via:

$$ACCUMCCNT_{i,k} = \sum_{l=1}^{k} CCNT_{i,l} \quad (1)$$

ACCUMCCNT is the number of accumulated clock cycles across all k intervals (i.e., since program execution commenced). Note that this transformation produces a monotonically increasing function. As ACCUMCCNT is a discrete function, we compute the area under the line using the summation function.

To identify outliers, we assume that our clock cycle observations (of which we have a large number) are normally distributed and apply the *z-test* [6] to remove any value that is 3 standard deviations from the sample mean. The area filter captures the differences between the functions in the left graph in Figure 1(a). It is not perfect however; for example, it (like the correlation filter) is unable to capture the differences in the data sets in the right graph.

We refer to our third filter as MADMSE. MAD stands for *median of the absolute deviations about the median* and MSE is *mean square error*. MADMSE first produces an average CCNT across all (n) runs for each interval (k) via:

$$MEANCCNT_k = \frac{1}{n} \sum_{j=1}^{n} CCNT_{j,k} \quad (2)$$

The MEANCCNT vector thus, records the *expected* number of CPU clock cycles for each interval. Next, the filter computes the mean squared distance between the cycle count in the ith run ($CCNT_i$) and MEANCCNT for each interval (k):

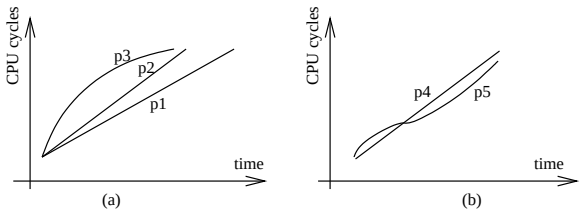


Figure 1: The figures show CPU clock cycles counts for repetitions of hypothetical program executions. In (a), even though $p1$, $p2$ are very different, they are perfectly correlated and pass the correlation filter easily. However, the area filter correctly identifies $p1$, $p2$ and $p3$ as three different variations. The area filter will not catch the differences between the data sets in (b).

$$MSE_i = \frac{1}{p} \sum_{k=1}^{p} (CCNT_{i,k} - MEANCCNT_k)^2 \quad (3)$$

The MSE function assigns higher scores to the outliers due to the square operation. We identify and remove the outliers using the *modified-z* test [6].

Since the sample mean and sample standard deviation considers all the observations, a single outlier can affect both significantly. Such conditions occur for example, during a file access for which the CPU cycle counts can differ by millions depending on whether the OS has buffered the data or not. The modified-z test substitutes sample mean with sample median, and sample standard deviation with MAD to reduce the effect of outliers. We compute MADE and the modified-z score, M_i, using:

$$MAD = median|MSE_i - \mu|$$

$$M_i = c(MSE_i - \mu)/MAD \quad (4)$$

where μ is the average MSE and $1 \leq i \leq n$ (i is the run and n is the total number of runs). The modified-z test specification recommends that we set c to 0.6745 and label observations as outliers if their $|M_i|$ score is greater than 3.5.

Our filters in combination eliminate much of the noise in the data sets of repeated executions for the same program/input. Our filters only capture noise due to variations in CCNT values. There are however, other sources of noise (such as power measurement errors due to the external apparatus). We expect these errors to be much less significant and ignore them for the scope of this paper (we plan to consider them as part of future work). Their impact contributes to the overall error of the model which is included in our overall results.

2.2.2 Empirical Computation Model Generation

We use a linear parametric function to model full system energy consumption. Linear models have been successfully used in previous studies to model energy consumption of various components including CPU and memory [**?**]. Our energy consumption model is as follows:

$$E(Joules) = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_r x_r \quad (5)$$

where x is the model input (i.e the HPM counter values), the α's are the weights determined by the model, and r is the number of parameters. We estimate energy consumption for periods of program executions (i.e. intervals) and combine them additively to produce an estimate of energy consumption for an entire program. We estimate the parameter weights using least squares linear regression

Computation Energy Consumption Model			
Coef.	Description	Ad-Hoc	K-Clustered
α_0	Constant	-8.478×10^{-3}	-1.265×10^{-2}
α_1	CPU cycles	7.496×10^{-9}	7.815×10^{-9}
α_2	Inst. Miss	-2.906×10^{-8}	2.380×10^{-7}
α_3	Inst. Not Dlvrd	-1.299×10^{-9}	-3.390×10^{-9}
α_4	Data Stalls	-1.064×10^{-9}	-1.428×10^{-9}
α_5	Inst. TLB Miss	-2.278×10^{-7}	6.097×10^{-7}
α_6	Data TLB Miss	-2.511×10^{-7}	-2.357×10^{-7}
R^2		0.998	0.999
Average Error		2.16%	1.92%
95^{th} Percentile Error		6.55%	5.40%

Table 3: Coefficient and fit statistics for the computation model. We evaluate two techniques for interval selection (Ad-Hoc and K-Clustered).

