A Hardware-Software Cooperative Approach for Application Energy Profiling

Jie Chen, Guru Venkataramani
The George Washington University, Washington, DC

Abstract—Energy consumption by software applications is a critical issue that determines the future of multicore software development. In this article, we propose a hardware-software cooperative approach that uses hardware support to efficiently gather the energy-related hardware counters during program execution, and utilizes parameter estimation models in software to compute the energy consumption by instructions at a finer grain level (say basic block). We design mechanisms to minimize collinearity in profiler data, and present results to validate our energy estimation methodology.

Index Terms—Energy Profiling, Energy Estimation, Energy Debugging

1 Introduction

Innovations in computer architecture and semiconductor technologies have increased the computational performance of systems exponentially. At the same time, energy costs incurred by software have grown rapidly resulting in the need to educate programmers on application energy consumption.

Conventionally, execution time (performance) of applications is a commonly adopted proxy measure for software developers to identify energy bottlenecks in their program code. Recent studies by Hao et al [2] have shown that the execution time and energy consumption do not have strong correlation because of several factors such as (1) multiple power states- at two different frequencies f_1 and f_2, even if the execution times are the same, the energy drawn will be different, (2) asynchronous design of system and API calls- when the application sends data over the network, the data is handled by the OS which results in the corresponding data-sending application not being charged for the data transmission time. Also, application energy profile and optimizations are often specific to the processor architecture and hardware configurations. Such factors result in the need to invest into energy profiling.

In this article, we present a hardware software cooperative approach to understand energy consumption by applications, and relate them back to the application code. In doing so, we enable the participation of programmers and software tools (such as compilers and runtime) in energy-aware software development without having to rely on expensive runtime energy saving strategies.

The contributions of this article are as follows:

- We motivate the need for energy profiling in applications, and propose a hardware-software cooperative framework to analyze application energy consumption.
- We explore fine-grain energy estimation methodologies, and study mechanisms to minimize the effects of collinearity in data gathering.
- We evaluate our proposed mechanisms using Splash-2 [14] and PARSEC-1.0 [1] benchmarks, and validate the effectiveness of our solutions.

Manuscript submitted: 23-Dec-2013. Manuscript accepted: 18-Apr-2014. Final manuscript received: 23-Apr-2014.

2 Motivation

With increasingly complex interactions between instruction execution and the associated timing in functional units (due to parallelism and pipelining), it is often difficult to equate performance with energy. To illustrate this effect, we conduct experiments on several real-world applications from Splash-2 [14] and PARSEC-1.0 [1] benchmark suites, where the energy consumption characteristics of individual functions drastically differ from their corresponding execution time profiles.

Table 2 shows the energy and execution time profiles on several applications with 8 threads running on 8 cores. All of our experiments were done using SESC [11], a cycle-accurate, multi-core architecture simulator that is integrated with McPAT power model [8]. Table 2 shows the processor configuration details input to the McPAT power model.

1) **Ocean**: relax() consumes 30.31% of the total energy but only accounts for 15.02% of the total execution time. On the other hand, slave2() accounts for 30.30% of execution time, but only consumes 18.98% energy. Upon further examination, we observed *highly overlapped execution* of double-word arithmetic instructions in relax() led to higher energy with lower execution time. However, slave2() had higher numbers of branch and load/store instructions leading to longer execution time despite consuming lower energy than relax().

2) **Radiosity**: v_intersect() consumes 11.06% of total energy with only 6.60% of the total execution time,

TABLE 1
Energy and performance profile of functions in Splash-2 and PARSEC-1.0 benchmarks with 8 threads

Ocean			Radiosity			Bodytrack		
Function	% of Energy	% of Time	Function	% of Energy	% of Time	Function	% of Energy	% of Time
relax	30.31%	15.02%	v_intersect	11.06%	6.60%	InsideError	25.38%	9.12%
slave2	18.98%	30.30%	compute_diff_ disc_formfactor	7.39%	14.07%	Exec	19.20%	4.34%
jacobcal2	14.47%	12.22%	traverse_bsp	4.83%	5.53%	EdgeError	18.53%	6.77%
laplacalc	12.68%	9.85%	four_center_points	3.03%	5.06%	ImageProjection	10.66%	3.67%

