Safari: Function-level Power Analysis using Automatic Instrumentation

Shinan Wang, Youhuizi Li, Weisong Shi
Department of Computer Science
Wayne State University
Detroit, Michigan
Email: shinan, huizi and weisong@wayne.edu

Lingjun Fan
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
Email: lingjun@ict.com.cn

Abhishek Agrawal
Intel Corp.
Email: abhishek.r.agrawal@intel.com

***Abstract*—Resolving excessive power dissipation of modern computer systems has become a substantial challenge. However, few research projects have targeted on application power analysis or application-aware power management, which becomes a rising factor in energy efficient system design. In this paper, we describe and implement an application function (subroutine call) level profiler, Safari. It can be used to generate power profiles of each function in an automatic manner. The experiment results using NPB parallel benchmark suite show that Safari is able to collect function level run-time information with overhead (16% on average) comparable to gprof. The power profiling results can be used for code optimization, power-aware scheduling, or even computing resource billing for future research.**

I. Introduction

Among all the factors that contribute to system operational costs, power dissipation and energy consumption are fundamental in modern computer systems [1]. In addition, power and energy affect technical evolvement, which includes data center designs [2], enterprise level server designs, battery life time on a smart phone [3], and circuit layouts on a microprocessor [4].

Different from hardware and system design and analysis, the impact of software on the power dissipation of a computer system has been overlooked. In fact, as the user of hardware resources, software has equivalent or even more effects on the power dissipation of a whole system. For example, Pathak *et al.* introduce a new type of bugs, energy bugs or ebug [3], on smartphones. Their results show that 35% of energy bugs stem from software, either OS or applications. Nevertheless, narrowing down the root causes of ebugs to a software component is one of the crucial steps to fix energy bugs. A recent study shows that software bloats introduce excessive resource usage in large software systems [5] as well. Better understanding of software behavior associated with resource usage is crucial to detect similar scenarios. In addition, workload phases provide interesting information for performance optimization and they are usually related to functions/methods [6]. Function level power profiling is supposed to reveal power behavior along with resource usage information. These information would help developers to understand and leverage power dissipation in a computer system from a new angle. However, information scarcity of dynamic power dissipation impedes developers' ability to produce more energy-efficient software. Even though the latest processors based on Sandy Bridge or Ivy Bridge Architecture provide power information from hardware counters, the power dissipation within each program block is still unclear. In addition, power model based approaches are still effective to estimate the power dissipation of other components, such as memories.

Regardless of the potential influence of software on power dissipation, its impact is usually underestimated. For example, most profiling tools are used to measure performance rather than power or energy. In addition, among the few available tools that estimate power dissipation, most of them do not consider the control flow of a program, which loses the insight of the execution of a program. Given such scarce information, it would be difficult for developers to evaluate or optimize the power usage of their programs.

Run-time profiling techniques usually collect information from profiling systems. Typically, in order to obtain more detailed results, a profiling process generates overhead. Consequently, a profiling process could disturb power measurement. In addition, the inaccuracy due to overhead can be exaggerated because the collected data for power profiling usually need to be processed by power models. For example, Linear Regression is a commonly used technique to generate estimated power from collected run-time information [7]. Moreover, the sizes of applications are growing rapidly, which posts more challenges to analyze software power behavior.

In this paper, we propose a software function level power profiling tool, Safari. The goal of Safari is to provide function level power analysis while minimizing profiling overhead. In order to use Safari, first, we compile a target application to insert instrumentation code for each function. Then, run-time information is collected for the resource usage during the execution of functions. Finally, we apply an off-line analysis based on a selected power model to generate function power profiles. As a result, code "hot region" can be spotted for further power optimization without hardware instrumentation.

This work is in part supported by US NSF grants CCF-0643521, CNS-1205338, and the Introduction of Innovative R&D team program of Guangdong Province $NO.201001D0104726115$, and the Office of Vice President Research of Wayne State University.

This work is done while Lingjun Fan stayed at Wayne State University as a visiting student.