Communication Energy Consumption Model			
Coef.	Description	Bytes+Packets	Bytes
TXB	TX bytes	2.40×10^{-6}	6.29×10^{-6}
RXB	RX bytes	-4.78×10^{-7}	-1.69×10^{-6}
TXP	TX packets	-2.90×10^{-3}	
RXP	RX packets	5.50×10^{-3}	
K	Constant	2.37×10^{-2}	2.00×10^{-1}
R^2		0.972	0.796
Average		26.9%	208%
95^{th} Percentile		59.6%	470%
Wireless Idle Power		562 ± 146 mWatts	

Table 4: Communication Energy Model. We model the energy consumption of wireless card as a function of transferred bytes and packets.

of program energy consumption. LSQ models are simple, they do not require a priori knowledge of the distribution associated with the observations and they continue to perform well even when the model includes too many or irrelevant parameters [4].

We employ the NONET training benchmarks for model parameterization and LSQ. Since we do not want any single benchmark to represent more than its fair share (which is possible since the benchmarks execute for different durations), we choose an equal number of intervals from each benchmark. To extend the range of possible behaviors, we select the first, middle, and last 10 intervals from each profile. We refer to this as the *ad-hoc* method for interval selection.

We also employ an interval selection technique based on k-means clustering. Prior work employs k-means clustering for interval-based phase detection to improve the accuracy of phase detection [3]. We divide each application into 3 phases using the k-means clustering algorithm and select the first 10 intervals from each phase. For the situations where some phases had fewer than 10 intervals (transient phases), we increase the phase count until we had at least 3 phases with 10 intervals each.

We present the coefficients of our model and evaluate its fit on the training benchmark set in Table 3. The final two columns show the data for the ad-hoc and k-means clustering techniques. The top portion of the table shows the coefficients for each of the HPMs. The bottom portion of the table shows the fit statistics. We form the model using the training benchmarks and compute the fit statistics using the same benchmark and interval set. We evaluate the accuracy of our model for the reference set in Section 3.

An interesting phenomenon in this data is the existence of negative coefficients. All parameters except the clock cycle counter, have a negative coefficient. The reason behind the negative coefficients is *multicollinearity*. Multicollinearity exists whenever some linear relationship exists between the model parameters (as is the case for HPM data). As the amount of linearity increases, the stability of the coefficient estimates decreases. However, this does not invalidate the model quality or the model specification in anyway [4].

The coefficient of determination, i.e., the R^2 fit statistic, indicates the amount of variation that the model explains. Under most circumstances, it is a reliable indicator of model goodness. The R^2 varies between 0 and 1, and larger values are better. The high R^2 value of our model is a positive indicator of its high quality.

The average error statistic shows the absolute model estimation error. We compute this value by averaging the error for each observation and multiplying by 100. We compute error using $|\text{measured}-\text{estimated}|/\text{measured}$.

The model fits very well to the data with an average error rate of 2%. However average error does not describe how large the worst case estimation errors are. To describe this, we use the 95^{th} percentile statistic. We define the 95^{th} percentile error as the maximum absolute error for 95% of the estimations. Our model error rate is 6.55% or less for 95% of the cases. The k-means clustering improves the results only slightly. Due to the complexity of the k-means algorithm, we employ the ad-hoc method as part of our final model.

2.3 Modeling Wireless Network Communication

We next introduce our model for sensor network wireless communication. The Stargate employs a NetGear 802.11b wireless radio card which we model. Our model is independent from our computation model to enable portability, i.e., we can swap the model for others for comparison or to improve accuracy. We combine the models via arithmetic addition of the two estimates.

Modeling the network interface is more challenging than modeling the processing unit because the network interface is significantly impacted by external effects such as RF interference, network congestion, asymmetric links due to badly calibrated hardware, etc. We have not tested our radio model with all of these conditions and we do not expect it to perform well in extreme conditions. Furthermore, we assume an 11Mb/s communication rate setting for the purpose of this paper. We plan to incorporate other supported rates into our model as part of future work.