TABLE 2
Processor Configuration and Power Model

Processor	3 GHz, 8-core CMP; 4-wide issue/retire, out-of-order execution; 4096-entry BTB, hybrid branch Predictor; 8-entry instruction queue; 176-entry ROB; 96 integer registers; 90 floating point registers; 64-entry LD/ST queues; 48-entry scheduler
Memory Sub-system	32KB, 4-way, I-cache; 32KB, 4-way, D-cache; 256KB, 8-way, private L2 cache; 16MB, 16-way, shared L3 cache; 64-entry ITLB/DTLB
Interconnect	shared bus below private L2 caches
Power Model	McPAT, 32 nm, V_{dd} = 1.25 V

while compute_diff_disc_formfactor() has 14.07% of the total execution time with only 7.39% of the total energy. On a closer review, we found that v_intersect() heavily used complex instructions like madd.d (that perform multiply-add of double word values) leading to higher energy, while compute_diff_disc_formfactor() had a lot of load operations leading to higher execution time despite consuming lower energy than v_intersect().

3) **Bodytrack**: The top four energy consuming functions account for 74% of the total energy, but only account for about 24% of the total execution time. About 64% execution time is actually spent on lock and barrier synchronizations implemented by pthread_cond_wait() that actually puts threads into sleep without consuming much energy.

2.1 Are Current Hardware Energy Meters Sufficient?

Modern high performance processor architectures [12], [5] have begun integrating hardware energy meters that can be read through software driver interfaces. For example, starting from Sandy Bridge, Intel provides a driver interface called RAPL (Running Average Power Limit) that can let programmers periodically sample processor energy usually at the granularity of a few milliseconds of program execution time. While this is going to be a useful first step toward helping programmers understand processor energy consumption, it is still far from providing them with a more practical feedback at a granularity that relates the processor energy consumption back to the program source code.

3 Fine-Grained Energy Profiling

To help programmers, compilers or runtime optimizers apply effective energy optimizations to the right code regions, energy profile information must be given at the level of fine-grained code blocks (say a few basic blocks). Toward this goal, we explore the use of hardware that can efficiently estimate energy using hardware counters, and apply light-weight software estimation techniques to uncover the energy share at a finer granularity, e.g., a few selected instructions.

3.1 Code Sequences

To attribute energy back to program source code, we choose to identify *code sequences* in applications. A code sequence is a series of basic blocks during dynamic execution that are formed as a result of executing a few static basic blocks one or more times. For example, if a loop comprises the execution of two static basic blocks b_1 and b_2 with execution counts of n_1 and n_2, respectively, its code sequence is recorded as $< b_1, n_1, b_2, n_2 >$. These code sequences can be identified dynamically by maintaining a hardware buffer to record the sequence of dynamically executing basic blocks, and recording the execution frequency of these basic blocks. The energy consumption of the corresponding code sequence, E, is obtained using the hardware energy meters. Subsequently, for every code sequence S, we gather a tuple $< E, b_1, n_1, b_2, n_2, ..., b_k, n_k >$, where n_i is the execution count for the constituent basic block b_i, and k is the total number of constituent basic blocks. The hardware constructs *identifier* partial tags using the basic blocks starting address (i.e., target address of the branch instruction in the previous dynamic basic block) and the ending address (i.e., branch instruction at the end of the current basic block). Once a code sequence finishes execution, we read the hardware energy meter to record the energy value E for that code sequence. In our experiments, we observed that even the most expensive energy consuming loops did not have more than 15 basic blocks. Hence, an on-chip hardware histogram buffer to temporarily hold 15 tags and 16 counters (15 for basic block counts and 1 for energy) is sufficient for every hardware core. Since the histogram buffer is off the critical path, and has access latency of 0.18 ns using Cacti 5.1 [6] (less than one CPU clock cycle), it is unlikely to adversely impact the processor performance. We note that code sequences with longer chains of static basic blocks might need to be chopped into multiple shorter code sequences such that the hardware can record energy information about the constituent basic blocks. Software support will be needed to identify and instrument such longer code sequences.

