Predicting Program Power Consumption

Chandra Krintz Ye Wen Rich Wolski

University of California, Santa Barbara
Department of Computer Science
{ckrintz,wenye,rich}@cs.ucsb.edu

Abstract

*In this paper, we investigate the degree to which power dissipation induced by program execution can be measured by application-level software tools and predicted by a compiler and runtime system. Application control of the power it uses while executing on a processor is critical to battery-powered mobile devices that are an integral to any realization of ubiquitous computing. Our work investigates the use of dissipation **rates** to make whole-program power-consumption estimates.*

1. INTRODUCTION

The Internet is a vast distributed system that connects a great diversity of electronic devices which over the last decade has come to include everything from desktops to supercomputers, from laptops to handhelds, and from web tablets to web phones. Each type of device differs in compute power, memory size, electricity consumption (and how power is acquired), connectivity, availability, among many other characteristics. A vision of ubiquitous computing is taking shape that will enable "anytime anywhere" access to the Internet's vast offerings and create wholly new applications based on a user's physical location.

While handheld, battery-powered devices are emerging 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 [18, 13, 20, 14, 12, 24] 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 [16, 13, 14, 12, 24]. Other techniques select instructions based on their energy consumption [15, 22, 23, 20].

A key feature of these low-level techniques is that they model the power dissipation associated with a particular subsystem within a computational device. Much work [22, 23, 15, 20, 19, 17] has focused on the power consumed by a CPU when executing a particular instruction. We observe, however, that the power dissipation caused by a program when running on a handheld device will involve many internal subsystems (e.g. CPU, memory, I/O, display, etc.). Further, even if efficient models for all subsystems are available, their independence is not obvious making a comprehensive compositional model potentially complex.

The method we describe in this paper relies on application-level observations of power 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 a power consumption 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 compilation system using currently available hardware (i.e. without new hardware features for measuring power dissipation). For this study, we use the iPAQ handheld device which uses a StrongARM SA-1110 processor [4, 21, 10, 11] — 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 makes two important contributions. First, it identifies the set of relevant set of instruction categories that are necessary to make accurate power dissipation estimates for the iPAQ. Secondly, it demonstrates the way in which benchmark readings for these categories can be *composed* into a dissipation estimate for a target program. We detail the accuracy of these estimates by comparing them to observed dissipation values for a set of target application programs. Our results show that with relatively few instruction categories, accurate estimates can be derived.

Moreover, as mentioned previously, we use dissipation measurements that are obtained via the operating system interface to the battery subsystem and our comparative results are empirical. As such, our results attempt to describe, as directly as possible, the efficacy that would be observed by a compilation system deployed on currently available devices.

2. EXPERIMENTAL 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 a StrongARM SA-1110 [21, 10, 11] — a commercially available, example of a popular, power-sensitive processor technology for mobile computing. Indeed, we note that many of the students at UCSB use iPAQs as productivity "enhancing" tools.

The processor on our iPAQ [4] is a 206MHz StrongARM SA-1110 [21, 10, 11, 5] that uses two on-chip data caches (DCache) and one on-chip 16KB instruction cache (ICache). The first, primary DCache is a standard level 1 (L1) data cache that holds 8KB of data in 256 lines (32 bytes each). It is 32-way set-associative. The second cache is also on-chip and is referred to as the minicache. It is a 512 byte write-back cache with 16 lines (each 32 bytes in length) with 32-way set-associative organization. The minicache is used to reduce thrashing caused by large data structures (objects) in the main cache. The processor has 27 registers; however, only 16 are available to the compiler for user code at any one time.

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 installed *Familiar* Linux [7] 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 multiple 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 [3]. We use millivolts throughout this text (since it is the metric exported) to describe battery level.

