Fast and Effective Power Profiling of Program Execution Based on Phase Behaviors

Xiaobin Ma*, Zhihui Du*† and Jason Liu‡

* Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, China
† Corresponding Author's Email: *duzh@tsinghua.edu.cn*
‡ School of Computing and Information Sciences, Florida International University, USA

***Abstract*—Power profiling tools based on fast and accurate workload analysis can be useful for job scheduling and resource allocation aiming to optimize the power consumption of large-scale high-performance computer systems. In this paper, we propose a novel method for predicting the power consumption of a complete workload or application by extrapolating the power consumption of only a few code segments of the same application obtained from measurement. As such, it provides a fast and yet effective way for predicting the power consumption of a single-threaded execution of a program on arbitrary architectures without having to profile the entire program's execution. The latter would be costly to obtain, especially if it's a long running program. Our method employs a set of code analysis tools to capture the program's phase behavior and then adopts a multi-variable linear regression method to estimate the power consumption of the entire program. We use SPEC 2006 benchmark to evaluate the accuracy and effectiveness of our method. Experimental results show that our power profiling method achieves good accuracy in predicting program's energy use with relatively small errors.**

I. Introduction

High-performance computing platforms, large-scale data centers, and cloud computing facilities are among the largest consumers of energy. According to a report issued by the Natural Resources Defense Council [1], the electricity consumption of U.S. data centers in 2013 is estimated to have reached 91 billion kilowatt-hours. The total energy demand of data centers has been projected to increase from 1.3% of worldwide electricity supplies in 2010 to 8% in 2020 [2]. By then, their carbon footprint will exceed that of the airline industry.

Power consumption has thus become an important decision variable for scale and performance for systems and applications. Understanding energy use of the applications and programs running on the high-performance computing platforms and data centers is a critical step in optimizing power management, resource allocation, and scheduling of program execution in order to reduce the systems' overall power consumption.

Existing approaches to power profiling include physical measurements, either by using special hardware instruments to measure power consumption at various computer components (e.g., [3]–[5]), or through embedded power meters and on-board sensors (e.g., [6]–[8]). Physical measurements can provide accurate power monitoring and power measurement in real time (subject to certain time granularity). In particular, with a programming interface, an embedded power meter can be of great help to designers of power-aware systems and applications. However, physical measurements report only the power consumption of a specific component or an entire system. It is difficult to translate measurements to energy use of individual application programs. Furthermore, physical measurements alone cannot be used for prediction of future power consumptions as they report only the current state.

Power profiling also includes analytical approaches. In this case, simulation can be used to estimate power consumption of various components at the microarchitectural level [9]–[12]. While simulation may provide a cycle-accurate estimate of power consumption from program execution, it can also be extremely time-consuming. In addition, simulation oftentimes considers only simplistic scenarios and ignores more complex effects, such as those from real operating systems (I/O, multi-tasking, and so on). Another type of analytical approach is to estimate power consumption by establishing correlations from the program's hardware performance events (e.g., [13], [14]). These approaches can provide power profiling more efficiently, but requires off-line analysis of logs.

In this paper, we present a fast and effective program power profiling technique based on program phases. We observe that applications exhibit distinct behaviors during execution, which often fall into repeating patterns, called program phases [15]. These phases have several important features:

- A program may consist of many program phases, which alternate and oftentimes repeat throughout the program's execution.
- Program phases occur at large time scales (typically consisted of execution of hundreds of millions to tens of billions of instructions).
- Within each program phase, the performance metrics of the program, such as the cache miss ratio, branch misprediction rate, execution speed (measured in instructions per cycle), as well as power consumption, maintain relatively stable.
- Between consecutive program phases, the program's performance metrics change significantly (sometimes by several orders of magnitude). They change synchronously,

although not necessarily proportionally.

Based on these observations, we propose to correlate program phase detection with power prediction. A program's execution is divided into intervals, each of which consists of a fixed number of instructions of execution (say, 100 million). A program phase can be defined as a set of intervals with similar behavior [15]. Note that the intervals belonging to the same program phase may not be adjacent to one another in time.

To detect program phases, we use an offline program analysis tool, called simpoint [16], which clusters the intervals according to the ratios in which different regions of the program (basic blocks) are being executed over time. We note that such classification can also be achieved online through dynamic branch profiling [15]. However, in this study, we limit to only off-line analysis.

For each program phase found, we choose one representative interval (say, at the centroid of the cluster) and measure its power consumption. This can be achieved using physical power measurement tools. In our study, we use the Running Average Power Limit (RAPL) toolkit designed for microarchitectures with embedded sensors (such as Intel's Sandy Bridge) [6]. To assess the power consumption of other intervals, we simply calculate the distance between the intervals in terms of the executed instructions, which we call *Executed Instruction Vector (EIV)*, and use a similarity metric to estimate its power consumption based on multi-variable linear regression.