As we did for the computation model, we employ a wide range of empirical observations from benchmarks to develop our communication model. Our wireless network includes a set of 6 hosts, including PDAs and laptop computers. The network load varies from idle to a few megabits/second and is susceptible to interference from two separate wireless networks. Thus, we believe our model captures a wide variety of common situations.

The previous research on 802.11 networks has shown that the energy consumption of a wireless card his related to the transmit and the receive time. This information however is very low level and not easily accessible. Our system employs transfer size characteristics instead. We extract this information using software performance monitors (SPMs) that we deploy in the Linux operating system of the Stargate. The SPMs count the number of bytes and packets transfered as efficiently as possible.

Our model is a linear parametric function like the computation model, and has four parameters: transmit bytes (TXB), receive bytes (TXB), transmit packets (TXP), and receive packets (RXP).

The model uses these parameters as follows:

$$E_n(Joules) = TXB\beta_1 + RXB\beta_2 + TXP\beta_3 + RXP\beta_4 + K$$

Our NET training benchmark suite considers three different scenarios: (i) upload heavy communications (ii) download heavy communications and (iii) almost symmetrical, mesh type communications. For the first two scenarios, we use the scp benchmark. To collect behavior from the symmetric communications, we use the netpipe benchmark to generate network load. Typically, netpipe transfers are ping-pong like, it transfers one packet to a server and receives another packet before continuing. This forces the network to transmit every single packet, without opportunity to stream multiple small packets together. Netpipe also exposes idiosyncrasies that result from the internal hardware buffer, by re-evaluating each packet size using a constant perturbation factor. We consider for transfer size categories: (1) small: < 100 bytes; (2) medium: 100 to 1000 bytes; (3) large: 1000 to 4000 bytes; and (4) very large: 4000 bytes to 200KB. We repeat each transfer 100 times for the first tree categories and 10 times for the last category. As we do for our computation model, we use 30 intervals (first, middle, last) from each benchmark to construct the regression model.

We consider two different models, one that considers both bytes and packets transfered and one that only considers bytes transferred. We refer to the former as (Bytes+Packets) and the latter as (Bytes). We present the LSQ coefficients for both models as well as the fit statistics for the training data set for the selected intervals in Table 4.

Both models exhibit much higher error rates than those from the computation model. The error is due to the difficulty of capturing external effects. However, these results are for energy consumption of the wireless card only. In our evaluation section, we consider benchmarks that perform both computation and communication. The error for the latter will impact overall estimation depending on the amount of communication performed by the application (relative to computation).

Interestingly, these results show that it is very important to consider both packet count and bytes transfered to produce an accurate model. By extending the byte model to include the packet counts, we improve the error rate of the model by almost an order of magnitude. The low accuracy of byte model reflects the non-linearity between transfer sizes and packet sizes. Small packets have a disproportionately large overhead due to protocol headers.

2.4 Modeling Persistent Storage Access

The final component of our full-system energy estimation system models energy consumption due to accesses to persistent storage. In particular, we model the 256MB Sandisk Compact Flash card that the Stargate uses. Even though micro-drives are also useful for secondary storage, the constantly increasing capacity of flash cards makes flash storage more popular.

We measure and model the energy consumption of the compact flash card by reading and writing random data of various size. We avoid sizes that are multiple of the 64KB page size. To eliminate the effects of buffering, we force the file system to flush buffers using the *sync* system call before and after each operation. Figure 2 shows the data transfer sizes and measured energy consumption per unit byte.

The data in the figure exhibits stable consumption behavior both for reads and for writes, except for very small size transfers. At present, we ignore the very small file sizes as the operating system and compact flash buffering mechanisms hide how and when they are going to be written to a file. Instead, for files larger than 300

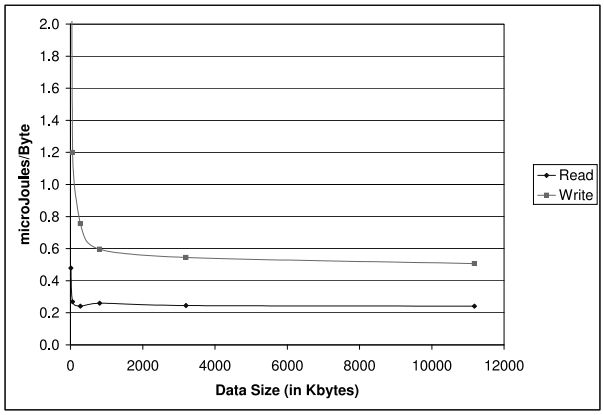


Figure 2: Compact Flash Energy Consumption Rate. The figure shows the measured energy consumption per byte for read and write in a 256MB Sandisk CF (fullness $< 10\%$).

