Pre-characterization Free, Efficient Power/Performance Analysis of Embedded and General Purpose Software Applications *

Venkata Syam P. Rapaka, Diana Marculescu
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
Email: {vp,dianam}@ece.cmu.edu

Abstract

This paper presents a novel approach for an efficient, yet accurate estimation technique for power consumption and performance of embedded and general purpose applications. Our approach is adaptive in nature and is based on detecting sections of code characterized by high temporal locality (also called **hotspots***) in the execution profile of the benchmark being executed on a target processor. The technique itself is* **architecture** *and* **input independent** *and can be used for both embedded, as well as for general purpose processors. We have implemented a hybrid simulation engine which can significantly shorten the simulation time by using on-the-fly profiling for critical sections of the code and by reusing this information during power/performance estimation for the rest of the code. By using this strategy, we were able to achieve up to 20X better accuracy compared to a flat, non-adaptive sampling scheme and a simulation speed-up of up to 11.84X with a maximum error of 1.03% for performance and 1.92% for total energy on a wide variety of media and general purpose applications.*

1 Introduction

Embedded or portable computer systems play an increasingly important role in today's quest for achieving true ubiquitous computing. Since power consumption and performance have a direct impact on the success of not only embedded, but also high performance processors, designers need efficient and accurate tools to evaluate the efficacy of their software and architectural innovations.

A designer can make a choice from a variety of simulators to estimate the power consumption and performance, which are at various levels of abstraction ranging from transistor- or layout-level [1] to architectural- [2, 3] and instruction-level [4, 5, 6, 7]. The lowest level simulators provide the most detailed and accurate statistics, while the higher level simulators trade off accuracy for simulation speed and portability. Although high-level simulators offer high speedup when compared to low-level simulators, they are still time consuming and may take days to simulate very large, practical benchmarks. At the same time, acceptable ranges of accuracy are subject to change when refining the design in various stages of the design cycle. Hence, it is desirable to speed-up existing simulators at various levels of abstraction without compromising on their accuracy.

In this paper, we describe our strategy for accelerating simulation speed, without significant loss in accuracy. Such a strategy has the additional benefit of being able to adapt itself to the behavior of the benchmark being simulated. Hence, it can predict the power consumption and performance statistics *without* complete detailed analysis of the execution profile of the application being executed and *without* any pre-characterization of the benchmark being simulated. Our strategy is generic and can be adopted by any simulator, at any level of abstraction, to accelerate the simulation process.

1.1 Prior Work

At software or architecture-level, various schemes and strategies have been proposed for speeding-up the power estimation process. Techniques like macromodelling [8, 9], function level power estimation [10] and energy caching [9] are some of the proposed strategies used for accelerating power simulators, while keeping the accuracy within acceptable range. Macromodelling and functional level power estimation provide speedup at the cost of time consuming pre-characterization of programs. Energy caching is based on the energy and delay characteristics of a code segment, and can be used only when these statistics are uniform across the entire execution of the code segment.

A *two-level* simulation approach has been described in [16] for estimating energy and performance with sufficient accuracy. The technique uses a flat, *fixed-window* sampling scheme, coupled with a program phase detection technique which decides on-the-fly when to use a detailed vs. a non-detailed mode of simulation. However, the approach does not adapt the *sampling window size* to the application profile to achieve better accuracy, nor does it try to detect finer-grain changes in program phase that could be exploited for better speed-up. In this paper, we introduce a *hybrid simulation engine* which is able to fully adapt to the application behavior and provide up to 20X better accuracy than the fixed-window sampling technique presented previously. Our work complements existing techniques for gate and RT-level power estimation based on sequence compaction [11] by recognizing the effect of fine and coarse grain temporal dependencies, present in common software applications.

*This research has been supported in part by NSF Career Award CCR-008479.

1.2 Paper Overview and Contributions

Our main goal is to accelerate existing simulators, by predicting power and performance values accurately. This scheme can be applied to simulators at various levels of abstraction to reduce the simulation time without compromising accuracy. To validate our strategy, as a baseline simulator, we have chosen *Wattch* [2], a framework for architectural-level power analysis and optimizations. *Wattch* has been implemented on top of *SimpleScalar* [12] tool set and is based on a suite of parameterizable power models. Based on these power models, *Wattch* can estimate power consumed by the various hardware structures based on per-cycle resource usage counts, generated through cycle-accurate simulation. *Wattch* has considerable speedup (1000X) when compared to circuit-level power estimation tools, and yet can estimate results within 10% of the results generated by Spice. But even with this speedup, it can take very long time to simulate most benchmarks of practical interest. It is thus, desirable to further reduce the simulation time without trading off accuracy.