EIV is a vector with elements each representing the number of times a certain type of instruction has been executed with an interval. For instance, if an interval contains 2 `mov` instructions and 3 `add` instructions, the values of the elements in the interval's EIV representing the two instructions will be 2 and 3, respectively. In doing so, we can easily define the distance (which is the inverse of similarity) between two arbitrary intervals using an algebraic distance between the two EIVs. Note that the EIV only represents the mixture of the instructions being executed within an interval without considering the order in which they are executed. We show that this first-order approximation in most cases is sufficient to identify different power regimes within a program. Also note that, in practice, one does not need to maintain a high dimension vector for each interval, since usually only a subset of instruction types are present within an interval. Therefore, we can simply use a dictionary that maps from the instruction type to the number of occurrences as a succinct way of representing EIV.

The major contributions of this paper are two-fold. First, we show that using program phase behaviors we can quickly establish the baseline for the program's power consumption. Second, we show that using a simple similarity metric (by comparing EIVs between the intervals) one can predict power consumption of the entire program's execution with good accuracy. Using this method, we can effectively predict the power consumption of long running programs based on measurements of only a small set of intervals. Presumably, our method can also be extended for *online* power prediction if one can incorporate our method with effective online classification techniques for program phase detection and using online power measurement tools for power estimation. In this paper, we focus only on static power profiling, and defer the online methods for future work.

The rest of the paper is organized as follows. In section II, we discuss background and related work. We first provide an overview of our power profiling method in section III, followed by a detailed discussion of the specific techniques in sections IV and V. We present validation experiments and the results in section VI. We discuss possible ramifications of our approach in section VII and finally conclude our paper in section VIII.

II. Background and Related Work

Power consumption is a measure of rate at which a computer system is consuming energy. It is a reflection of how various components of the computer is utilized over time. While most programs show variable behavior during their run time, they typically exhibit a certain amount of repetitive behavior—for example, a numeric kernel loops through the same regions of an array or a matrix at certain periods of time. Program phase analysis is a method of identifying such repetitive behavior; it can be used for workload characterization, resource management, dynamic performance and power optimization, and so on.

During a program's execution, it passes through several phases where its hardware resource requirements and consequently its performance characteristics vary. One effective method for detecting program phases is by observing changes in the program's memory demand, such as the working set, which can be defined as a collection of memory locations recently referenced by a program [17]. For example, Dhodapkar and Smith [18] proposed a phase detection method using working set signatures, which are highly compressed working set representations using only a small number of bytes (on the order of 32-128 bytes).

Program phases can also be detected by directly observing the program's execution trace. Sherwood et al. [19] introduced the concept of Basic Block Vectors (BBV). A basic block is a straight-line sequence of instructions that has but one entry point and one exit point (in particular, no branches during execution). When a program is executed, it runs through many basic blocks, each of which may be executed multiple times. By taking a snapshot of the number of times each basic block is executed within a sampling period (i.e., a fixed number of executed instructions), one can obtain a vector representing the proportion of basic block executions, which can then be used as a signature of the program's activities during this period.

BBV can be used to identify program phases using clustering techniques [20]. To find how intervals of program's execution relate to one another, we can quantify similarity between two intervals as the Manhattan distance between their basic block vectors. Using this metric, one can cluster all the intervals into a small set of groups, where each group can be identified as belonging to a distinct program phase and subsequently represented by a sample point chosen to be the closest to the group's centroid. This method has been shown to be very effective in detecting program's phase behaviors. In addition, it can be extended for online phase detection since the basic block executions can be approximated using instruction counts separated by branches [15].

Our power profiling method proposed in this paper uses Sherwood's phase analysis scheme. In particular, we establish a correlation between distinct regimes of power consumption and program phases. Our method is based on the observation that the program's phase behavior, shown as distinct stable periods in performance and power consumption, is directly a function of the code being executed.

Contrary to our method, Isci and Martonosi [21] proposed a method of identifying a program's phases through its power behavior. Analogous to BBV, they use a power vector, which consists of estimated power values for 22 processor components, to be the signature for the intervals. Accordingly, they defined a similarity metric and used a thresholding algorithm to partition distinct power behaviors into groups. Their method can be seen as complementary to ours, as both methods consider correlation between program behavior and power consumption. Their method uses power measurements to identify phases, while our method uses phases to predict power consumption. We note, however, that our method does not depend on detailed power measurements of the system. We take advantage of advanced program analysis tools, which are considered to be more mature today and should be readily available in most modern systems.