KB, we model the energy consumption using this linear model:

$$E_w = size(inbytes) \times 4.99 \times 10^-7 + 0.094 \quad (6)$$

$$E_r = size(inbytes) \times 2.40 \times 10^-7 + 0.014 \quad (7)$$

Both models result in an R^2 value of 0.999.

3. EVALUATION

In this section, we empirically evaluate the efficacy of our techniques. In the subsections that follow, we first present results for our outlier filters. We then evaluate the accuracy of our computation model and compare it to an extant approach for estimation of CPU and memory power consumption. Finally, we present accuracy results for our combined models (computation, communication, and persistent storage access).

3.1 Efficacy of Variability Filters

To evaluate our filters, we collect profiles from our reference set (we also include Java-UCSD from our training set here). We execute each program 25 times for each experiment; in each experiment we collect observations of power, energy, SPM, and HPM data. Overall, we collect over 700 datasets. To present our results we use the NONET, NET, and FLASH groups that we specify in Figure 1 in Section 2.1. For the FLASH group, we execute the benchmarks (and inputs) using the compact flash file system (as opposed to RAM as we do for the NONET and NET groups).

We evaluate the quality of our filters using the number of outliers extracted by each of the three tests (Correlation, Area, and MADMSE). Typically, a receiver operating characteristics (ROC) curve illustrates filter quality, however, in our case, we have no baseline to which we can compare. We therefore, present outlier data and discuss individual cases.

Table 5 summarizes the overall results. Each entry shows the number of executions filtered out by a particular test. In the last row, we show the total number of filtered data sets and executions (in parentheses). Overall, we only filter a small subset of the experiments. This is important as it shows that such outliers are an infrequent phenomenon and yet we are able to recognize them.

Relatively, the filters reject a many more NET experiments than the others; the FLASH benchmarks have the least rejections. The former is due to the high variability in wireless network communication. MADMSE identified these cases. Interestingly, the other tests did not. This indicates the importance of employing MADMSE

	NONET	NET	FLASH
Correlation	4	33	1
Area	3	2	1
MADMSE	44	8	5
Combined	46 (400)	34 (225)	5 (75)

Table 5: Filter performance.

Category	Application	Correlation	Area	MADTEST
NONET	mpegencode	0	0	7
	jpegencode	1	0	7
	treeadd	0	0	1
NET	scprecv	25	0	0
FLASH	jpegdecode	0	3	3

Table 6: Applications with the most filtered data sets.

as part of the suite. The FLASH results are surprising given the non-determinism of compact flash writes. The impact of this non-determinism however, does not cause the overall performance data to differ significantly.

All applications have at least one data set that is rejected by the filters. The applications in Table 6, have a larger number of rejections. In the NONET set, mpegencode, jpegencode and treeadd have seven executions filtered out. In the NET group, the correlation test rejects all twenty-five executions – rejecting the entire benchmark.

In Figure 3, we present two executions of mpegencode. We took the data from the NONET test. In the figure, we plot the squared difference of observed and expected clock cycle counter values for two executions of mpegencode. The execution labeled `outlier` fails MADMSE. The outlier curve exhibits higher variance (difference from the expected value).

The scprecv is an interesting benchmark. MADMSE and the area test do not detect any anomalous data sets, however, the correlation test does. We find that in scprecv the correlation of CCNT vectors is quite low, $0.6 < r < 0.8$. For this particular benchmark we also observe low correlation with other statistics such as power and energy. Across all of our applications, scprecv is most susceptible to external effects. As scprecv downloads a large file from network, its performance, and execution characteristics reflect the data transmit rate of the remote host, noise in the network, etc.

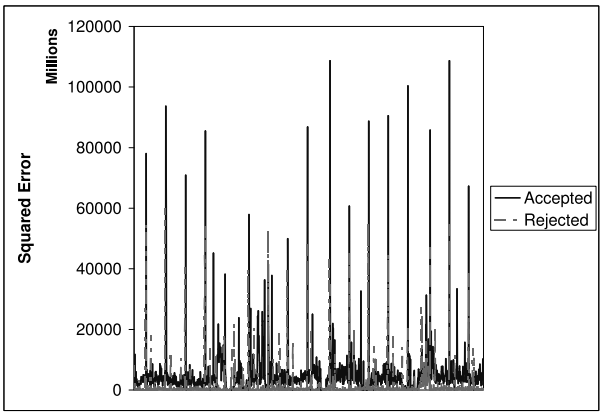


Figure 3: Deviation in Mpegencode. The figure compares the deviation curves from two mpegencode executions. The MADMSE test correctly identifies the outlier due to its high deviation.

