Application-level Prediction of Battery Dissipation

Chandra Krintz Ye Wen Rich Wolski
Computer Science Department
University of California, Santa Barbara
{ckrintz,wenye,rich}@cs.ucsb.edu

ABSTRACT

Mobile, battery-powered devices such as personal digital assistants and web-enabled mobile phones have successfully emerged as new access points to the world's digital infrastructure. However, the growing gap between device capabilities and battery technology requires novel techniques that extend battery life. Key to the success of such techniques, is our ability to accurately predict the power consumption of a program.

*In this paper, we investigate the degree to which battery dissipation induced by program execution can be measured by application-level software tools and predicted by a compiler and runtime system. We present a novel technique with which we can accurately estimate whole-program power-consumption for an arbitrary program by composing battery dissipation **rates** of benchmarks. We empirically evaluate our technique using an iPAQ hand-held device and a number of MiBench and other programs.*

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms

Measurement, Performance

Keywords

Battery Life Estimation, Resource-restricted devices, Application-level prediction

1. INTRODUCTION

While hand-held, battery-powered devices have emerged as new access points to the world's digital infrastructure, their cost and short battery life are factors that are holding back their enormous potential. While economic factors will reduce the former, mechanisms are needed to enable executing programs to adapt to dwindling battery life. Many such techniques have been proposed that facilitate energy conservation through different modes of operation at both the device and device component level, i.e., active, idle, standby, and sleep modes [2, 16, 13, 12, 25]. Other techniques select instructions based on their energy consumption [14, 22, 23, 21].

Key to the success of such techniques, is our ability to accurately *predict* the power consumption of a program. Much work [22, 23, 14, 21, 15] has focused on the power consumed by a CPU when executing a particular instruction. We observe, however, that the battery dissipation caused by a program when running on a hand-held device will involve many internal subsystems. Further, even if efficient models for all subsystems are available, their independence is not obvious making a comprehensive compositional model potentially complex.

We address the problem of predicting program power consumption from a different perspective. Our method relies on app-lication-level observations of *battery dissipation* for a representative set of benchmarks when running on the entire device (and not any subsystem in isolation). We show how these benchmark dissipation rates can be combined to form an estimate for an arbitrary program. By observing the power consumed by the whole device as a "black-box", our technique does not require a composition of subsystem models. At the same time, we use only measurements that are available via standard operating system interfaces making the methodology practical for implementation in a runtime compilation system using currently available hardware, i.e., without new hardware features for measuring power consumption. For this study, we use the iPAQ hand-held device with a StrongARM SA-1110 processor [6] — a popular Personal Data Assistant (PDA) that is commonly available.

To combine individual benchmark values into an estimate for a non-benchmark program, our work

- Identifies the relevant set of instruction categories that are necessary to make accurate battery dissipation estimates for the iPAQ.
- Demonstrates the way in which benchmark readings for these categories can be *composed* into a dissipation estimate for a target program.
- Details the accuracy of these estimates by comparing them to observed dissipation values for a set of target application programs.
- Presents empirical, non-simulated, results that show that with relatively few instruction categories, accurate estimates of battery lifetime can be derived.

As such, our results attempt to describe, as directly as possible, the efficacy that would be observed by a dynamic (runtime) compilation system deployed on currently available devices.

ISLPED'04, August 9–11, 2004, Newport Beach, California, USA.

In the next section, we provide an overview of our approach and describe the empirical platform that we used for this study. We then describe our benchmark programs and present observed battery drain curves for each. In Section 4, we then detail how we *compose* battery dissipation curves from the benchmarks to predict the drain behavior of an arbitrary program. We then present the efficacy of our approach using empirical data in Section 5 and conclude in Section 6.