2.1 Application-Level Observations and Power Prediction

Our objective with this work is to determine the degree to which program power consumption could be *predicted* using *application-level measurements* of battery drain that could be made available to a compilation system. Previous work [22, 23, 15, 20, 19, 17] has studied the power dissipation characteristics at the processor instruction level. The iPAQ is a complete system consisting of a display, memory, I/O subsystem, etc. Rather than attempting to compose a complete dissipation model from component dissipation models for each subsystem, we chose a "black-box" approach in which we attempt to observe the dissipation characteristics *for the entire iPAQ system* while it is executing a particular type of instruction. Our goal is to combine these system-comprehensive observations into an estimate of power dissipation for arbitrary programs. Similarly, because our ultimate aim is to design power-sensitive compiler optimizations, we wish to examine the efficacy of using measurements that a compiler and/or run time system could access without special-purpose hardware.

2.2 Benchmarks

Our methodology uses a set of observed drain-rate curves from a suite of *benchmarks* to determine the *power dissipation rate* associated with a particular kind of instruction. Then, using the observed dissipation measurements for individual instruction categories, we compose an estimate of overall program dissipation rate for an arbitrary program. For example, if the compilation system determined that a program consists of 70% integer operations and 30% integer loads and stores, our system would compose the dissipation rates from an integer operation benchmark and and integer memory benchmark to make an estimate for the program. However the fraction of each, as we describe more completely in Section 3.2, is not 70–30, but rather is proportional to the time the program spends executing instructions from each category.

Each benchmark

- contains only a single kind of instruction (with the exception of jumps for looping), and
- has been crafted to fit within the ICache of the iPAQ.

For example, to test the power dissipation of integer add instructions, we crafted an assembly language program (that produces no useful output) consisting of only integer add instructions. The length of the program (i.e. the number of integer adds) is long enough to fill the ICache, but not spill out of it since we wished to "wash out" any of the overhead introduced by a jump instruction. That is, the simple add-jump-back loop would consist of 50% adds and 50% jumps. We wished to minimize the effect of the looping jump. In addition, we used constant address offset to minimize register activity in the memory-testing benchmarks, and similarly chose a stride-8 access pattern to ensure each access touched a new cache line.

Initially, we identified four categories of relevant instruction types: integer register operations, integer loads and stores, floating point register operations, and floating point loads and stores. The remainder of this text, refers to these benchmarks as **IReg**, **IMem**, **FPReg**, and **FPMem** respectively. In addition, we wished to 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). The in-cache versions of the IMem and FPMem benchmarks are denoted **IMem_Cache** and **FPMem_Cache** respectively, and the sizes are given in context. We verified that all benchmarks in fact exercised only the CPU and memory subsystems we intended using a StrongARM version of the SimpleScalar simulator [2].

Finally, 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. Documentation and developer reports for the iPAQ vary with respect to the functioning of the cache

system, particularly for cache-write-back. The IMem, FPMem, IMem_Cache, and FPMem_Cache use only load instructions. We developed versions of the integer benchmarks (**IMemW** and **IMemW_Cache**) that implement only store instructions. We did not implement **FPMemW** or **FPMemW_Cache** because on the iPAQ we chose, *all* floating point operations (including loads and stores) were implemented in an operating system trap. As such, we assume that they have equivalent power consumption characteristics. The entire set of benchmarks we have defined for this work are freely available.

To build executables from the benchmarks for the iPAQ, and to compile the test *programs* that we use to verify our results, we used the gcc StrongARM cross-compiler (arm-linux-gcc) on a Debian Linux version 2.4.17 X86-based machine. The cross-compiler (as well as other tools, e.g., objdump, as,ld, etc.) was obtained from [9].

2.3 Benchmark Power Consumption

To measure benchmark power consumption, 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 via a serial connection, converted the HAL raw data value to millivolts, and logged the result on the remote computer. We performed this experiment repeatedly for each of the benchmarks: IReg, FPReg, IMem, FPMem, IMem_Cache, and FPMem_Cache. We varied the array sizes for both the in-cache and out-of-cache benchmarks but 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.