Even though the tests are very sensitive, they have been designed to handle only the changes in CCNT vectors. Even though the clock cycle counter is a very powerful indicator program behavior, it does not explain the whole behavior of the system. An interesting example of this limitation occurs in bisort. Figure 4 shows the clock cycle counter (left graph) and energy (right graph) observations for five repetitions of the bisort Java benchmark. We observe similar behavior CPU clock cycle counts for all the five executions however, the energy consumption for I4 is significantly different from the others – it seems to be an offset of the others. At approximately 140 million instructions (time), the energy consumption of all programs increase; this increase is much larger I4. As there is no network activity and no other programs running, we believe this may be a result of some hardware state. Since our filters only consider CCNT values, they cannot catch such anomalies. If we remove this particular execution from the dataset we improve the accuracy of our estimation for bisort by more than 30%. We plan to investigate filters for such (non-CCNT-based) anomalous behaviors as part of future work.

3.2 Accuracy of the Computation Model

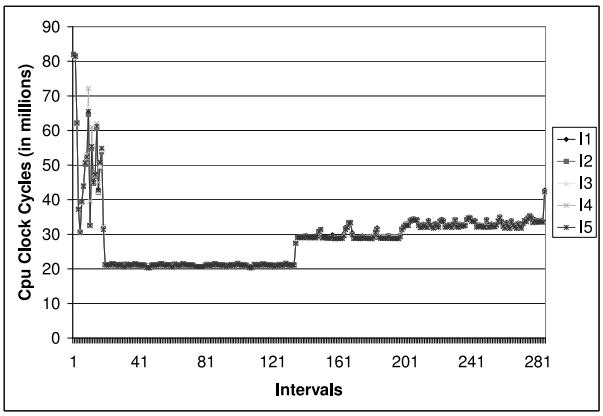
We next evaluate the accuracy of our computation model. In Section 2.2, we present results on the fit of the model which we generate using training data set from which we form the model. Moreover, this study only considers the performance data from the intervals that we select for model parameterization. Our results indicate that our model produces estimates with an average error of 2% across benchmarks. The error is the difference between the estimates produced by the model and the observed data.

We evaluate the computation model that we describe herein and compare it to a model from prior work [3] to which we refer to as CPUMEM. CPUMEM is the best-performing model in the literature for estimating the power consumption for the Intel XScale CPU and memory. Moreover, it is very similar in construction to our own model (to which we refer to as CompMod). Both are developed for the same Intel XScale CPU and memory configuration and they employ HPM counters for parameterization. Our model however, is intended to estimate the energy consumption of the entire device (and thus we use system energy data for model construction), whereas the previous model is intended to estimate the power consumption of the CPU and memory system in isolation. The evaluation of the prior model shows that it is highly accurate for doing the latter (the authors report a 4% error rate on average).

Our goal with evaluating this model from prior work for our *computationally bound benchmarks only* is to see how well it is able to predict the energy characteristics of the full system for these programs. We expect this model to perform well since the CPU and memory subsystem consume a large portion of the full system energy for these devices for computationally intensive programs.

Figure 5 shows show the error percentage for the full training (left graph) and reference (right graph) benchmark sets. This data employs all of the NONET (computation-bound) benchmarks as well as all of the intervals that execute for each. Our model is demarked CompMod and the model from prior work is demarked CPUMEM. We compute the error rate as we describe in Section 2.2. For the training set, our model achieves a 2% error rate on average, with a worst case error of 6%. The average error rate for CPUMEM was 65%, with a worst case error of 81%. For the reference set, our model achieves a 3% error on average, with a maximum of 9%.

The poor performance of CPUMEM emphasizes the importance of taking a global view of energy estimation. The best case for the CPUMEM model is em3d (17% error rate). Our investigation into CPUMEM accuracy shows a high correlation the error rate and