2. APPLICATION-LEVEL POWER PREDICTION

Our objective with this work is to determine the degree to which the impact a program has on *battery dissipation* (more specifically, the battery drain curve) can be *predicted* using *appli-cation-level measurements* of battery drain that could be made available to a runtime compilation system. Note that the battery dissipation of a program is a compositional effect caused by both the program's energy consumption and the characteristics of the battery itself. Previous work [22, 23, 14, 21, 15] has studied the power consumption characteristics at the processor instruction level. Combined these with battery models [7, 3, 10, 19, 18, 17, 20], one can estimate the battery dissipation of a program. However, due to the limitations of battery models, such as high computational cost and complex parameterization [26], compositional model-based methods are not suitable for runtime compilation systems. Furthermore, it is unclear how the composition process will impact prediction accuracy.

In our work, we chose a black-box approach in which we attempt to observe the dissipation characteristics at the application-level. In particular, we profile the battery drain behavior for a set of hand-coded benchmarks that each execute a single type of instruction. We then use this data to estimate the impact that the execution of an arbitrary program will have on battery lifetime.

To predict the battery dissipation behavior for an arbitrary program, we compose the dissipation curves obtained from our benchmark profiles. Note that we are *not* computing the consumption of individual instructions in the program from the benchmark measurements. Since the dissipation curve changes over time, so does the impact an instruction will have on remaining battery life. As such, we construct a complete battery dissipation curve for the target application. We compute the rate of drain for an arbitrary program directly from the rate of drain of the benchmarks that implement the constituent instruction types of the programs. A compilation system can then use this constructed dissipation curve to extract an estimate of battery dissipation for the program, given the current battery level.

We do not consider the effect of using the display, wireless communication, or file I/O. Our initial goal is to understand whether battery dissipation curves from single-instruction benchmarks can be used to estimate the dissipation curve of arbitrary, more complex program power consumption behavior. We plan to investigate other hand-held subsystems as part of future work.

In the next section, we describe our hand-coded benchmarks and their battery consumption characteristics. First though, we detail the experimental methodology that we use for the benchmark performance results that we present.

Experimental Platform and Methodology

To determine the observable power consumption characteristics of applications, we chose the Compaq iPAQ H3600 personal digital assistant (PDA) as a test platform. The iPAQ's processor is the StrongARM SA-1110 that uses two on-chip data caches (DCache) and one on-chip 16KB instruction cache (ICache). The device can use AC or DC power; the battery that supplies the latter is a Danionics Lithium-Ion Polymer Battery (#DLP 305590) [1]. The voltage range for the battery is specified as 3.0 to 4.2 Volts.

We employ *Familiar* Linux [9] 0.5.1 with kernel version 2.4.16-rmk1. *Familiar* implements battery management using the Hardware Abstraction Layer (HAL) which exports battery data via the /proc file system. The data values exported by HAL can be directly converted to millivolts: Given the voltage range of our battery and observed HAL maximum and minimum values of 953 and 705, respectively, we multiply the HAL raw data value by 4.2 to compute millivolts. A similar computation is performed by the *Familiar* kernel for power management and visualization facilities [5]. We use millivolts throughout this text (since it is the metric exported) to describe battery level.

3. BENCHMARKING BATTERY DISSIPATION BEHAVIOR

Our methodology uses a set of observed drain-rate curves from a suite of *benchmarks* to determine the *battery dissipation rate* associated with a particular kind of instruction. Using the observed dissipation curves for individual instruction categories, we compose an estimate of overall program dissipation for an arbitrary program. Throughout this text we distinguish *benchmarks* from *programs* in this way.

We identified four general categories of relevant instruction types: integer register operations, integer loads and stores, floating point register operations, and floating point loads and stores. In the remainder of this text, we refer to these benchmarks as **IReg**, **IMem**, **FPReg**, and **FPMem**, respectively. In addition, we examine the effect of cache-only data access versus full memory subsystem access. To do so, we varied the address range of the IMem and FPMem benchmarks between 8000 bytes (cache partially filled), 16000 bytes (cache full) and 32000 bytes (complete cache flush). We refer to the in-cache versions of the IMem and FPMem benchmarks as **IMem_Cache** and **FPMem_Cache**; in addition, the address range in bytes are given in context. We verified that all benchmarks exercised only the CPU and memory subsystems we intended using a StrongARM version of the SimpleScalar simulator [4]. All of the results we present, however, were generated by executing directly on the device.