A set of results that is representative of those collected is shown 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 use of a coprocessor: 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

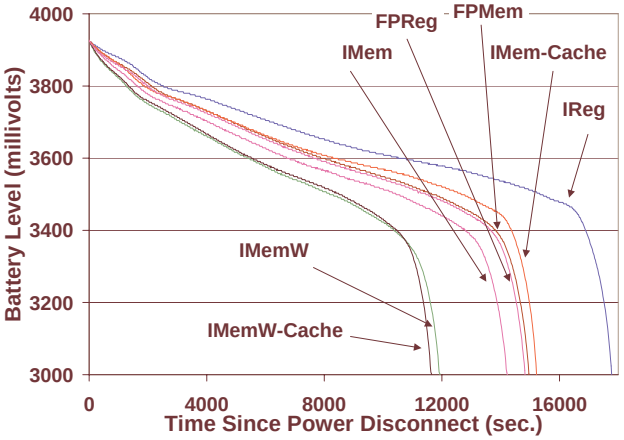

Figure 1: Comparison of battery drain rates for our benchmarks: IReg, FPReg, IMem, and FPMem. The memory benchmarks implement different types of memory accesses: cache-only (8000B) and memory only (32000B).

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 [11] 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 3.2. In summary, the benchmarks consumption rates that we compose to make predictions in this study are IReg, IMem, IMem_Cache, IMemW, and FPReg.

3. COMPOSING BENCHMARK POWER CONSUMPTION RATES

Given the HAL millivolt curves for the selected 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 instructions in each of aforementioned categories (integer register operations, floating point operations, integer loads, and integer stores).

Notice that the HAL millivolt curves in Figure 1 are not linear, i.e., consumption rates for the same execution that occurs at different points in battery life. As such, a dynamic compilation or runtime system will require a predicted *rate curve* from which it can extract a prediction of program consumption given the current battery level.

BasicMath	MiBenchmark: Automotive/Industrial Mathematical calculations commonly unsupported in embedded processors
BitCount	MiBenchmark: Automotive/Industrial Bit manipulations
Dijkstra	MiBenchmark: Network Dijkstra's shortest path algorithm performed on an adjacency matrix
FFT	MiBenchmark: Telecommunications Fast Fourier Transform
LUD	Numerical recipe [6] L/U decomposition of a matrix
MMult	Integer matrix multiply using a 2D array of X integers
QSort	MiBenchmark: Automotive/Industrial Quicksort on an array in memory

Table 1: Description of the benchmarks used in this study.

Alternately, we could simply profile HAL consumption rate curves for the program we are interested in. However in some cases, such a utility may not be available to us. In addition, program execution characteristics can be obtained by alternate means, e.g., simulation, estimation, etc. Many such systems do not have accurate battery consumption measurement available to them. As such, we investigate alternate means of battery consumption prediction that is based on composition of dissipation rate curves from benchmarks. We believe that this work also opens doors to new research on mechanisms that estimate program execution time and the accuracy with which such estimates impact power prediction. We plan to investigate such questions as part of future work.

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

3.1 Empirical Evaluation

To verify (or disprove) this thesis (that battery consumption of individual instructions composes for arbitrary combinations of register and memory instructions), we predicted and observed the consumption rates of seven C programs from the MiBench embedded program suite [8] and from hand-coded implementations of other well known algorithms from [6]. A description and various statistics on the programs we report results for is shown in Table 3 and Table 3.1, respectively. 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%.

Table 3.1 contains 7 columns of data. The first column is the execution time of the program in seconds. The second column is the dynamic instruction count (IC) in millions of instructions. The third through sixth column shows the percentage of these dynamic counts that constitute each of the four instruction categories: integer register operations (IReg), integer load (IMem), integer stores (IMemW), and floating point operations (FPReg). As mentioned above, we include floating point loads and stores in the latter. 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.