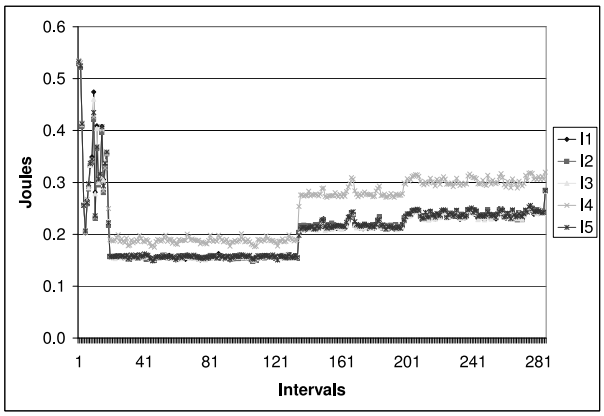
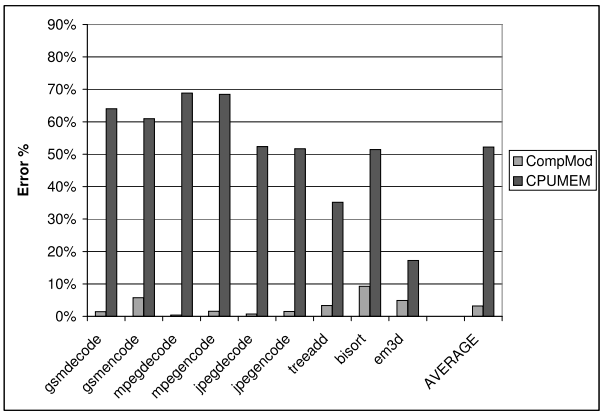


Figure 4: Bisort behavior. The left graph shows CPU cycle count data for five executions of bisort. The right graph shows the energy consumption measurements for the same five executions. One execution is clearly an outlier, however our variability filters are unable to capture reject this data set since there is no change deviations CPU cycle count.



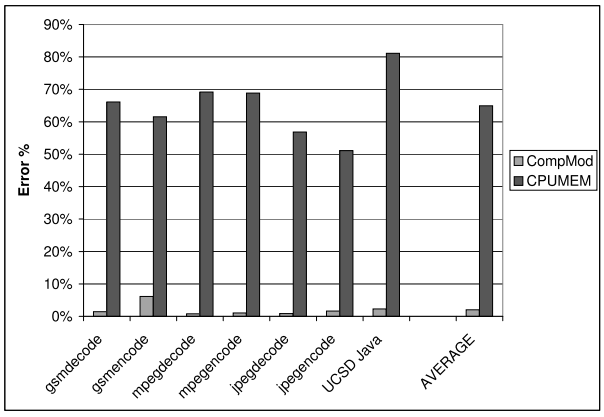


Figure 5: Error rate for the computation model. The left graph shows the results for the training benchmarks and the right graph for the reference benchmarks. The light bar is the accuracy of the model we present and the dark bar is the error rate for a competitive approach for estimation of CPU and memory power consumption. The results indicate that the latter cannot be used to predict accurate the full system energy consumption. Our model achieves an average error of 2% for the training set and 3% for the reference set.

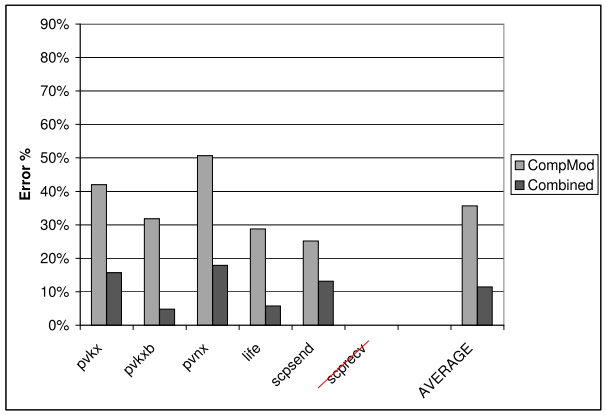


Figure 6: Error rate for the combined computation and communication models for the reference NET benchmarks. CompMod is the computation model alone.

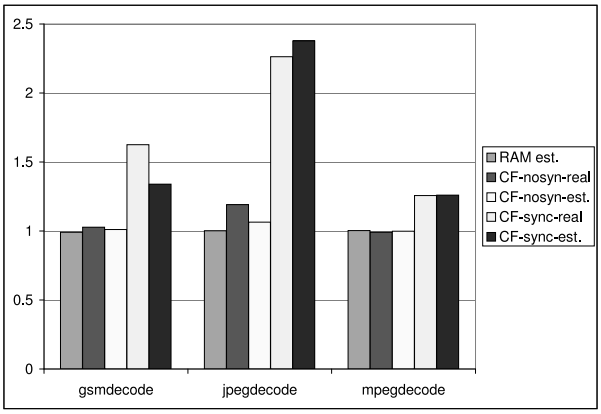


Figure 7: Error rate for the combined computation and persistent storage access models for the reference FLASH benchmarks.

the data TLB misses (we omit the data due to space constraints). This may indicate that that the poor performance is the result of underestimation of CPU and overestimation of memory cost.