III. Overall Design

Our power profiling method is based on the identification of program phases. In particular, we use Sherwood's method for detecting phases [15]. A program's execution is divided into a sequence of slices, called intervals, each with a fixed number instructions (we chose to use 100 million executed instructions per interval). Separate tools are used to obtain the basic block vectors (BBVs) for the intervals (which we describe in the next section), in addition to those included in the distribution of the simpoint tool.

Simpoint is an offline program analysis tool developed by Sherwood et al. at UCSB [16]. It takes as input a list of BBVs, one for each interval, and then runs the k-means algorithm to cluster the intervals. The result grouping is output as the phases. To save computational time, simpoint uses random linear projection to reduce the dimensionality of the BBVs. To determine the final number of groups, simpoint first runs the clustering algorithm with different k and then chooses the one with the best fit (i.e., having the highest Bayesian information criterion index).

Although program phases can be identified using simpoint, we still need to establish correlation between distinct regimes of power consumption and program phases. To obtain a power model, we need to solve two problems:

1) We need a method to succinctly describe program behaviors using necessary measurements that can help determine power consumption at each interval.
2) We need a method to effectively predict power consumption based on the measurements collected at each interval.

To address the first problem, we recognize that program's power results from running the program code that engages with different parts of the system, such as processing (integer, floating-point, branch prediction) and memory access (caches at different levels, TLB). In [22], the authors proposed to measure the power consumption at 22 different components at the microarchitecture. We resort to applying simple program analysis that can be piggy-backed to the calculation of BBVs so that our method can be used together with simpoint during phase detection.

While power consumption can be largely influenced by the intensity of numerical calculations, memory access patterns, and so on, it is ultimately determined by the type and the number of instructions being executed, which can be obtained using program analysis. We define a vector, called the *Executed Instruction Vector (EIV)*, to represent the executed instructions at each interval. EIV can be derived from BBV. In this case, we expect our method can be extended so that EIVs can be obtained during the program's execution since BBVs can be approximated using online classification methods [15]. In section IV we discuss the details of our method for obtaining EIV.

To address the second problem above, we need to find an effective method to map from the workload characteristics of the intervals to their power consumption. Many power models exist. We choose to use an empirical method. Once the program phases have been determined, we can measure the power consumption of the representative intervals, one for each identified program phase. We can choose the interval nearest to the centroid of the group corresponding to the program phase as the representative interval. Power estimation of the interval can be achieved by using measurement tools, such as RAPL. RAPL is Running Average Power Limit toolkit developed by Intel to measure power of CPU and memory. It relies on Model Specific Registers (MSRs) that report CPU and memory access events to estimate power consumption [6].

Once we determine the power consumption of the representative intervals, we can predict the power consumption of other intervals using a similarity metric. We use the Euclidean distance between the EIVs to represent the similarity between the intervals and apply a multi-variable linear regression method to determine the power consumption of all other intervals based on the measured power consumption at the representative intervals. The details of our method are discussed in section V.

IV. Executed Instruction Vectors

A running program consists of a set of instructions executed in order. We want to characterize the distinct behavior of a program, from which its power consumption can be derived. This can be achieved through static program analysis and by inspecting the program's execution log. In this section, we describe a method that can be piggy-backed with the existing method for detecting program phases.

Our method allows for counting the number of executed instructions of various types during an interval. Currently, we focus only on the type of instructions and ignore the effect from the sequence of their execution on power consumption. We have two reasons. First, as shown later in this section, the number of executed instructions during an interval can be obtained relatively easily using static program analysis from the basic block vectors created for phase analysis. Second, in our method, the power consumption of an interval is first determined by the program phase (i.e., the group which the interval belongs to). Since different intervals in same program phase are expected to differ only slightly with respect to the frequency of visits to the same set of basic blocks, the difference in the power consumption between them is more likely attributed to the difference in the numbers of executed instructions, rather than their execution order.

In the rest of this section, we describe a relatively straightforward way to count the number of executed instructions of different types within an interval. A basic block vector is a vector with h elements, one for each basic block in the program. Let B be the set of basic blocks in the program (i.e., $h = |B|$). During an interval, let v_i be the number of times program execution visits basic block i (where $i \in B$). Let s_i be the total number of instructions in basic block i. The element in the basic block vector is therefore the number of visits to the basic block v_i, multiplied by the number of instructions of the basic block s_i. The vector is the normalized by dividing each element by the sum of all elements in the vector [15]. More formally, we denote the basic block vector of the current interval as $BBV = (b_1, b_2, \cdots, b_h)$, where $b_i = v_i s_i (\sum_{k \in B} v_k s_k)^{-1}$. The interval length, denoted by L, is fixed (in our studies, we use 100 million instructions). That is, the denominator is approximately the same: $L \approx \sum_{k \in B} v_k s_k$.