In addition, we developed memory benchmarks that implemented only loads or only stores to determine whether the difference between battery consumption for loads and stores is significant. The IMem, FPMem, IMem_Cache, and FPMem_Cache use only load instructions. We developed IMemW and IMemW_Cache benchmarks that use only store instructions. We did not implement **FPMemW** or **FPMemW_Cache** since, on the iPAQ we chose there is no floating point (FP) unit (FP operations are implemented via a trap instruction). As such, we assume that FP loads and stores have equivalent battery dissipation characteristics. The entire set of benchmarks is freely available.

To measure benchmark battery dissipation, we modified our benchmarks so that each looped infinitely. We then fully charged the iPAQ battery (to approximately 4000 mV) and executed each benchmark until the battery died (at approximately 3000 mV). We periodically polled (every 20 seconds) the Linux HAL resource interface and logged the result. We performed this experiment repeatedly for each of the benchmarks. We varied the array sizes (address ranges accessed) for both the in-cache and out-of-cache benchmarks; we report on only 8000B (IMem_Cache) and 32000B (IMem (out of cache)) here for brevity. However, curves for other array sizes were nearly identical to the in-cache and out-of-cache representatives that we present here.

We present a set of results that is representative of those we col-

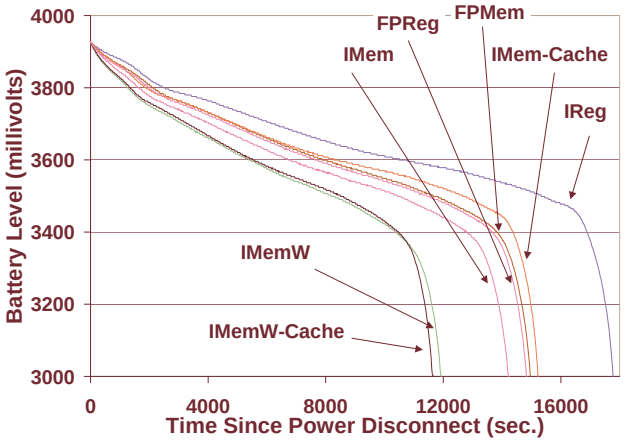


Figure 1: Comparison of battery drain rates for our benchmarks: IReg, FPReg, IMem, and FPMem. The memory benchmarks implement different types of memory accesses: cache-only (8000B) and memory only (32000B).

lected in Figure 1. The x-axis is the time in seconds since power was disconnected. The y-axis is the percentage of battery available as reported by HAL (converted to millivolts). We include arrows to help distinguish the different benchmark dissipation curves.

The graph contains many interesting details. First, as expected, the rate at which the battery is consumed is considerably slower when registers (IReg) are used than when the memory system is accessed. Shutdown occurs 3553 seconds earlier for IMem than for IReg. Secondly, floating point register operations consume battery power at a rate very similar to that of floating point loads and stores.

We do not include FPMem_Cache in this graph for clarity. However, the curves exhibit similar behavior to FPMem. Likewise FPMem and FPReg are very similar. This is due to the lack of a floating point unit: Each floating point instruction traps to the operating system kernel which uses library routines to emulate floating point operations. As such, in the remainder of this study, we consider all floating point operations equal: We predict the consumption rate of these operations using only the FPReg benchmark.

Next, load instructions that miss the cache (IMem) drain the battery 993 seconds earlier than those that hit the cache (IMem_Cache). However, this relationship does not hold for in-cache and out-of cache store instructions. This seems to indicate that our benchmark causes the iPAQ to exhibit ”write-through” behavior. This is due to a combination of the ”no write-allocate” cache implementation of the iPAQ’s StrongARM processor. and our IMemW benchmark implementation: Since we only use store instructions, a load is never executed and hence a cache line is never allocated.