Wattch uses per cycle statistics generated by SimpleScalar to estimate the power consumed by various components of the architecture being simulated. As in most other cycle-accurate tools, to get sufficiently accurate statistics, one must perform a *detailed* simulation of the benchmark program, at the cost of increased simulation time. In both *Wattch* and *SimpleScalar*, the program can also be executed in a *fast* mode, in which the program will be executed correctly, but without cycle accurate information.

Our strategy involves using a hybrid simulator which is capable of switching between the detailed and fast modes of simulation. The rationale for using such a simulation strategy stems from the inherent behavior of most programs of practical interest. In fact, most benchmark programs are made up of tightly coupled regions of code or *hotspots* [13], in which the program behaves in a predictable manner, by executing sections of code with high temporal locality. Once the program enters a hotspot, one can identify critical sections of the code that should be simulated in detailed mode, record per cycle information like *Instructions Per Cycle* (IPC) and *Energy Per Cycle* (EPC), and complete functional simulation of the hotspot by switching into fast mode. While this strategy has been used before in the context of a fixed-size sampling window [16], we identify the shortcomings associated with such a scheme and propose a truly application-adaptive sampling scheme with up to 20X better accuracy. Our scheme provides up to 11.8X speed-up compared to the baseline, cycle-accurate simulation engine, while keeping accuracy within 2% on average for both performance and power consumption. In addition, the proposed approach offers superior accuracy for fine-grain, per module energy estimates (less than 2% compared to up to 18% estimation error), as well as energy and performance run-time profiles that closely follow the actual, detailed profile for benchmarks under consideration.

1.3 Organization of the Paper

The rest of this paper is organized as follows: Section 2 discusses hotspots and the behavior of code inside hotspots. We describe our proposed approach for identifying program regularity and present our strategy in greater detail in Section 3. Practical considerations are discussed in Section 4. We present our experimental results and discuss them in Section 5. Section 6 concludes the paper with some final remarks.

2 Program Behavior

As it is well know, most common applications exhibit sections of code with high temporal locality. Such characteristics define the so-called *hotspots* which are collections of tightly coupled basic blocks, executing together most of the time. When an application program enters a hotspot, it executes only the basic blocks belonging to that hotspot, and only rarely steps out of this set of basic blocks. Two typical hotspots are shown in Figure 1. In this case, the code executes the basic blocks of hotspot A for a significant portion of the total execution time, before it starts executing those of hotspot B. The program maintains a very high temporal locality once it enters a hotspot, and, due to high temporal locality, it behaves in a *predictable* manner while running inside the hotspot.

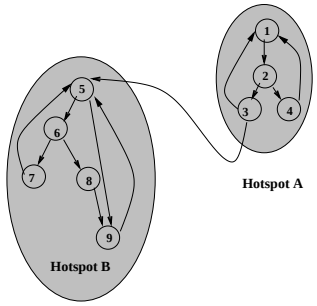


Figure 1. An example of two hotspots [16]

2.1 Hotspot Properties

Hotspots are typically tight loops or series of instructions that are executed repetitively for the entire duration of the hotspot. This repetitive behavior is reflected in how performance and power-related statistics behave. In fact, inside a hotspot, all functional units along with other hardware blocks are accessed in a specific repetitive pattern. This is true even for architectures supporting *out-of-order* execution as the dynamic schedule for the same set of instructions almost always results in the same scheduled trace of instructions inside of the given hotspot.

Hotspots create regular patterns of execution profiles during the course of execution of a benchmark. Due to these execution patterns, it is *not essential* to do a cycle accurate detailed simulation for the entire duration of each of the hotspots. Thus, one can use the IPC and EPC values of a sampling window to predict the future behavior of the program. We exploit this feature of the program execution behavior in order to accelerate micro-architectural simulation. In Section 3, we describe how metrics of interest can be obtained via sampling inside detected hotspots.