The basic block vector is then used to cluster the intervals into groups, identified as program phases. However, it does not contain information about what specific instructions are being executed. This information would be important for characterizing the power consumption of the intervals. We define the Executed Instruction Vector (EIV) to be a vector, where each element represents the number of instructions being executed for a specific type of instruction during an interval. More formally, $EIV = (e_1, e_2, \cdots, e_m)$, where m is the total number of instruction types at the target architecture.

If we know the distribution of the type of instructions at each basic block, we can calculate EIV from BBV. More specifically, we can calculate the number of executed instructions of each type using the number of instructions belonging to that type in each basic block multiplied by the number of times program execution visits the basic block during the interval. More formally, let $I_i = (a_{i1}, a_{i2}, \cdots, a_{im})$ be the instruction distribution vector for basic block i, where a_{ij} is the number of instructions belonging to type j. We can calculate the elements of EIV: $e_k = \sum_{i \in B} a_{ik} v_i$, where v_i is the number of times program execution visits basic block i. We can derive v_i from BBV: $v_i = b_i L / s_i$, where L is the interval length and $s_i = \sum_{0 \le k < m} a_{ik}$ is the total number of instructions in basic block i.

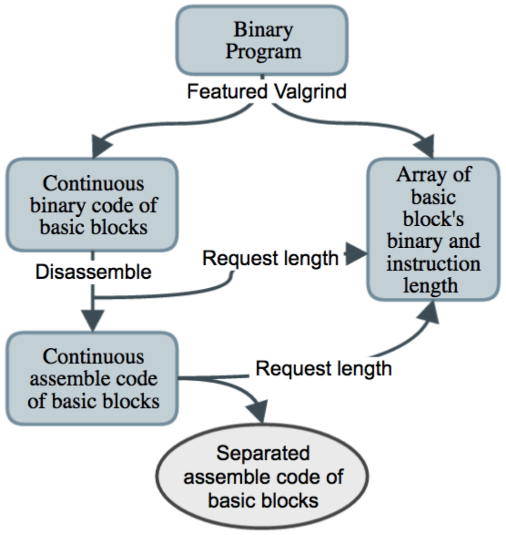
Binary Program
Featured Valgrind
Continuous binary code of basic blocks
Array of basic block's binary and instruction length
Disassemble
Request length
Continuous assemble code of basic blocks
Request length
Separated assemble code of basic blocks

Fig. 1. Using valgrind to obtain instructions of basic blocks.

In our implementation, we use valgrind [23] to generate both BBV and EIV. BBV is given to simpoint [16], which uses it to detect program phases. We use EIV to estimate power consumption (described more in the next section). Valgrind is an open source instrumentation framework for building dynamic analysis tools for debugging and profiling, including an experimental tool for generating basic block vectors. The tool already generates important information for each basic block, including its starting address, the name of the function containing the basic block, and the number of instructions in the basic block. Conceptually, one can directly use the information to obtain the instruction distribution vector using a disassembler (such as `objdump`). In practice, however, we encountered several problems that prevented us from correctly identifying the function names in dynamically linked libraries and as a result we chose a different path.

We modified the code in valgrind for analyzing the basic blocks. Every time a basic block is entered that has never visited before, in addition to just counting the number of instructions in the basic block, we write the instructions out to a log file. Later we use a disassembler (in our case, GDB) to restore the instructions belonging to each basic block. The process is illustrated in Fig. 1.

V. Power Estimation via Multi-variable Linear Regression

We would like to estimate the power consumption of the entire program using the power consumption at a few reference intervals. For each program phase, we choose to use the interval that is closest to the centroid of the group as the representative of all intervals belonging to that phase. We establish a similarity metric based on the Euclidean distance