As with floating point operations, we assume that all stores are equal and as such predict their consumption using the IMemW benchmark. We detail the implications (in terms of prediction error) of these assumptions in Section 5. In summary, the benchmarks consumption rates that we will compose to make predictions in this study are IReg, IMem, IMem_Cache, IMemW, and FPReg.

4. COMPOSING BENCHMARK POWER CONSUMPTION RATES

Given the HAL millivolt curves for the benchmarks, we set out to predict this curve for arbitrary programs using only our benchmark data and various program execution statistics. The latter includes program execution time for a single run, and the percentage of time spent executing in each of aforementioned categories (integer register operations, floating point operations, integer loads, and integer stores). For example, if a program spends 30% of its execution performing memory operations and 70% on integer operations, we use these times to find the corresponding dissipation in each benchmark curve (IReg and IMem).

For our predictions to be accurate, the dissipation rate for register instructions and memory instructions must *compose*. That is, the battery dissipation curves for benchmarks implementing individual instruction categories must sum to equal the battery dissipation curve of the program of interest. To our knowledge, this is the first such work that evaluates empirically the degree to which battery dissipation composes to form the overall dissipation for arbitrary programs *given only application-level information*.

Notice that the HAL millivolt curves in Figure 1 are not linear; they drop off sharply when battery power gets low. As such, a dynamic compilation or runtime system will require a predicted *drain rate curve* from which it can extract an estimate of battery dissipation given the current battery level.

This methodology assumes that we have battery dissipation samples for the benchmark drain curves at a sufficiently fine granularity. However, since the measurement itself consumes energy, we cannot sample too often. For this study, our sample rate is every 20 seconds. Given this frequency, we are unable to make predictions at a granularity finer than 20 seconds. As such, we construct our predicted rate curve in a piecewise fashion, using the average millivolt change over each 20 second interval.

Notice also that we are only able to compute the predicted rate curve until the underlying benchmark rate curves terminate. That is, when the IMem benchmark ends (which is earlier in time that our IReg benchmark curve), our predicted rate curve ends also. As such, our predicted rate curves will terminate earlier than do the observed curves.

5. EMPIRICAL EVALUATION

To verify (or disprove) this thesis (that the battery dissipation curves of individual instructions composes for arbitrary combinations of different types of instructions), we predicted and observed the dissipation curves of seven C programs from the MiBench embedded program suite [11] and from hand-coded implementations of other well known algorithms from [8]. We first describe our experimental setup and then present our results.

5.1 Experimental Setup

The programs that we used to evaluate our technique and their various statistics are shown in Table 1. Column one is the execution time of each program in seconds. The second column is the dynamic instruction count (IC) in millions of instructions. The third through sixth columns show the percentage of these dynamic counts that constitute each of the four instruction categories: integer register operations (IReg), integer loads (IMem), integer stores (IMemW), and floating point operations (FPReg). As mentioned previously, we consider floating point loads and stores equivalent to floating point register operations (FPReg) in terms of battery drain rate. Programs with 0 in the FPReg column perform no floating point operations and as such, are integer programs (BitCount, Dijkstra, and MMult); all others are floating point programs. In addition, we executed each program using the StrongARM version of the SimpleScalar simulator to determine L1 DCache miss rate for the programs. On average the miss rate is 1.4%.