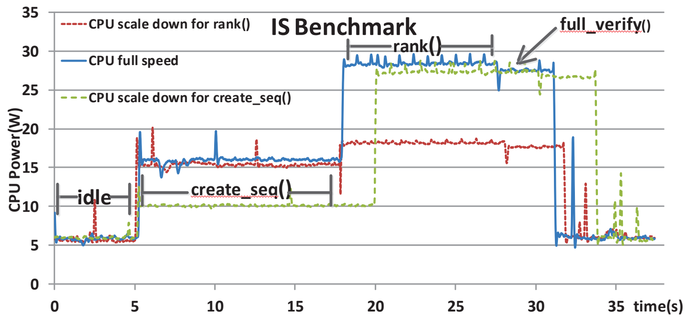

Fig. 1. Power dissipation of IS.A on a Intel Core2 Quad 8200 processor.

The rest of the paper is organized as follows. We start the paper by presenting a motivating example in Section II. In Section III, the design considerations of Safari are described, followed by the evaluation results shown in Section IV. Related work is discussed in Section V. At last, we describe future work and conclude the paper in Section VI.

II. Motivating Examples

Usually different functions in a program have distinct power behavior. For example, we retrieve the function profile of IS.A benchmark from NPB 3.0 benchmark suite. There are three major steps in IS.A: `create_seq()`, `rank()`, and `full_verify()`. The power dissipation of IS.A is closely related to the three major functions as shown in Figure 1. We use two 0.005 Ohm current sense resistors(CSR) series connected to the 12V cable from a standard ATX2.0 power supply. The CPU current is measured by reading the voltage on the resistors using NiDAQ 9205 unit and dividing the resistor value. We can calculate CPU power dissipation using the measured current and voltage value.

Given distinct power dissipation information along with application execution, one of the usage of power profile is to guide run-time power management. In this example, `rank()` function, which produces approximate 0.15 Instructions Per Cycle (IPC), is less CPU-bound during its execution (IPC values are used broadly as CPU power model input [8]). There are, therefore, chances that systems can provide more fine-grained power management or scheduling schemes if resource usage information can be retrieved beforehand. For example, we are able to use DVFS to scale down CPU frequency from 2.34GHz to 2.00GHz during the execution of `rank()`. As a result, we achieved 24% energy saving with 3% performance loss. On the contrary, 10% energy saving is achieved with 10% performance loss if we scale down CPU frequency during the execution of `create_seq()`, which has a higher average IPC value during its execution (around 0.6). In addition, we observe multiplication operations are intensively used in the source code of the`create_seq()` function, while the `rank()` function mainly contains branch-prediction and data movement operations. Hence, it is possible to utilize profiling results to guide system power management in a fine-grained fashion.

Based on this example, we observe that software characteristic is an indispensable part to analyze the power dissipation of a computer system. Safari attempts to accurately estimate power dissipation of a function. Our rationale of using function level profiling includes the following aspects: first, application subroutine/functions are the basic units of executing some tasks; second, it guarantees appropriate scale for optimization: coarser than the instruction level yet finer than the process or thread level; third, as the module design is a common method to develop large scale software, function level power profiling fits this pattern well.

III. Method

Given the fact that power dissipation of function invocation is one of the major break points to understand software dynamic power dissipation, it is worth developing a profiling mechanism to generate function level power profile. The goal of Safari is *automatic power profiling based on per function resource usage*.

In order to achieve this goal, there are three major points need to be considered. First, profiling overhead must be minimized. Although functions can be treated as the basic unit to generate a profile, a majority of functions only accomplish tiny tasks, such as printing timestamps or reversing a string, which hardly present any potential for optimization or tuning in most cases. However, profiling them consumes as much system resources as profiling major functions. For example, a `create_seq()` function invokes `randlc()` millions of times in the IS benchmark. As a result, instrumenting and profiling `randlc()` function produces much more overhead than profiling `create_seq()` along. Moreover, function power behavior varies according to different input data and execution paths. It is important for function level power profiling to reflect those characteristics. Additionally, the core part of power profiling is power models [9], which usually utilize system information and Performance Monitoring Counters (PMCs) as input. In this case, the collected information has to be associated to each function in an application.