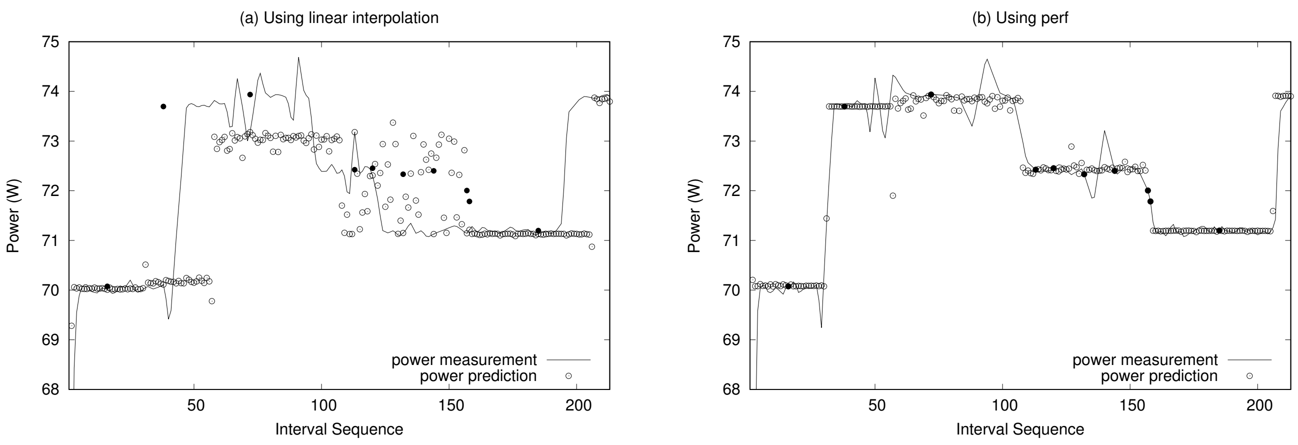

Fig. 2. Power prediction accuracy using linear interpolation vs. using `perf` (for benchmark `473.astar`).

between the executed instruction vectors of the intervals and use the coefficients calculated using a multi-variable linear regression method to project the power consumption of all the intervals from the representatives.

Let $c_1, c_2, \cdots, c_g$ be the representative intervals (one from each program phase), where g is the total number of program phases. We quantify the similarity between two intervals, a and b, using the Euclidean distance between their EIVs. Suppose $EIV(a) = (e_{a,1}, e_{a,2}, \cdots, e_{a,m})$ and $EIV(b) = (e_{b,1}, e_{b,2}, \cdots, e_{b,m})$, the similarity between a and b can be defined as:

$$\alpha_{a,b} = \sqrt{\sum_{0 \le k < m} (e_{a,k} - e_{b,k})^2} \tag{1}$$

For any interval k, we can calculate the similarities from k to all representative intervals, which we denote using a vector: $\boldsymbol{x_k} = (\alpha_{k,c_1}, \alpha_{k,c_2}, \cdots, \alpha_{k,c_g})^T$.

We define matrix $\boldsymbol{X} = (x_{c_1}, x_{c_2}, \cdots, x_{c_g})^T$. Let $\boldsymbol{y}$ be the vector containing the power consumption at all representative intervals. That is, $\boldsymbol{y} = (P(c_1), P(c_2), \cdots, P(c_g))^T$, where $P(k)$ is the power consumption for interval k. The linear regression model can be expressed as $\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{\beta} = (\beta_1, \beta_2, \cdots, \beta_g)^T$ are the coefficients and $\boldsymbol{\epsilon} = (\epsilon_1, \epsilon_2, \cdots, \epsilon_g)^T$ represent the noise. We can use the standard multi-variable linear regression method to solve for $\boldsymbol{\beta}$, and then use them to predict the power consumption of any interval k, using the equation $\boldsymbol{x}_k^T \boldsymbol{\beta}$.

To estimate the power consumption at the representative intervals, currently we use measurements from RAPL [6]. RAPL, or Running Average Power Limit toolkit, provides the energy and power consumption information through a set of counters. It is a software power model that relies on performance-related events recorded in hardware performance counters and an I/O model to estimate energy usage.

RAPL collects power consumption based on time. The intervals for phase detection, however, use instruction count (say, 100 million). There can be discrepancies in this case. A straightforward approach is to apply linear interpolation to estimate power consumption at the interval boundaries. We found that linear interpolation can provide good accuracy for programs whose power consumption do not changed significantly. However, the problem with this approach is that it can introduce significant errors when the boundaries of the time window used by RAPL belong to different phases with significantly different power regimes. Fig. 2(a) shows an example of power prediction of a benchmark program using linear interpolation; significant errors can be observed especially in the middle section of the plot.

To solve this problem, we use a tool, called `perf`, which is available on Linux platforms. The tool is is able to record both the instruction count and the execution time of an interval. Having an accurate representation of the power consumption for all representative intervals is essential for the overall prediction accuracy of the entire program's execution. Using `perf` turns out to be a far more reliable method for power prediction than linear interpolation. Fig. 2(b) shows the same benchmark example using `perf`. The power prediction in this case is quite accurate. The reason is that we use the power consumption of the representative intervals as the training set. As such, any errors occurred in matching the power data from RAPL and the intervals used by simpoint for phase detection can be amplified. Our experiment using the SPEC 2006 benchmarks shows that on average using `perf` can reduce relative error from 5.3% using linear interpolation to 3.8%, and the standard deviation from 5.53 using linear regression to 4.79.