3.2 Fine-Grained Energy Estimation Methodology

The energy spent by a code sequence is essentially an additive function of all of its constituent basic blocks' energy consumption. We show this additive function in Equation 1, where E is the measured code sequence energy; e_i is the unknown *unit basic block energy*, that is, basic block b_i's average energy consumption; n_i is the known execution count of basic block b_i; k is the total number of constituent basic blocks. Note that Equation 1 can be vectorized as $E = \vec{n} \cdot \vec{e}$.

$$E = \sum_{i}^{k} (n_i \times e_i) \tag{1}$$

The next step is to find the estimate of the unknown unit basic block energy. Among several different parameter estimation methods, Least Square Estimation (LSE) and Maximum Likelihood Estimation (MLE) [9] are two popular techniques. LSE can make very efficient use of the data, and typically desirable estimated parameters can be obtained with relatively smaller numbers of sample data. Mathematically, MLE can also yield good estimates, but it has two major disadvantages compared to the LSE: 1) estimating unknown parameters in MLE often requires solving complex non-linear equations [9]; 2) MLE assumes the underlying data to follow a specific probability distribution model. Therefore, we chose to use LSE as our method to estimate the unit basic block energy.

$$C = \sum_{i=1}^{n} [E_i - \vec{n}_i \cdot \vec{e}]^2 \tag{2}$$

In our design, LSE performs parameter estimation by minimizing a cost function C using Ordinary Least Squares (OLS) method [9]. Specifically, the unknown parameter $\vec{e}$ (that denotes unit basic block energy) is estimated by finding the optimal parameter value vector $\hat{\vec{e}}$ that minimizes the cost C, where C is the sum of the squared distance between E and $\vec{n}_i \cdot \vec{e}$ over n samples (Equation 2). However, one of the assumptions that OLS makes is that basic block execution has deterministic energy consumption. In other words, the execution of the same sequence of basic blocks should always consume similar amounts of energy. This assumption, however, may not hold all the time due to variability in architectural events during program execution (e.g., cache hits and misses for the same load/store instruction at different times).

$$C = \sum_{i=1}^{n} w_i [E_i - \vec{n}_i \cdot \vec{e}]^2 \tag{3}$$

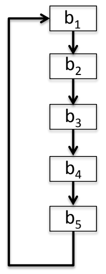
When such variable measurements exist in our data, Weighted Least Squares (WLS) [9] is better suited to estimate unit basic block energy. In WLS's cost function (Equation 3), we multiply every squared distance, $[E_i - \vec{b}_i \cdot \hat{\vec{e}}]^2$, with a non-negative weight, w_i, that reflects how much contribution that each sample would have on the final parameter estimation. These w_i's are heuristically derived by assigning higher weights to samples with lower variances.

We use OLS for code sequences with lower variability due to architectural events, and use WLS for code sequences with higher variability.

3.3 Energy Profile Gathering: The Problem of Data Collinearity

In statistics, data collinearity refers to the situation where one of the columns (predictors) of the sample data matrix is highly correlated with other columns. Often seen in time-series or region data, collinearity might be caused by one predictor variable being the exact duplicate of another, or by one predictor being equal to a linear combination of other predictors. When using such data as the training set, it will result in erroneous estimation of the coefficients.

To illustrate the data collinearity in energy profiling, we show an example in Figure 1 (a). In this case, consecutive basic blocks (b_1 through b_5) get executed repeatedly resulting in all of b_i's having the same execution count. This affects our ability to estimate the individual energy contribution of $b_i's$, effectively leading to them being treated as a single unit (like extended basic block) for energy estimation purposes.



(a)

CS #	N	Start*	# b_1	# b_2	# b_3	# b_4	# b_5	Energy
1	149	b_3	30	29	30	30	30	E_1
2	103	b_2	20	21	21	21	20	E_2
3	181	b_4	36	36	36	37	36	E_3
4	257	b_5	52	51	51	51	52	E_4
5	227	b_4	45	45	45	46	46	E_5

* Denotes the first b_i appearing in the code sequence sample

(b)

Fig. 1. Illustration of Collinearity-aware Energy Profiling

3.4 Collinearity-aware Energy Profiling

To mitigate the problem of data collinearity and be able to estimate basic block energy at a fine grain level, we design a collinearity-aware energy profiling approach. We intentionally create code sequences with differing numbers of basic blocks in each sample such that we are able to form at least k unique samples as inputs for our parameter estimation model. Specifically, the hardware randomly pick a prime number N from a pre-populated vector of prime numbers, and use this randomly chosen N as the *dynamic* basic block count within a code sequence sample. This process is repeated at least k times to generate enough unique samples for our parameter estimation. For each sample, a random starting point (basic block b_i) is picked and the code sequence is terminated after N dynamic basic blocks. Figure 1 (b) shows an example of our collinearity-aware profiling approach where code sequence samples have differing (prime) numbers of dynamic basic block counts to break the data collinearity among them.