The last column in the table shows the observed millivolt battery drain for a single run of each program when invoked with a battery

Program	Exec ET (secs)	Dyn Insts (*1M)	I-Reg (pct)	I-Mem (pct)	I-MemW (pct)	FP-Reg (pct)	mV drain
BasicMath	153.96	214	57	11	12	20	7.35
BitCount	46.56	6576	50	32	17	0	2.12
Dijkstra	39.43	5061	45	49	6	0	1.88
FFT	121.73	341	51	9	13	27	6.09
LU	92.31	302	44	31	6	18	4.03
MMult	18.14	1789	70	27	3	0	1.04
QSort	45.29	161	73	8	15	4	2.32
Average	73.92	2063	56	24	10	10	3.55

Table 1: Execution statistics for the programs used for our empirical evaluation. The last column shows the observed millivolt battery drain for a single run of each program when invoked with an arbitrary battery level of 3864mV (almost fully charged).

level of 3864mV (almost fully charged). The starting point (battery level) is arbitrary and we include the values to give the reader an example of program battery dissipation for a single execution.

We measured each program (modified to loop infinitely) as we did for the benchmarks to obtain an *observed* dissipation curve. We then compared this curve to the predicted curves we obtain by composing the drain curves from the constituent instruction types.

5.2 Results

The integer programs (BitCount, MMult, Dijkstra), implement two types of instructions: register operations and memory operations. As such, we composed the dissipation rate curves of the IReg and the IMem benchmark to construct the predicted drain curve. By using IMem, we are assuming that loads and stores in the program consume battery power at a rate equivalent to our benchmark that performs only loads that miss the cache. We refer to the resulting predicted curve as IReg-IMem. We also computed this predicted rate curve using IMem_Cache instead of IMem. This configuration assumes that loads and stores in the program consume battery power at a rate equivalent to our benchmark that performs only loads that hit in cache. We refer to the resulting predicted curve as IReg-IMem_Cache. These curves (IReg-IMem and IReg-IMem_Cache) provide a lower and upper bound on prediction error, respectively.

For the floating point programs (BasicMath, FFT, LU, QSort), we computed the predicted curves using the drain curves from three benchmarks: IReg and IMem only (IReg-IMem), IReg and IMem_Cache only (IReg-IMem_Cache), and IReg, IMem, and FPReg only (IReg-IMem-FPReg).

We first present four of the seven resulting curves in Figure 2 (two integer (top row) and two floating point (bottom row) programs). We omit the graphs for the other programs due to space constraints. These graphs, however, are representative of both types of programs that we studied. We report error rates of the predicted curves for all benchmarks at the end of this section.

The *Observed* curves show the measured drain rate curves; the *Predicted* curves show the predicted drain rate curves that we composed using the benchmarks described above. The x-axis in each graph is time (in seconds) since the battery was disconnected. The y-axis is the battery life in millivolts exported using the HAL interface.

For the integer programs, we can observe that the IReg-IMem predicted curves are nearly indistinguishable from the observed curve for all benchmarks. As such, these results indicate that for integer programs, benchmark battery dissipation for constituent instruction types can be composed to *accurately* predict the dissipation of arbitrary programs.

For the floating point programs, IReg-IMem and IReg-IMem_Cache again bound the observed curves (providing lower and upper bounds, respectively, on prediction error). When we include FPReg in the composition, the resulting predicted curve is remarkably similar to the observed curves. This set of results indicates that floating point operations should be considered in battery dissipation prediction. In addition, doing so results in an accurate prediction of floating point program battery dissipation.