Another issue related to using RAPL is that the power measurement done by RAPL is at the processor level, not at the core level. This is because the event counters, called MSR, used by RAPL to register performance-related events for the software power model are integrated with the physical processor, not core. In a processor with high core count (such as the one we use that contains 12 cores), running a single threaded benchmark (occupying a single core) may contribute only a small proportion of the total power consumption of the entire processor. In fact, the other cores on the same

TABLE I
BENCHMARK RESULTS

Benchmark	Phases	Intervals	σ	Error
401.bzip2	9	180	2.06	0.020
403.gcc	6	43	7.79	0.084
410.bwaves	5	808	14.24	0.082
429.mcf	14	31	3.12	0.027
433.milc	10	332	6.80	0.058
435.gromacs	10	47	3.42	0.027
436.cactusADM	4	92	1.63	0.015
437.leslie3d	16	399	1.13	0.006
444.namd	23	606	1.13	0.005
445.gobmk	9	384	4.97	0.028
453.povray	3	24	3.42	0.100
456.hmmer	7	142	1.46	0.016
458.sjeng	4	143	1.99	0.009
464.h264ref	20	851	5.60	0.030
465.tonto	17	35	5.46	0.061
470.lbm	7	66	1.75	0.009
471.omnetpp	3	17	9.26	0.090
473.astar	12	235	5.33	0.025
482.sphinx3	7	68	2.91	0.018
average	**19**	**356**	**4.40**	**0.038**

processor may be loaded to run other kernel threads that are not controlled in our experiment. To solve this issue, we decided to run multiple instances of the same single threaded benchmark program simultaneously so as to occupy all cores of a processor. So long as the processes are running synchronously through the phases, which is the case given that in our experiments each interval consists of 100 million instructions, the power measurement collected by RAPL at the processor level would be simply magnified by the number of cores in the processor.

In practice, we use a for-loop to launch the different instances of the same benchmark program (using the same runtime parameters). In the experiment, the length of the interval is around 50 milliseconds. We have observed that the maximum time difference between the first and last instances of the benchmark never exceeds 20 milliseconds. There would be a slight discrepancy in the power measurement; however, as long as the neighboring intervals of a representative interval belong to the same phase as the representative interval, it would have very little effect on the accuracy of the power estimation.

VI. EXPERIMENTS

We use SPEC CPU 2006 for our experiments. SPEC CPU 2006 is an industry standard CPU performance benchmark. It contains two sets of benchmark programs: CINT2006 has 12 programs for integer performance test and CFP2006 has 17 programs for floating point performance test. We chose 19 single-threaded benchmark programs that can execute long enough for our model to capture the power behavior. The experiments were conducted on a Linux workstation with a 12-core Intel® Xeon® CPU E5-2670 v3 and 128 GB of memory. We used gcc compiler version 4.1 with optimization level 2 (`-O2`) to compile all the benchmark programs.

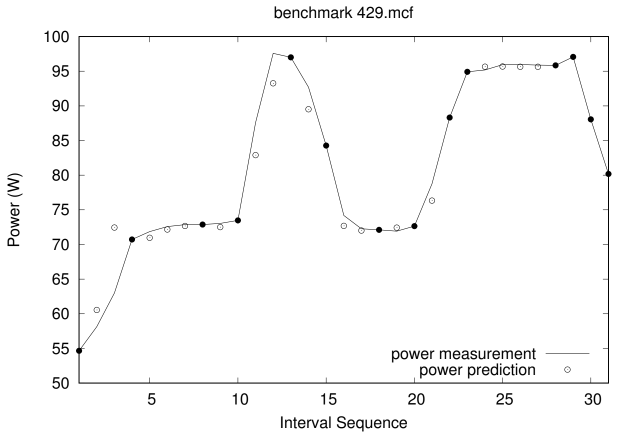

Fig. 3. Power measurement and prediction for benchmark `429.mcf`.

The results are shown in Table I. The "Intervals" column shows the number of intervals (each with 100 million instructions) in the execution of each benchmark program. The "Phases" column lists the number of program phases (i.e., clusters) identified by simpoint. We use standard deviation (σ) and relative error ("Error") to measure the accuracy of our model. For all 19 chosen benchmark programs, there are a total of 6,764 intervals and 361 phases detected. The same number of representative intervals as the number of phases were used to train our model (that's 5.3%).