3.5 Validation of Energy Estimation

Due to highly overlapped execution of basic blocks, it is impossible to accurately measure the energy consumption of individual basic blocks. Hence for validation, we compute the relative differences between $\sum_{i=1}^{k}(n_i \times \hat{e_i})$, the sum of estimated basic block energy within the code sequence, and E being the actual measured code sequence energy. In our experiment, we adopt Ten-fold Cross-validation [7] method where 90% of the code sequence samples are used as the training set and the remaining 10% of them are used for validation. This step is repeated ten times where a different validation set is selected during each time. We calculate the average cross-validation errors on the code sequences that account for 90% of the total application energy consumption.

$$Error = |(\sum_{i}^{k}(n_i \times \hat{e_i}) - E)/E| \tag{4}$$

Figure 2 shows the average cross-validation errors of our energy estimation model with collinearity-aware energy profiling. Note that, in our experiments, we use a prime number vector that holds prime numbers less than 500. The average cross-validation error across all applications is less than 1%, and the maximum error is less than 2.5% (observed in raytrace).

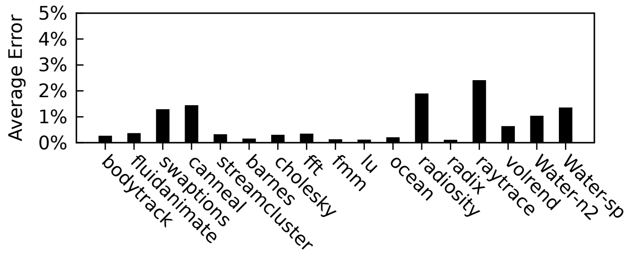


Fig. 2. Cross-validation errors in Splash-2 and PARSEC-1.0 benchmarks

4 Related Work

Isci et al [3] propose runtime power monitoring techniques for processor core and functional units. Powell et al [10] and Jacobson et al [4] show the feasibility of using a limited set of of metrics to estimate functional unit power. In contrast to these prior schemes, we investigate ways to provide fine-grain energy feedback to help programmers understand their application's energy characteristics.

Prior works [13] that estimate energy using instruction counts assume a pre-determined cost for various instruction types, and ignore dynamic hardware effects such as parallelism and interference that occur in most modern architectures. Alternative strategies that use a specific set of hardware events (such as cache misses) for energy estimation often fail to include a comprehensive view of application execution and ignore system-level effects and interactions with other instructions (e.g., pipeline stalls, pipeline flushes due to mispredicted instructions). In contrast to these prior approaches, our methodology directly queries the hardware to accurately gather total energy consumed by a segment of program code, and utilizes parameter estimation techniques to attribute energy to fine grain sets of instructions.

5 Conclusions and Future Work

In this article, we showed the necessity to gather fine-grained energy information about program code. We explored a hardware-software cooperative solution to attribute energy back to the program source code, and observed very low estimation errors (less than 2.5%) when tested on Splash-2 and PARSEC-1.0 benchmarks. As future work, we will study ways to extend our framework by incorporating the energy spent on GPU, main memory, and I/O.

Acknowledgments

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557.

References

[1] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Princeton University Technical Report TR-811-08*, January 2008.
[2] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of ICSE*, 2013.
[3] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of MICRO*, 2006.
[4] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer. Abstraction and microarchitecture scaling in early-stage power modeling. In *Proceedings of HPCA*, 2011.
[5] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, and S. Naffziger. An x86-64 core implemented in 32nm soi cmos. In *Proceedings of ISSCC*, 2010.
[6] N. P. Jouppi et al. Cacti 5.1. *http://quid.hpl.hp.com:9081/cacti/*, 2008.
[7] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, 1995.
[8] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.
[9] NIST. *http://www.itl.nist.gov/div898/handbook/pmd/pmd.htm*, 2013.
[10] M. D. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, and S. Yardi. Camp: A technique to estimate per-structure power at run-time using a few simple parameters. In *Proceedings of HPCA*, 2009.
[11] J. Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.
[12] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro, IEEE*, 2012.
[13] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee. Instruction level power analysis and optimization of software. *J. VLSI Signal Process. Syst.*, 13(2-3), Aug. 1996.
[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of ISCA*, 1995.