A. Overview

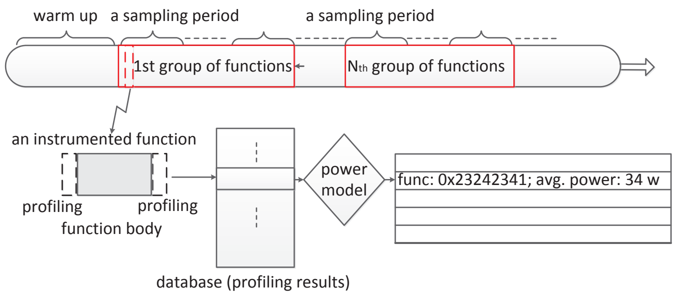

Fig. 2. An overview of the profiling process.

The proposed profiling procedure is demonstrated in Figure 2. First of all, the execution of an application is divided into different parts. During the warm-up period, no profiling data are collected since usually only start-up activities are executed during this period. The rest of execution is divided into different sampling periods. During a sampling period, only a certain number of selected functions are profiled. As a results, functions are randomly grouped into several categories. Only one instance of a function is profiled even the same function can be executed more than once during the same sampling period. Target function groups are switched as time elapsed. Safari collects data exactly before and after a function being executed. Off-line analysis generates power profile based on a predefined power model. We will discuss the details of the profiling procedure in the following sections.

B. Function Level Power Profiling

Instrumentation Automation To instrument an application, there are two commonly used methods. One is to design a set of APIs that controls the procedure of data collection at run-time [8], [10]. The other approach is automatic instrumentation using some available compilation tools, such as PIN [11]. In a considerable large program, there can be more than ten thousands function prototypes. The goal of function level power profiling is to locate the relative power hungry part of source code. Each function block can be a candidate if we treat the whole program as a black box, which means all of them need to be considered. It takes excessive human efforts if we instrument each function manually. As a result, we adopt the second approach because of its simplicity to developers. Whereas, automatic instrumentation has its disadvantage such as it dose not distinguish major functions and trivial ones. If we simply apply this technique, the profiling process could cause unnecessary overhead. Safari adopts several techniques to overcome this effect.

In the implementation of Safari, we use a function instrumentation utility designed for GCC compiler, `-finstrument-functions`, to insert two profiling functions that will be invoked at every entry and exit of each function, namely, `__cyg_profile_func_enter()` and `__cyg_profile_func_exit()` as illustrated in Figure 2. At run-time, in addition to execute instructions in a normal function body, two profiling functions are attached to the both ends of a function to collect *per thread resource usage information*.

Warm up Usually a system is not stable during the warm-up phase of an application. For instance, buffers need to be initialized. In order to get accurate profiling results, the data collection for profiling starts after a warm up period as shown in Figure 2. The total length of warm up depends on a specific program and is tunable at start-up.

Overhead Reduction As described in the previous paragraphs, in order to use automatic instrumentation properly, the key issue is reducing instrumentation overhead. There are two major concerns associated with function level power profiling. First, a specific function can be invoked many times during a relative short duration. This scenario affects not only power profiling of the function itself but also other threads. For instance, context switch or other system activities might rise. Second, nested function calls will add more inaccuracy to outsider ones. Automatic instrumentation in Safari is based on insertion of two additional function calls at the entry and exit of each function. If a function has many nested functions calls, the collected model input can be misleading (dominated by the inserted profiling functions).

The solution to the first problem is limiting the instrumentation of same functions. If a function is invoked many times, Safari only samples one instance in order to eliminate the overhead of repeated profile. However, this method has a potential problem: if the code path in this specific function is changed due to different parameters or input of the function, the power dissipation of this function will also change. We solve this problem by using multiple records. In the implementation, we use *bloom filter* [12] to record functions that have been profiled. Before a function is to be profiled, Safari checks the *bloom filter* first. This method is named Safari_1 in the rest of the paper. We demonstrate this idea in Figure 2. During a sampling period, the same function call is only profiled once by checking the bloom filter. Although *bloom filter* is able to control which function to be profiled, checking bloom filter itself consumes system resources. Normally this part of overhead is acceptable unless an application has extremely high rate of function calls.