The result is satisfactory. The average standard deviation among all chosen benchmark programs is 4.79 and the average relative error is 3.8%. The maximum relative error is 10% (for benchmark `453.povray`). In general, we observe that the accuracy of our model is better for benchmarks whose power consumption is more or less stable during the execution of the program; the model generates slightly higher errors for benchmarks whose the power consumption fluctuates frequently. Overall, our model shows good accuracy in predicting the power.

Fig. 3 shows a simple example (in this case, benchmark `429.mcf`), which we use to show how our model is taking advantage of a few training data points to estimate the power consumption of the entire program. The program's execution is divided into 31 intervals grouped into 14 phases. We use the measured power consumption of the 14 representative intervals to train the model, which subsequently estimates the power consumption for the rest of the intervals. Solid circles in the figure are those used used to train the model; they are the measured power of the intervals selected as the representatives (closest to the centroid of the respective phase). The hollowed circles in the figure are estimated power from our model.

It is important that we choose representative intervals as the training set for improved prediction accuracy. Fig. 4(a) shows the results from the power prediction model if we simply select the intervals randomly. The results show large variations and poor prediction accuracy. Fig. 4(b) shows the results of using representative intervals from phase detection. Simpoint selects

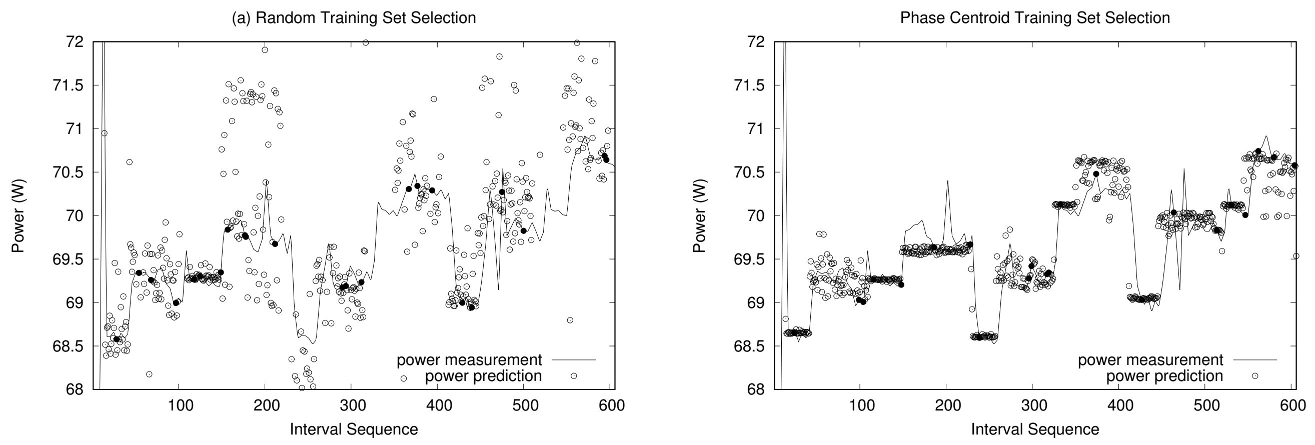

Fig. 4. The importance of training set selection (results for benchmark `444.namd`).

the intervals closest to the centroid of each detected phase. The method shows far better results: the standard deviation is reduced from 12.51 to 1.023, and the relative error from 22.27% to only 2.66%.

VII. Discussion

Our preliminary results show that the power model is effective; yet several issues exist which would surely warrant further investigation.

First, we can reduce the dimensionality of the instruction distribution vectors collected at the basic blocks. By doing so, we can reduce the computation and memory cost for collecting them and for computing the similarities. Many instructions are simply variations of one another; for example, the conditional move instructions come with many forms: `cmova`, `cmovb`, `cmove`, `cmovg`, and many others. By combining similar instruction types, according to their primary function and the expected power consumption (e.g., arithmetic operations, data transfers, control flows, and I/O instructions). we can group the instruction set into fewer categories.

Second, we use the Euclidean distance to quantify the similarity in power consumption between the intervals, in which case all instruction types contribute equally to the energy use. This is obviously problematic. For example, compute intensive operations (such as arithmetic and logic operations) should have a different power profile than those instructions involving memory accesses or engaging with I/O operations. Consequently, we should use different weights for different types of instructions when calculating the similarities according to their relative effect on the power consumption. We believe the weights can be obtained through measurements.