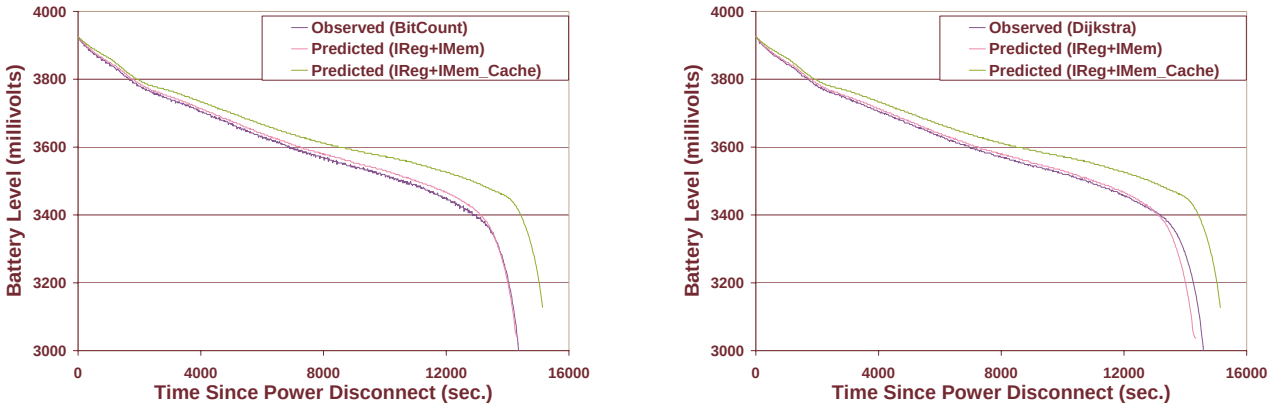
We next present results for all of the programs in terms of prediction error. Table 2 shows the errors in millivolts for the various prediction techniques for both integer and floating point programs. In addition to the various compositions shown in the above graphs, we also provide error values that result when we consider store instructions.

The first seven columns of data in the table show the mean absolute error of the predictions over the predicted drain curves. IReg-IMem, IReg-IMem_Cache, and IReg-IMem-FPReg are the same as presented in the graphs above. IReg-IMem-IMemW prediction uses the IReg curve to compute the drain due to the percentage of integer register operations, the IMemW curve for the percentage of stores in the program, and IMem for all other instructions; this assumes that all memory accesses miss the cache. Since the IMemW curve ends at 12000s, so does our prediction (and error measurement). The final row of data shows the average values. For the IReg-IMem-FPReg columns we only average the values of the floating point programs (BasicMath, FFT, LU, QSort).

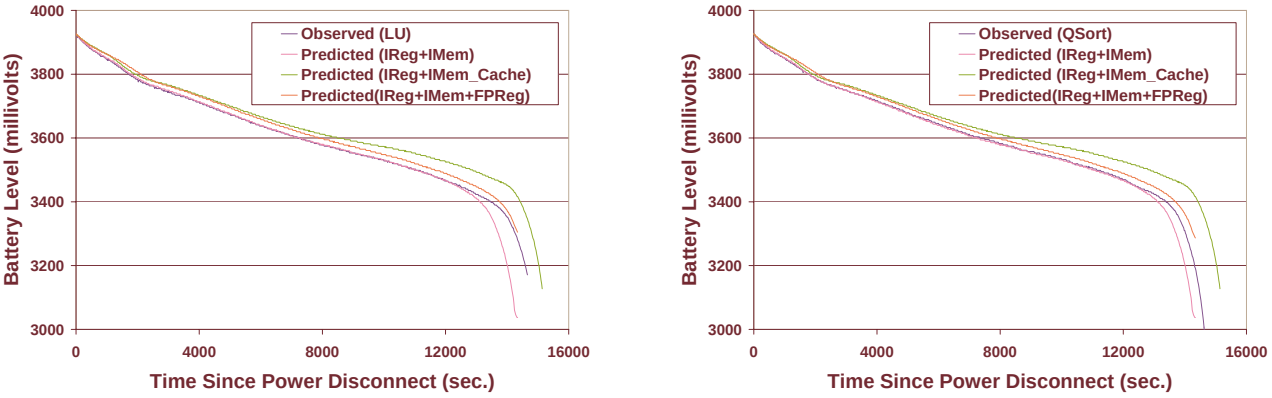
Each of these data sets includes two error values, one for the entire curve (”Curve”) and one for the curve up to 12800 seconds (”12800s”). As shown previously, HAL dissipation curves drop off sharply when battery power gets low. As such, this dramatic change in slope (a small change in time is a very large change in millivolts) causes a large error values in our prediction data which is reflected in the average (”Curve” data). The 12800s data indicates the mean absolute prediction error up to this point. In other work, we have developed a technique which allows us to transform the dissipation curve to enable more consistent and accurate prediction across the entire curve [26].