The total number of instrumented functions needs to be controlled in order to solve the second problem. The overhead produced by nested function calls can be reduced if only a set of selected functions are profiled during a certain period, which means other functions execute normally without profiling. In addition, profiling module by module for a large program (for example, an application might contain over 10,000 functions) is extremely useful in order to generate accurate power profiling results. Figure 2 shows that only a group of functions are profiled for several sampling periods. By adjusting group sizes, overhead can be effectively reduced. However, some functions might not get profiled if this method is used. Commercial software, such as base station controller, is deployed to run for a considerable long period (months or years). Statistically, most of functions can be profiled in such a setting. This strategy is refereed to as Safari_2 in the evaluation part. Functions can be grouped alphabetically or according to their addresses. By adjusting the group size and the total length of the warm up period, Safari is able to generate power profile for most of functions.

Multiple Records For a frequently invoked function, we should profile it multiple times in order to generate correct power profile because the code path of an application varies. As aforementioned, a function is only sampled once in a sampling period. The result can be inaccurate if the code path changes afterward. The solution is to profile the same function during different sampling periods as Figure 2 shows. Statistically, random sampling represents characteristics of the whole sample space if the function has been invoked multiple times and the execution is sufficient long (high confidence level). For example, random sampling can be used to approximate the percentage of a path occurrence of a function with *"if" or*

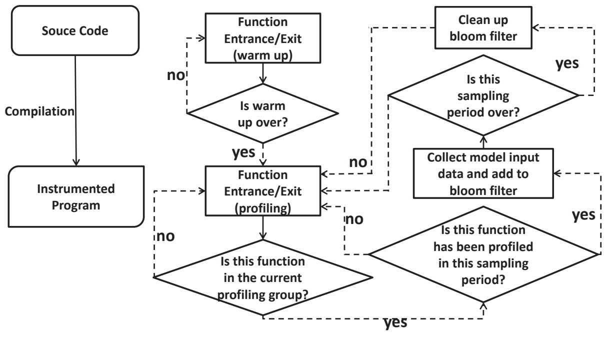
Souce Code
Compilation
Instrumented Program
Function Entrance/Exit (warm up)
no
Is warm up over?
yes
Function Entrance/Exit (profiling)
no
Is this function in the current profiling group?
yes
Clean up bloom filter
yes
Is this sampling period over?
no
Collect model input data and add to bloom filter
yes
no
Is this function has been profiled in this sampling period?

Fig. 3. Run-time profiling and information collecting.

"switch" statement in it if the profiling process is sufficient long.

Power Model In order to collect the input data for a power model, the following aspects need to be considered. 1) availability: the input data should be easy to collect. Sometimes system features constrain the data that are able to be sampled for a system. For example, usually two PMCs values can be collected simultaneously given the limited number of hardware registers [13]. 2) complementary: the collected input data is most useful when it covers a certain range of system events. Given the fact that the total amount of data can be collected is limited in order to reduce overhead, it is important to explore the resource usage of a system as much as possible. For instance, cache miss rate and bus transaction rate contains some overlapping information because CPU retrieve data from memory through bus if the data cannot be found in the last level cache. In this paper, we use OS level metrics and some commonly used PMCs as the model input:

- CPU utilization: it represents the average CPU usage during the execution of a function. The value can be retrieved from Linux PROC File System.
- Cache miss rate: it partially quantifies how frequently memory has been used for read and write. The value can be retrieved from PMCs.
- Context switch rate: we use the context switch rate to estimate the overhead of running multiple processes in a system and attribute it to the functions in threads, during which context switches occurred. The value can be retrieved from Linux procfs.
- Instruction per cycle (IPC): we use IPC values to calculate the effectiveness of a CPU executing instructions and consuming power. A strong relationship between IPC and power dissipation has been revealed in several articles [14], [8].
- CPU frequency: the frequency of CPU is directly linear related to power dissipation. The value cane be retrieved from cpufreq subsystem from Linux.