3.3 Computation and Communication

We next integrate the computation and communication model and evaluate the accuracy of the combined model. We estimate the energy consumption using $E_t = E_l + E_n$ where E_t is the total energy consumption, E_l is the computation model output and E_n is the network model output.

We evaluate our model using the NET reference benchmarks. The benchmarks exhibit (i) upload heavy, (ii) download heavy, and (iii) mesh like scenarios. We have selected these scenarios since they represent a wide set of sensor network applications. In terms of network activity, the total amount of data transfer is 2MB for pvkx, 550KB for life, 2.1MB for pvnx, 1.8MB for pvkxb and 1.7MB for scpsend.

Figure 6 shows the percent error for the benchmarks when we use the combined model (dark bar demarked Combined). We also include the results (light bar demarked CompMod) for when we use the computation model alone – to investigate the importance of considering full system behaviors to estimate energy consumption. In the figure, we omit scprecv since, as we explained previously, all of the data sets were rejected by our outlier filters due to high variability.

The average error for the combined models is 11%. If we only employ our computation model, the average error is 35%. The best case for our combined model is pvkxb and life, with error rates of 4.7% and 5.7%, respectively. Our worst case is pvnx with an error rate of 17.9%. Given the high variability in network performance we expect a higher error rate for applications that exercise the network. As part of future work, we plan to investigate ways of improving our network model by considering multiple transfer rates, noise, and multiple wireless cards.

3.4 Computation and Persistent Storage

To evaluate our permanent storage model, we employ our FLASH reference benchmarks. We combine our persistent storage access and computation model as we did for communication and computation in the prior section ($E_t = E_l + E_{fw} + E_{fr}$). E_l is the estimation from the computation model; we produce E_{fr} and E_{fw} using Equations 7 and 6, respectively.

We perform two experiments. First, we collect data without interfering with the OS and file system, thus the operating system manages and caches the file as usual. Second, we force the operating system to flush its buffers before and after each execution.

Figure 7 compares the energy consumption of applications (normalized to the observed energy consumption for RAM-only execution). The bars show estimated energy consumption (i) estimated when running in RAM (RAM est), (ii) observed when running in compact flash with file system buffering (CF-nosyn-real), (iii) estimated when running in compact flash with file system buffering (CF-nosyn-est), (iv) observed when flushed (CF-sync-real), and (v) estimated when flushed (CF-sync-est). To our surprise, there is no significant difference between the applications that run on the RAM and the applications that run in flash memory with buffering. However, when we force the file system to flush, we increase the cost of gsm and jpeg more than 60%.

4. RELATED WORK

In our work, we model full system energy consumption for sensor network gateways. Our model uses hardware and operating system monitors to estimate the power consumption of a complete device, the Crossbow Stargate. The work most related to our own is on HPM-based models for CPU and memory energy estimation [**?**, 2, 12, 11, 20, 13, 9, 3]. In recent work, Bircher et al. [2] presents a power model for the Pentium-IV class of processors. They develop their power model using least squares regression (LSQ) [4]. Their model is different than ours for computation in that it estimates only CPU energy consumption and uses two performance counters. Our work shows that using such a model for full system energy estimation is highly in accurate even for computation-bound programs.

In embedded systems, the most similar study to our own is by Contreras et al. [3]. In this work, the authors use LSQ to develop a power model for an Intel XScale processor attached to a development board. Using this model, the authors are able estimate CPU energy consumption with a 4% error rate. However, their efforts to construct a memory power model did not perform as well due to the lack of hardware counters in the CPU that count memory events. In our work, we extend (and compare) this HPM model to estimate full system energy consumption by employing software counters and a statistical filtering process.

Another line of research that is related to our study is focuses on run-time variability. Run-time variability is an important problem when program behavior is observed repeatedly. Recently, Isci et al. [10], analyzed program power phases of applications using

real devices. In this context, the authors categorize run-time variabilities as time-shifts, time dilations, phase mutations, transitional glitches, and gradients. However, instead of identifying and eliminating variability, their technique assumes that these variabilities are an integral part of the model. We eliminate outlying data sets and focus our model development on the filtered data set. The number of data sets we eliminate is small. As part of future work, we plan to couple our model with one that characterizes the energy consumption of outliers. We believe that we must address them separately to achieve highly accurate estimations.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we present a system for accurately estimating the power consumption of the Crossbow Stargate intermediate sensor network device. Our system couples statistical techniques that employ empirical data to model various program activities including computation, communication, and persistent storage access. We collect the data using a subset of the hardware performance counters available on the device and using software performance monitors in the Linux operating system. We gather the data using a large number of repeated runs of programs that exhibit behaviors typical of sensor network tasks. To mitigate the perturbation in the profiles caused by isolated, asynchronous, and transient events in the hardware and operating system, we present a set of statistical techniques that effectively filter outliers from the data to improve the quality of our model.