2.2 Hotspot Detection

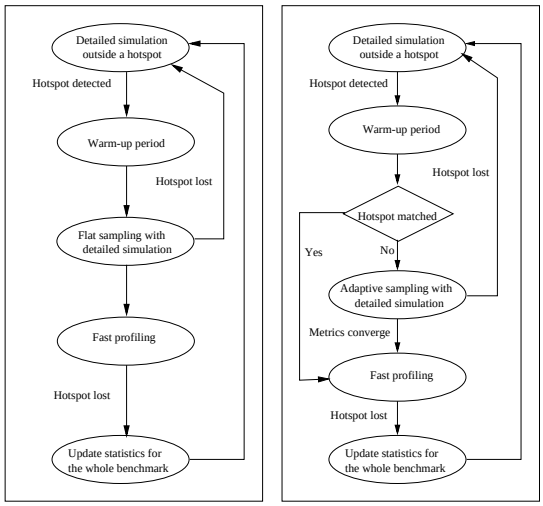
Hotspots are mainly characterized by the behavior of the branches inside the hotspot. A hotspot can be detected by keeping track of the branches being encountered during the execution of the program. To keep track of branches, we use a cache-like data structure called the Branch Behavior Buffer (*BBB*) [13]. Each branch has an entry in the *BBB*, consisting of an Execution Counter and a one-bit Candidate Flag (*CF*). The execution counter is incremented each time the branch is taken, and once the counter exceeds a certain threshold (512, in our case), the branch in question is marked as a candidate branch by setting the *CF* bit for that branch. The simulator also maintains a saturating counter called the hotspot detection counter (*HDC*) which keeps track of

candidate branches. Initially, the counter is set to a maximum value (64K in our case); each time a candidate branch is taken, the counter is decremented by a value *D*, and each time a non-candidate branch is taken, it is incremented by a value *I*. When the *HDC* decrements down to zero, we are in a hotspot. For our implementation we chose *D* as 2 and *I* as 1, such that exiting the hotspot is twice as slow as entering it.[1] The details of the hotspot detection scheme can be found in [13], [16].

3 Hybrid Simulation

As described previously, common programs exhibit high temporal locality in various sections of their code and behave in a predictable manner inside hotspots. Our strategy is to employ a two-level hybrid simulation paradigm which can perform architectural simulation at two different levels of abstraction and with different speed/accuracy characteristics.

- A *low level* simulation environment, which can perform cycle accurate simulation and provide accurate metrics associated with the program execution.
- A *high level* simulation environment, which can perform correct functional simulation without providing cycle accurate metrics.



(a) Flat scheme (b)Adaptive scheme

Figure 2. Hybrid simulation

Our chosen strategy is to achieve considerable speedup by exploiting the predictable behavior inside the hotspots. We employ the hotspot detection strategy described in [13] for determining the entry and exit points for a hotspot. We use the low-level simulation engine for the code outside a hotspot and for a fraction of the code executed inside a hotspot (also called the *sampling window*) and use high level simulation for the remainder of the hotspot. To estimate the metrics for the entire hotspot, we use the metrics acquired during the detailed simulation of the sampling window. The time spent by a program inside a hotspot is dependent on the program itself and the specific input being used, but we have observed that the average fraction of time spent inside detected hotspots is 92%. Thus, accelerating simulation of the code inside the hotspots should provide considerable speedup for the entire benchmark.

[1] This is done to prevent false exits from a hotspot.

For estimating the statistics associated to a hotspot, it is imperative to select a suitable sampling window. We will describe our proposed sampling techniques in the remainder of this section.

3.1 Flat Sampling

This scheme corresponds to the sampling strategy employed in [16] and is illustrated in more detail in Figure 2(a). Once the program enters a hotspot, a fixed window of 128K instructions is selected as the sampling window. The metrics collected in this window are used for estimating the metrics of the whole hotspot. This window is selected after skipping a warm-up window of 100K instructions.

3.2 Convergence based sampling

The flat scheme blindly chooses a fixed window size and assumes that such a window will be enough for achieving convergence for power and performance inside all hotspots, across various applications. However, such an assumption is far from being valid. To account for these different behaviors, a convergence-based sampling scheme is proposed. In such a scheme, the simulator starts off in detailed mode and switched to fast mode only upon convergence. To check for convergence, a sampling window *w* is employed. Convergence is declared only if the metrics sampled in a number of consecutive windows of *w* instructions are within a threshold of *p* (the precision for convergence). If convergence is not satisfied, the window size *w* is increased and the process is repeated.