The results in Table 2 indicate that for these programs, we achieve the most accurate prediction when we compose the dissipation rate for the appropriate percentage of register instructions with the dissipation rate of IMem for all other instructions (IReg-IMem). On average, the absolute error is 15 millivolts. We can obtain additional accuracy for floating point programs when the FPReg dissipation rate is used for the percentage of floating point operations in the programs. The average absolute error across floating point programs only is 14 millivolts.

Our predicted rate curves can be used by a dynamic compiler or runtime system to accurately predict the battery dissipation of a program. For example, if the current battery level is 3864mV, an estimate of battery dissipation for a program can be obtained using an appropriate predicted rate curve. We performed this experiment using the IMem-IReg curves for each program. The absolute prediction error (not averaged) for each program (if execution is to begin at 3864mV) is shown in the final column of the table. The observed millivolt drain for each program during this period is included as the last column of Table 1. On average, for a single program execution at this battery level, our prediction error is 0.64mV. We believe that this technique will be useful for many different applications, e.g., to guide optimization and dynamic code generation, migration, quality-of-service, voltage scaling, etc. We plan to investigate the efficacy of our techniques for such services as part of future work.



Integer Benchmarks BitCount (left) and Dijkstra (right)



Floating Point Benchmarks LU (left) and QSort (right)

Figure 2: Predicted and observed battery dissipation curves for representative integer (top row) and floating point (bottom row) programs studied.

Prog.	IReg-IMem Curve	IReg-IMem To 12800s	IReg-IMem_Cache Curve	IReg-IMem_Cache To 12800s	IReg-IMem-IMemW To 12000s	IReg-IMem-FPReg Curve	IReg-IMem-FPReg To 12800s	IReg-IMem mV drain (1 run) start: 3864mV
BasicMath	26.33	13.49	19.21	16.63	14.88	7.27	5.62	1.25
BitCount	10.75	10.60	54.88	40.61	2.31	0.00	0.00	0.60
Dijkstra	11.54	7.30	47.04	37.25	7.15	0.00	0.00	0.44
FFT	19.55	8.15	26.23	22.24	20.32	10.01	10.36	0.66
LU	11.88	2.78	37.45	31.60	29.21	19.40	19.11	1.20
MMult	17.00	7.05	28.52	23.06	6.75	0.00	0.00	0.11
QSort	9.58	3.01	36.70	27.94	24.90	17.61	15.03	0.24
Average	15.23	7.48	35.72	28.48	15.08	13.57	12.53	0.64

Table 2: Prediction error in millivolts for the various prediction techniques. The first seven columns of data are the mean absolute errors for the entire drain curve for each program given predictions of different types. Columns entitled "Curve" is the average error for the entire battery drain curve. Those entitled "12800s" show the average error prior to the battery drop off that commonly occurs at 12800s. The final column is the absolute error (not averaged) due to drain prediction of a single run of the programs. For each prediction, various benchmark curves (IReg, IMem, IMem_Cache, IMemW, and FPReg) were used according to the percentage of instruction categories executed by each program.

6. CONCLUSION

Tools that dynamically control program power consumption on battery-powered devices are essential for the success of next-generation mobile devices and applications. To enable development of such tools we first must fundamentally understand application power consumption and its impact on battery dissipation. The techniques presented herein are an initial step.

Our work investigates the degree to which battery dissipation can be sensed and predicted at the application-level. For each of our techniques, we compare the power dissipation effects of different processor activities on measurable power drain. We show how these benchmark dissipation rates can be combined to form an estimate of battery dissipation for an arbitrary program. By observing the battery life impact by the whole device as a "black-box", our technique does not require a composition of subsystem and battery models. At the same time, we use only measurements that are available via standard operating system interfaces making the methodology practical for implementation in a compilation system using currently available hardware, i.e., without new hardware features for measuring battery consumption. Although we have only tested our methodology on iPAQ, we believe it is general and can be used with other processors and batteries since our benchmark programs can be easily adapted and we are not hardware-specific.