In order to construct a power model, usually a mathematical method such as linear regression or nonlinear regression is used. Given the model input, we use the following equation to estimate power dissipation: $P = a_1 \times cpu_u + a_2 \times cache + a_3 \times cs + a_4 \times ipc + a5 \times cpu_f + P_{idle}$, where $\{a_1 \dots a_n\}$ are coefficients to be determined by a set of training benchmarks. The power model is not the major concern of this paper. The model can be substituted with other models.

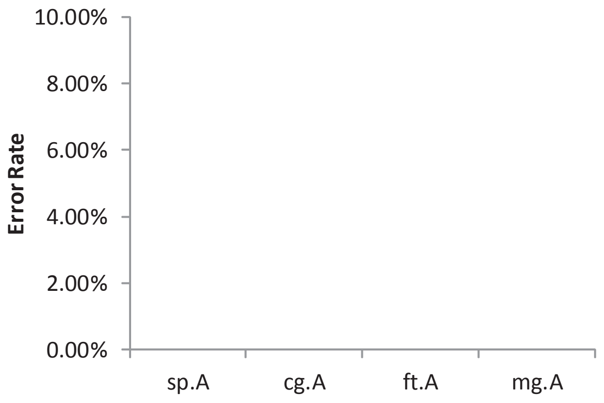

Fig. 4. Estimation error rate of CPU power for different functions, the function names are shown in Table I.

The method is summarized in Figure 3. Given the source code, 1) we compile to generate an instrumented version of executable. The source code needs to be compiled with `-finstrument-functions` option. 2) The compiled object files are linked to a static library provided by Safari, $libsafari.lib$. 3) Safari library collects run-time function resource usage information. Inside the execution, there is no information collected during the warm up period. Then, the instrumented program determines if this function has been profiled or not by checking the bloom filter during one sampling period. In addition, functions are divided into groups to reduce profiling overhead as well. The collected data are the input of the power model.

IV. Evaluation

We mainly evaluate the effectiveness of Safari and the overhead introduced by function level power profiling. The experiment platform contains a Intel Core 2 Quad 8200 CPU with 6GB memory. The processor is able to work on two frequencies, 2.00GHz and 2.34GHz. All the results in this section are generated by setting CPU frequency to 2.34GHz. We use a NiDAQ 9205 unit to record the CPU power dissipation from the 4 pin power supply on the motherboard. The original sampling rate is 1KHz. IIR low bandpass filter is utilized to filter noise. Data are re-sampled at the rate of 50Hz. We mainly use NPB3.0 benchmark OMP implementation as the target applications.

First, we use Safari to sample CPU activities and produce CPU power profiles. We utilize linear regression to construct the power model based on training benchmarks. Power model is not the major concern because Safari if flexible to use different models and run-time information. In order to obtain a stable external power measurement, we deliberately execute the target functions in a infinite loop. The error rates are demonstrated in Figure 4. The CPU power estimation has an average error rate of 6.85% for the selected four benchmarks.

TABLE I
ACTIVITIES INSIDE THE FUNCTIONS.

Benchmark	Function	IPC	CPU utilization	Cache miss rate
SP.A	compute_rhs__	1.33	97	1162.15
CG.A	conj_grad__	1.28	98	1603.67
FT.A	fftxyz_	2.08	98	438.06
MG.A	mg3p_	1.45	99	1045.28

The detailed activities inside each function are shown in Table I. The accuracy of the collected activities is closely related to the system resolution. For instance, CPU utilization is obtained from PROC File Systems, which usually utilize jiffy as the basic unit. No correct information could be retrieved if a function's execution time is beyond that resolution. However, this constraint does not affect most major functions.