There are two possible ways of increasing the window size *w*:

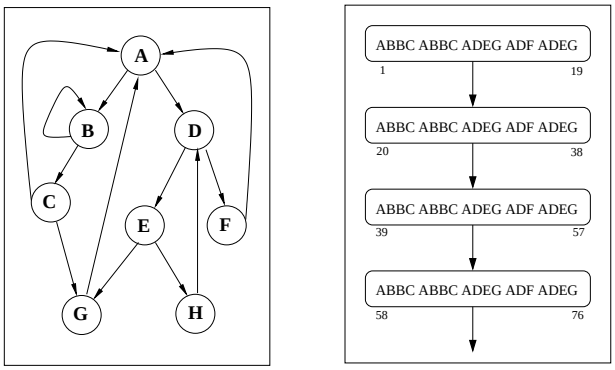
- Exponentially increasing window size. In this case, the current window size *w* is *doubled* if the convergence condition was not satisfied (possible window sizes are *w, 2w, 4w, 8w, 16w....*).
- Linearly increasing window sizes. In this case, the current window size is *incremented* by a fixed size of *w* if the convergence condition was not satisfied (possible window sizes are *w, 2w, 3w, 4w, 5w....*).

In our case, convergence is declared when metrics are maintained within a threshold *p* for 3 consecutive windows of size *w*. The exponential approach attains large window sizes in smaller number of iterations, so it starts with a smaller window size *w* of 2.5K, while the linear approach starts with 128K. Both the approaches can be used with three different threshold values for checking convergence (0.001, 0.0005 and 0.0001). The overall hybrid simulation strategy is illustrated in Figure 2(b).

As it can be expected, there is a trade-off between speed and accuracy in both cases. While an exponentially window size may suit some applications better, it may happen that the sampling window size is increasing too fast and fine-grain program phase changes may be missed. At the same time, a linearly increasing window size may prove inappropriate in cases where convergence is achieved only for large window sizes. To find a good compromise between the two, we have developed an adaptive sampling mechanism which tries to identify fine-grain program phases, also called *microhotspots*.

3.3 Adaptive Sampling

While the hotspot detection scheme makes sure that tightly coupled basic blocks are detected, it does not ensure that the sequencing information is also maintained. For example, for the



(a) Control flow graph (b) Execution trace in the hotspot

Figure 3. Microhotspot Detection

control flow graph in Figure 3(a), the set of basic blocks A, B, C, D, E, F, G is identified as being part of a hotspot. In effect, this means that the *occurrence probability* of any of these blocks is sufficiently high (related to the execution counter value in Section 2.2). However, second (or higher) order effects related to the sequencing information for these basic blocks are completely ignored. In fact, these effects determine whether a slower or faster increasing sampling window size should be used. For example, if during the execution of the hotspot, the basic blocks are executed as $((AB^nC)^mADEGADFADEG)^*$,[2] the sampling window size w should be directly related to the values of m and n. The *adaptive sampling scheme* tries to match the window size with the run-time sequencing information of the basic blocks. Such an application-adaptive window is called a *microhotspot*.

For the example shown in Figure 3(a), a possible series of candidate branches being executed in this typical hotspot is shown in Figure 3(b), where each letter represents the basic block corresponding to a candidate branch. In this example, the trace ABBCABBCADEGADFADEG represents the repeating microhotspot. We detect the microhotspot by keeping track of the most recent occurrence of every candidate branch. Whenever the difference between two consecutive occurrences of a candidate branch is larger than the current window size, a potential microhotspot is declared. The window size is changed to this new difference and the simulator checks this window for convergence. If the same branch is occurred again and the metrics of interest (EPC, IPC) have not converged yet, the window size is stretched to accommodate the new occurrence of the candidate branch. This process continues by checking for microhotspots for other candidate branches until convergence is achieved or the hotspot ends.