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

3.2 Consumption Rate Prediction Results

To predict the power consumption rate for an arbitrary program we composed the consumption rates obtained from our benchmarks profiles. Note that we are *not* computing the consumption of individual instructions from the benchmark measurements. Since the dissipation curve changes over time, so does instruction-level consumption. As such, our goal is to construct a complete rate curve for the target application. Our compilation system can then, given the current battery level, obtain an estimate of the power consumption for the program.

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. However, the computation is complicated since a program that executes 70% register instructions does not spend 70% of the execution *time* performing register operations. As such, we must first compute the amount of *time* required to perform these register operations using the IReg benchmark characteristics. Likewise we compute the *time* required to execute the remaining instructions using the IMem benchmark characteristics. We then use these times to find the corresponding dissipation in each curve (IReg and IMem).

This methodology assumes that we have battery consumption samples 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 commonly terminate earlier than do the observed curves.

Prog.	ET (secs)	DynIC (*1Mil)	IReg (pct)	IMem (pct)	IMemW (pct)	FPReg (pct)	mV drain start: 3864mV
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 2: Execution statistics for the programs described in Table 3. Column 1 is the execution time of the program in seconds. Column 2 is the dynamic instruction count (IC) in millions of instructions. The third through sixth column shows the percentage of these dynamic counts that constitute each of the four instruction categories: integer register operations (IReg), integer load (IMem), integer stores (IMemW), and floating point operations (FPReg). The last column shows the millivolt battery drain for a single run of each program when invoked with a battery level of 3864mV (almost fully charged).

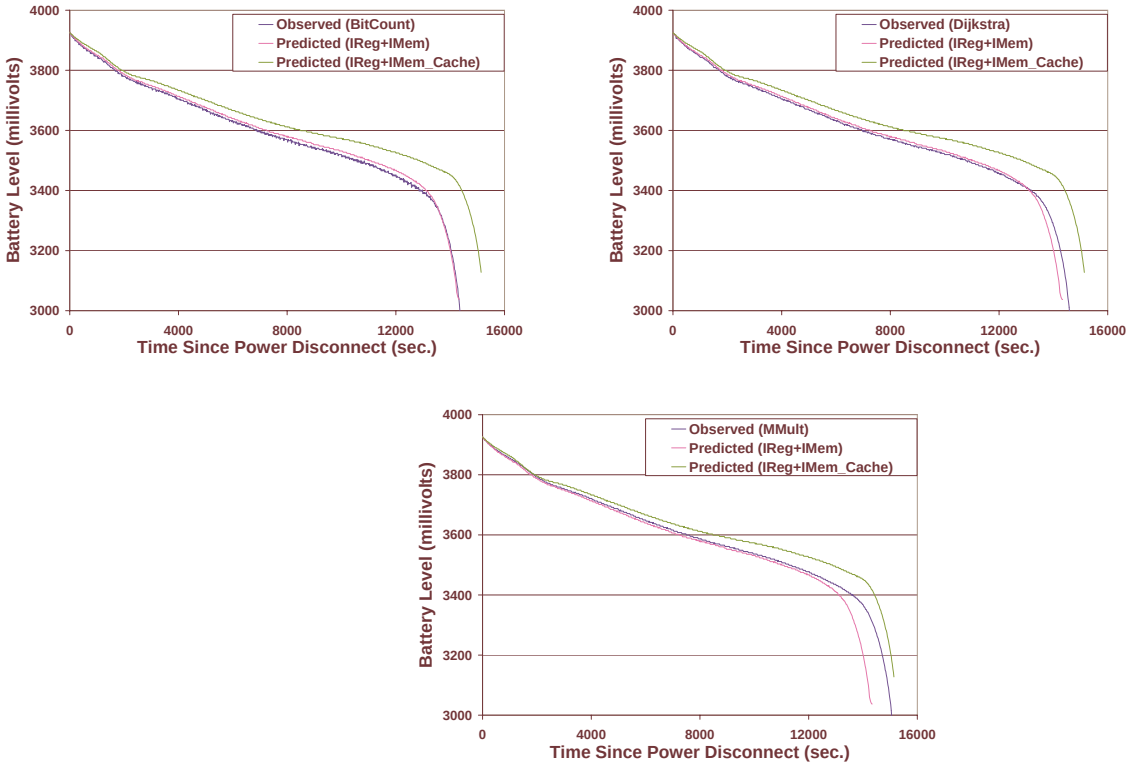

Figure 2: Predicted and observed battery consumption curves for the integer programs studied.