Next, we measure the overhead introduced by Safari. Because Safari has two policies to reduce profiling overhead, (one sample for a function inside a sampling period and different function groups), we first only deploy the first mechanism, Safari_1. Functions are not divided in Safari_1. The profiling results are shown in Table II. The number of sampling periods is also a factor affecting overhead since more samples for each function can extend total execution time. Therefore, different sampling periods are used in this evaluation. For example, the column labeled with *overhead (1/8)* means that there are 8 sampling periods totally during the execution. In other words, maximum 8 samples can be collected for each function during application execution. The overhead is measured as execution time when we explore profiling techniques.

As Table II shows, the overhead generated by Safari is comparable with that of generated by gprof in most cases. As expected, the overhead increases slightly as the number of sample collected increases. Overall, the overhead generated by OMP version of benchmarks is higher compared with SER cases for both Safari and gprof because the contention of recording information in one single file for multiple threads. It is obvious that the overhead generated by BT and IS benchmark is as high as 546% on average for SER and OMP benchmarks. The root reason is because these two benchmarks have extremely high function calls rates. On average, the function call rates of BT and IS are 203 times higher than that of the rest five benchmarks. The BT benchmark has nested function calls that generate excessive overhead.

We deploy both Safari_1 and Safari_2 to reduce overhead for BT and IS benchmarks, especially. The number of sampling periods is denoted as n. We divide functions in a workload into m groups, where $m \in [1, n] \wedge n = am, a \in \mathbb{Z}$. If $m = 1$, the effect of Safari_2 disappears. This setting is for simplicity. The values of m and n are more flexible if execution time is long enough. We evaluate Safari_2 on BT and IS benchmarks with at most one sample is collected for each function. The results are shown in Table III. For a fixed m value, as the n increase, the total overhead increases as well since more samples are collected. If n is fixed, the total overhead drops as m doubled because trivial functions might not be profiled with a bigger m value. However, major functions in both benchmarks are profiled because they usually iterate for more than one time.

TABLE III
PROFILING OVERHEAD WITH SAFARI_2.

Type	Benchmark	n/m	2	4
SER	BT	4	29.62%	26.45%
		8	56.81%	42.32%
	IS	4	16.86%	18.62%
		8	24.30%	23.34%
OMP	BT	4	57.49%	45.65%
		8	82.42%	54.21%
	IS	4	14.14%	12.45%
		8	28.4%	20.60%

In order to further measure the overhead, we let the program to profile only one function repeatedly. The results are used to compare with the execution time without profiling. Besides, the aforementioned platform, we use Cavium 6300 evaluation board as an example of embedded system. The board is equipped with six cnMIPS II processor cores, 4GB DDR3 memory and some other co-processing units such as compressor and encrypter. The results are shown in Table IV. The profiling overhead means that functions are instrumented and model input data is collected. To profile a function once introduces about 0.8ms overhead on Cavium 6300 evaluation board. While, if only instrumented without actually profiling (for example, the program found the function is already been profiled during a sampling period) consumes much less overhead. Both of them is neglectable compared to a function body conducting 512*512 matrix calculation which takes few seconds. In addition, as we avoid frequent profiling for a given sampling period, the overall overhead is under restrict control.

TABLE IV
PROFILING OVERHEAD

Platform	Profiling overhead (sec)	Instrumentation overhead (sec)	512*512 matrix mcl (sec)
Cavium 6300	$8 * 10^{-4}$	$5 * 10^{-7}$	7.4
Intel Core 2	$4 * 10^{-4}$	$2 * 10^{-7}$	1.4

V. RELATED WORK

Software power dissipation is directly related to dynamic power, which becomes an increasing portion under the context of energy-proportional computing. However, only few research projects focus on software power analysis.

Although system level power managements has effectively been investigated in recently years, there is a realization that software has dramatic impact on power dissipation. Therefore, in-depth understanding software power dissipation becomes one of the major consideration when designing power-aware systems. Ge *et al.*, propose PowerPack [10] to generate component level power profile. This approach targets on the cluster level. PowerPack provides APIs to synchronize external power measurement and function execution of the target application. However, manual instrumentation is inconvenient for large

TABLE II
PROFILING OVERHEAD WITH SAFARI_1.