We implement our models for computation, communication, and persistent storage access using this filtered data. By doing so and by combining these models into a single energy estimation, we are able to accurately characterize program energy consumption at the device level. We evaluate our model using a wide range of benchmarks and validate it using high-precision measurements of power and energy. Our model achieves an error rate of of 3% on average for computational tasks and of 11% on average for tasks that use both computation and communication. We achieve similar error rates for tasks that employ computation and flash file system access. We compare our approach against a prior model that employs HPM data to model the CPU and memory subsystem of the Stargate. We find that this prior model is not effective for predicting the energy consumption of the full system.

As part of future work, we plan to improve our model by considering other sensor task activities such as serial communication (a typical Stargate to mote path). We plan to investigate additional programs to test further the robustness of our system in terms of accuracy. In addition, we plan to identify a model that accurately estimates the power consumption of outliers. Our current filtering system eliminates such data sets from our empirical data. Although, these sets are very few in number, we believe that we can couple the system herein with an additional model (or set of models) that recognizes outliers and estimates their power consumption. Finally, we plan to implement our prediction system as part of the Linux OS in the Stargate and iPAQ handheld to enable online estimation of power consumption as part of energy-aware compiler and runtime optimizations.

6. REFERENCES

[1] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. In *In Proceedings of Design, Automation and Test in Europe*, 2000.

[2] W. L. Bircher, M. Valluri, J. Law, and L. K. John. Runtime identification of microprocessor energy saving opportunities. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 275–280, New York, NY, USA, 2005. ACM Press.

[3] G. Contreras and M. Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 221–226, 2005.

[4] R. Freund and P. Minton. *Regression Methods, A tool for Data Analysis*, volume 30. Marcel Dekker, INC, 1979.

[5] S. Gurun, C. Krintz, and R. Wolski. Nwslite: a light-weight prediction utility for mobile devices. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 2–11. ACM Press, 2004.

[6] B. Iglewicz and D.C. Hoaglin. *How to Detect and Handle Outliers*. ASQC Press, 1993.

[7] Intel Corporation. *Xscale*. www.intel.com/design/intelxscale/.

[8] Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. In *WWC '03: Proceedings of the Sixth International Workshop on Workload Characterization*, 2003.

[9] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO '03: Proceedings of the 36th ACM/IEEE International Symposium on Microarchitecture*, 2003.

[10] Canturk Isci and Margaret Martonosi. Detecting recurrent phase behavior under real-system variability. In *IISWC '05: Proceedings of the 2005 International Symposium on Workload Characterization*, 2005.

[11] Canturk Isci and Margaret Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *HPCA '06: Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[12] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140, New York, NY, USA, 2001. ACM Press.

[13] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin, and A. Sivasubramaniam. vec: virtual energy counters. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–31, New York, NY, USA, 2001. ACM Press.

[14] C. Krintz, Y. Wen, and R. Wolski. Application-level Prediction of Battery Dissipation. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, August 2004.

[15] Olaf Landsiedel, Klaus Wehrle, and Stefan Gtz. Accurate Prediction of Power Consumption in Sensor Networks. In *In Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005.

[16] D. Rakhmatov and S. Vrudhula. Time-to-failure estimation for batteries in portable electronic systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2001.

[17] Victor Shnayder, Mark Hempstead, Bor rong Chen, and Matt Welsh. PowerTOSSIM: Efficient Power Simulation for TinyOS Applications. In *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

[18] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner-Allen, and Matt Welsh. Simulating the Power Consumption of Large-Scale Sensor Network Applications. In *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

[19] K. C. Syracuse and W. Clark. A statistical approach to domain performance modeling for oxyhalide primary lithium batteries. In *In Proceedings of Annual Battery Conference on Applications and Advances*, January 1997.

[20] Andreas Weissel and Frank Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246, New York, NY, USA, 2002. ACM Press.

[21] Y. Wen, R. Wolski, and C. Krintz. History-based, Online, Battery Lifetime Prediction for Embedded and Mobile Devices. In *Workshop on Power-Aware Computer Systems (PACS)*, April 2003.