As part of future work, we plan to investigate further the limitations of our approach, e.g., the impact of inter-instruction interaction and cache miss rate estimation. One such technique is to add an offset to the prediction curve according to the isolated energy consumption of these effects in a way that is similar to that used in [24]. We also plan to use dynamic feedback, e.g., about cache miss rate and instructions executed, from the program to improve the accuracy of our estimates. In addition, we will extend our method by including the effects of other subsystems, e.g., storage devices and the display.

7. REFERENCES

[1] Lithium-ion polymer batteries - dlp 305590. `http://www.danionics.com/products/index.asp`.

[2] L. Benini, A. Bogliolo, and G. Micheli. Dynamic power management of electronic systems. In *International Conference on Computer-Aided Design*, 1998.

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

[4] D. Burger and T. Austin. The simplescalar toolset, version 2. Technical Report 1342, University of Wisconsin Madison Computer Science Department, Jun 1997.

[5] Conversion code from hal raw data to percentage battery remaining and voltage: h3600_micro_battery_ack. `linux/2.4.18-rmk3/arch/arm/mach-sa1100/h3600_micro.c`.

[6] Compaq Computer Corporation. Compaq ipaq pocket pc h3700 series, 2002. `http://www.compaq.com/products/quickspecs/10973_na/10973_na.HTML`.

[7] M. Doyle, T. F. Fuller, and J. Newman. Modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal of Electrochem Society*, 141(1):1–9, January 1994.

[8] W. Press et.al. *Numerical Recipes in C*. Cambridge University Press, 1992.

[9] Familiar linux on ipaq. `http://familiar.handhelds.org/`.

[10] S. Gold. A PSPICE macromodel for lithium-ion batteries. *In Proceedings of Annual Battery Conference on Applications and Advances*, pages 215–222, 1997.

[11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th IEEE International Workshop on Workload Characteristics*, pages 3–14, Dec 2001.

[12] Intel corporation. Pentium III processors: Low Power Consumption via SpeedStep.

[13] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proc. IEEE Design, Automation and Test in Europe Conf. (DATE)*, 2001.

[14] A. Krishnaswamy and R. Gupta. Profile guided selection of arm and thumb instructions. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*, Jun 2002.

[15] S. Lee, A. Ermedahl, and S. Min. An accurate instruction-level energy consumption model for embedded risc processors. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, Jun 2001.

[16] R. Maro, Y. Bai, and R. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *PACS*, pages 97–111, 2000.

[17] D. Panigrahi, C. Chiasserini, S. Dey, R. Rao, A. Raghunathan, and K. Lahiri. Battery life estimation of mobile embedded systems. *The 14th IEEE International Conference on VLSI Design*, 2001.

[18] D. Rakhmatov and S. Vrudhula. Time-to-failure estimation for batteries in portable electronic systems. *In Proceedings of the International Symposium on Low Power Electronics and Design*, August 2001.

[19] D. Rakhmatov, S. Vrudhula, and D. A. Wallach. Battery lifetime prediction for energy-aware computing. *In Proceedings of the International Symposium on Low Power Electronics and Design*, August 2002.

[20] P. Rong and M. Pedram. Remaining battery capacity prediction for lithium-ion batteries. *Conference of Design Automation and Test in Europe*, March 2003.

[21] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. Hu, C-H.Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*, Jun 2002.

[22] V. Tiwari, S. Malik, and A. Wolf. Power analysis of embedded software: A first step towards software power minimization. In *IEEE Transactions on VLSI Systems*, Dec 1994.

[23] V. Tiwari, S. Malik, and A. Wolf. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[24] V. Tiwari, S. Malik, and A. Wolf. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[25] Transmeta corporation, crusoe processor. `http://www.transmeta.com/technology/index.html`.

[26] 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'03)*, Dec 2003.