We first consider the integer programs (BitCount, MMult, Dijkstra). These programs implement two types of instructions: register operations and memory operations. As such we compose the dissipation rate curves of the IReg and the IMem benchmark. With the use of the latter, 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 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 measured each program (modified to loop infinitely) as we did for the benchmarks to obtain an *observed* consumption curve to which we can compare our predicted curves. The integer program results are shown in Figure 2. A graph is shown for each program (identified by the Observed legend entry). 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.

Each graph contains two curves, one predicted using IReg and IMem (IReg-IMem) and one predicted using IReg and IMem_Cache (IReg-IMem_Cache). These curves provide a lower and upper bound on prediction error. We report error rates of the predicted curves at the end of this section. However, 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 consumption rates of constituent instruction types can be composed to *accurately* predict program consumption rates**.

We next investigate the efficacy of this approach for floating point programs (BasicMath, FFT, LU, QSort). The result graphs are shown in Figure 3. As in the previous figure, the x-axis is time (in seconds) since the battery was disconnected and the y-axis is the battery life in millivolts as reported by HAL.

For these programs, we computed the predicted curves using three compositions of benchmarks: IReg and IMem only (IReg-IMem), IReg and IMem_Cache only (IReg-IMem_Cache), and IReg, IMem, and FPReg only (IReg-IMem-FPReg). Each of the resulting prediction curves is shown in the graphs for each program; the observed curves are also included.

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 consumption prediction. In addition, doing so results in an accurate prediction of floating point program battery consumption.

Table 3.2 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 hold the mean absolute error of the predictions over 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 benchmark curve to compute the drain that results from 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 in 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"). HAL consumption 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.

These results indicate that for these programs, we achieve the most accurate prediction when we compose the consumption rate for the appropriate percentage of register instructions with the consumption rate of IMem for all other instructions (IReg-IMem). On average, the absolute error is 15 millivolts. Additional accuracy can be obtained for floating point programs when the FPReg consumption 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 to accurately predict program battery consumption in an embedded system dynamic compiler and runtime system. For example, if the current battery level is 3864mV, an estimate of program consumption 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 3.1. On average, for a single program execution at this battery level, we achieve an average error of 0.64mV. Power consumption prediction can be used to guide optimization and dynamic code generation, migration, quality-of-service, and voltage scaling among other techniques. We plan to investigate such services as part of future work.

4. CONCLUSION

While handheld, battery-powered devices such as personal digital assistants (PDA's) and web-enabled mobile phones are emerging 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. Worse yet, the cost of such devices might even widen the "digital divide" rather than extending the reach of the Internet not just to anywhere and at any time but also to **everyone**.