Type	Benchmark	Overhead (8/1)	Overhead (4/1)	Overhead (2/1)	Overhead (1)	Overhead (gprof)	Call rate (calls/sec)
SER	CG	9.89%	6.91%	6.18%	6.04%	1.03%	68989.07
	MG	1.89%	0.52%	0.78%	0.31%	0.21%	99.26
	FT	1.49%	1.00%	0.93%	0.65%	0.50%	2719.37
	EP	0.45%	0.27%	0.05%	0.33%	0.47%	0.33
	LU	1.08%	1.01%	0.98%	0.75%	0.35%	14591.35
	SP	11.09%	10.89%	10.85%	10.72%	1.29%	102004.79
	BT	288.94%	288.14%	287.27%	286.48%	5.36%	3706727.08
	IS	908.28%	907.25%	905.39%	911.84%	15.23%	11158773.53
OMP 4 threads	CG	42.21%	21.04%	11.90%	18.28%	47.54%	38134
	MG	72.36%	39.02%	24.64%	15.54%	2.67%	175.93
	FT	41.44%	32.04%	33.77%	20.99%	21.24%	77491.14
	EP	1.34%	1.32%	1.21%	0.94%	0.39%	0.17
	LU	2.11%	2.10%	1.78%	1.55%	1.55%	6347.21
	SP	3.66%	3.48%	3.33%	3.03%	1.93%	15996.28
	BT	309.58%	298.44%	292.87%	284.51%	63.56%	4936017.01
	IS	508.08%	499.23%	490.79%	495.19%	124.48%	2905743.91

scale applications. Hänig *et. al*, propose SEEP [15], which uses symbolic execution to explore possible code paths and entities in a program and to generate energy profile for specific target platform. Instruction level energy profile is needed for each platform in advance in order to generate energy profile for a program.

Moreover, as the energy consumption and power dissipation of a computer system stem from the interplay of hardware and software, they must be considered equally important. Bhattacharya *et al.* propose an analytical model to estimate energy cost of software bloat on a specific platform [5]. The results show that reducing software bloat can achieve as much as 40% energy saving. However, the study shows both hardware and software need to be considered to improve energy efficiency.

VI. SUMMARY

In this paper, we propose a function level power profiling tool, Safari. It can be used to associate run-time resources usage with the execution of application functions. Finding the power hot spot is only the first step towards power-aware programming. The experiment results show that Safari is able to produce function level profiling with limited overhead (on average 16% overhead if maximum one sample is collected for each function). It can be used to connect application activities to hardware for energy-efficient design, such as application aware power management and fine-grained scheduling.

REFERENCES

[1] S. Reda, "Thermal and power characterization of real computing devices," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 1, no. 2, pp. 76 –87, june 2011.

[2] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceeding of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 319–330. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000103

[3] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[4] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," *SIGPLAN Not.*, vol. 44, pp. 205–216, March 2009. [Online]. Available: http://doi.acm.org/10.1145/1508284.1508269

[5] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "The interplay of software bloat, hardware energy proportionality and system bottlenecks," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, ser. HotPower '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:5. [Online]. Available: http://doi.acm.org/10.1145/2039252.2039253

[6] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere, "Method-level phase behavior in java workloads," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 270–287. [Online]. Available: http://doi.acm.org/10.1145/1028976.1028999

[7] W. Bircher, M. Valluri, J. Law, and L. John, "Runtime identification of microprocessor energy saving opportunities," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, aug. 2005, pp. 275 – 280.

[8] S. Wang, H. Chen, and W. Shi, "SPAN: A software power analyzer for multicore computer systems," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23 – 34, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S221053791000003X

[9] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810108

[10] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 658–671, 2010.

[11] "Pin-A Binary Instrumentation Tool," http://www.pintool.org/, Aug 2010.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[13] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, sept. 2010, pp. 207 –216.

[14] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 160–171, 2003.

[15] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "Seep: exploiting symbolic execution for energy-aware programming," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, ser. HotPower '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/2039252.2039256