Third, we use the instruction distribution as the first-order indicator for the difference in the power consumption of the intervals. While instruction types do play an important role in determining the energy use, the execution order of the instructions may also be important. Our method can be extended to indirectly include the effect of the execution sequence, by collecting statistics related to computation intensity at the basic blocks, e.g., the ratio between the number of executed instructions and the memory operations. Another place worthy of further investigation is to consider the execution sequence of the basic blocks. They may also influence the power consumption.

Fourth, the effectiveness of the linear regression method may depend on the size of the training set, which in our case is the number of identified program phases, which could be either too many or too few. We have not fully investigated such effects. Also, the elements in the training set may have unequal variance (heteroskedasticity), in which case the level of noise in the data may be dependent on what region the intervals are located in the feature space. Furthermore, we need to understand that linear regression could be fundamentally limited in that the power profile from instructions may not be linear at all.

Last, a fundamental limitation of our current approach is that the power prediction model is limited to single thread execution only. In a multithreaded setting, the program phase, for general applications, may not be as obvious as it would depend on the mixing of separate threads of control, which can oftentimes be nondeterministic. For many *scientific* applications, however, such as those using OpenMP, the programs may still exhibit a good deal of phase behavior. In this case, our method will still apply.

VIII. Conclusions

During the execution, a program typically exhibits a set of distinct behaviors or phases that are often repeated throughout the execution. These phases occur at large time scales (tens of milliseconds or even seconds), during which the program's performance and power consumption stay relatively stable. Based on this observation, we propose a new power prediction method that extends a cluster-based program phase detection scheme. We apply program analysis to identify the types and the number of instructions being executed during fixed intervals, and we show that by coupling that with the program phase behaviors we can quickly establish a baseline for predicting the program's power consumption. In particular, we show that, using the power measurement of a few representative intervals, we can apply a multi-variable linear regression method to estimate the power consumption of entire program with good accuracy.

Our method is fast and effective for profiling the power consumption by analyzing program execution. In future work, we would like to extend our method to incorporate online power prediction. In particular, we would like to integrate an online classification technique for program phase detection and couple that with advanced power models, to order to achieve accurate online power prediction.

Acknowledgment

This research is supported in part by the National Natural Science Foundation of China (Nos. 61440057 and 61363019), the Sci-Tech Interdisciplinary Innovation and Cooperation Team Program of the Chinese Academy of Sciences, the Specialized Research Fund for State Key Laboratories, MOE Research Center for Online Education Foundation (No. 2016ZD302), and the National Key Research and Development Program of China (No. 2016YFB1000602). We would like to thank the anonymous reviewers for their constructive comments.

References

[1] Natural Resources Defenese Council (NRDC), "Data center efficiency assessment, scaling up energy efficiency across the data center industry: evaluating key drivers and barriers," http://www.nrdc.org/energy/data-center-efficiency-assessment.asp, 2014.

[2] J. Koomey, "Growth in data center electricity use 2005 to 2010," *Analytics Press*, August 2011.

[3] 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 Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 658–671, 2010.

[4] J. H. Laros, P. Pokorny, and D. DeBonis, "PowerInsight - a commodity power measurement capability," in *Proceedings of the 2013 International Green Computing Conference (IGCC)*, 2013, pp. 1–6.

[5] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, "PowerMon: Fine-grained and integrated power monitoring for commodity computer systems," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, 2010, pp. 479–484.

[6] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, 2012.

[7] J. Demmel, A. Gearhart, J. Demmel, and A. Gearhart, "Instrumenting linear algebra energy consumption via on-chip energy counters," UC Berkeley, Tech. Rep. UCB/EECS-2012-168, 2012.

[8] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, 2012.

[9] Mentor Graphics Corporation, "QuickPower," 1997.

[10] Synopsys Corporation, "Powermill data sheet," 1999.

[11] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of SimplePower: A cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference (DAC'00)*, 2000, pp. 340–345.

[12] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*, 2000, pp. 83–94.

[13] R. Joseph, D. Brooks, and M. Martonosi, "Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs," in *Proceedings of the Workshop on Complexity Effectice Design*, 2001.

[14] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Proceedings the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 93–104.

[15] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, 2003.

[16] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction-level Parallelism*, pp. 7:1–28, 2005.

[17] P. J. Denning, "Working sets past and present," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 64–84, 1980.

[18] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, 2002, pp. 233–244.

[19] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques(PACT'01)*, 2001, pp. 3–14.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

[21] C. Isci and M. Martonosi, "Identifying program power phase behavior using power vectors," in *Proceedings of the 2003 IEEE International Workshop on Workload Characterization*, 2003, pp. 108–118.

[22] ——, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003, p. 93.

[23] Valgrind, http://valgrind.org/.