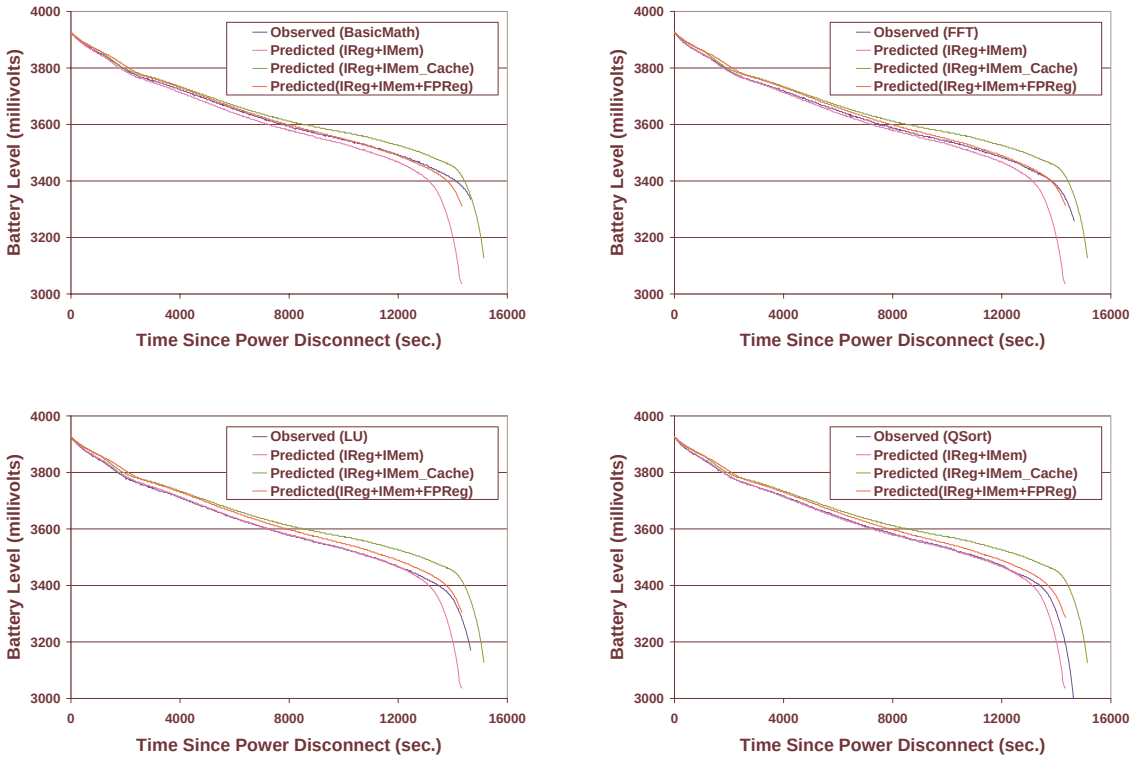

Figure 3: Predicted and observed battery consumption curves for the floating point programs studied.

	IReg-IMem		IReg-IMem_Cache		IReg-IMem-IMemW	IReg-IMem-FPReg		IReg-IMem mV drain (1 run)
Prog.	Curve	To 12800s	Curve	To 12800s	To 12000s	Curve	To 12800s	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 3: 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.

Tools that dynamically control application power consumption are essential. To enable development of such tools we first must fundamentally understand application power consumption. The techniques presented herein are an initial step.

Our work investigates the degree to which power 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 a power consumption 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 compilation system using currently available hardware (i.e. without new hardware features for measuring power dissipation).

5. REFERENCES

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

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

[3] 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`.

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

[5] I. Corporation. Intel strongarm sa-1110 microprocessor brief datasheet, 2002. `ftp://download.intel.com/design /strong/datashts/278241.htm`.

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

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

[8] 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.

[9] Handhelds.org. StrongARM toolchain `ftp://ftp.handhelds.org/pub/linux /arm/toolchain/`.

[10] J. Hicks and J. Gettys. *Compaq iPAQ H3600 Hardware Design Specification - Version 0.2f*. Compaq Computer Corporation, 2000. `http://www.handhelds.org/Compaq /iPAQH3600/iPAQ_H3600.html`.

[11] Intel. Instruction set; chapter 4.

[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] J.Heeb. The next generation of strongarm. In *Embedded Processor Forum*, May 1999.

[15] 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.

[16] L.Benini, A. Bogliolo, and G. Micheli. Dynamic power management of electronic systems. In *International Conference on Computer-Aided Design*, pages 696–702, 1998.

[17] 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.

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

[19] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *International Conference on Computer Design (ICCD '98)*, Oct 1998.

[20] 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.

[21] D. Seal, editor. *The ARM Instruction Set; Chapter A3*. Addison-Wesley, 2000.

[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 sofware. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[24] Transmeta corporation. Crusoe Processor `http://www.transmeta.com/technology/index.html`.