In practice, the simulator starts with an exponential convergence scheme with $p = 0.0001$ and $w = 2.5$K. It first tries to achieve convergence using the exponential scheme until it encounters the first potential microhotspot, defined as the set of instructions between two occurrences of a candidate branch. Then the simulator continues with microhotspot detection and stops doubling the current window size. Once metrics of interest converge, the simulator switches to the fast mode of profiling. The detailed diagram for the adaptive sampling scheme is shown in Figure 2(b).

Identifying microhotspots has not only the advantage of being able to select the right size for the sampling window, but offers additional potential for speeding-up the simulation, as described next.

[2]Classic notations from formal language theory have been used.

4 Practical Considerations

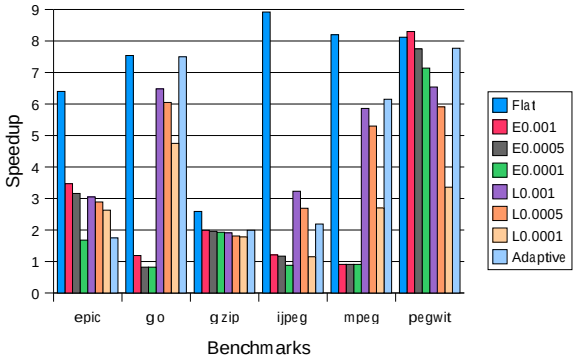
To further speed-up the simulation, we also maintain a *monitor table* to reuse the information about earlier visited hotspots. This monitor table is similar to the one described in [13], and can be used to cache information and for determining whether the current hotspot has been visited earlier. The monitor table consists of entries corresponding to a unique hotspot. Each unique hotspot is identified by a unique number (*HSP_id*) and it has it's own characteristic *signature*. This signature is made up of the top seven most frequently executed branches after the hotspot period. The signature consists of the addresses of these branches, along with their corresponding frequencies. This entry also contains the necessary information required for estimating the statistics for a matched hotspot. It includes the average IPC and EPC values for the entire hotspot, along with the average per component EPC values. These are required to accurately determine per component energy values for the entire benchmark. These IPC and EPC values are the recorded values before the simulator switches to fast mode of profiling and after adaptive sampling converges.

Whenever the program enters a hotspot, the simulator tries to match the current hotspot with one of the entries in the monitor table both in terms of branch addresses and occurrence probability. At the end of the warm-up period, the simulator stores information about the most frequent branches of the present hotspot and tries to match the current hotspot with one of the hotspots from the monitor table. This is done by comparing each of the *top five* branches of the current hotspot with the signature of each hotspot entry in the monitor table. [3]

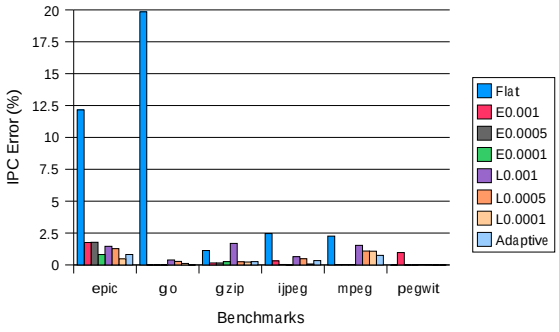
The monitor table entries are used in conjunction with the hybrid simulation mechanism shown in Figure 2(b).The simulation starts in the detailed mode and continues to monitor branch behavior. Once a hotspot is detected, the simulator tries to match the current hotspot with one of the entries of the monitor table after the initial warmup period. If the current hotspot is not matched, then the simulator tries to find an appropriate sampling window, which can be used for the estimation of various values. Once this window is determined through adaptive sampling, the simulator switches to the fast mode of profiling after recording the necessary values for estimation of power and performance. The simulator keeps track of the branch behavior and once the hotspot is lost, it reverts back to the detailed mode of simulation after updating the global power and performance metrics. If the hotspot is lost before adaptive sampling finds the required sampling window, the simulator waits until a new hotspot is detected and starts all over again. If the simulator finds a matching hotspot it directly switches over to the fast mode of profiling. The various parameters of the matching hotspot are used for updating the metrics after the current hotspot ends. If the current hotspot does not match any of the existing entries, it is added as a new entry into the monitor table after the hotspot is sampled partially or completely. This entry is created irrespective of whether the metrics had converged during adaptive sampling. Thus, hotspots which do not achieve convergence of metrics during adaptive sampling, can be simulated in fast mode if they are encountered again.

The exponential and linear convergence based sampling

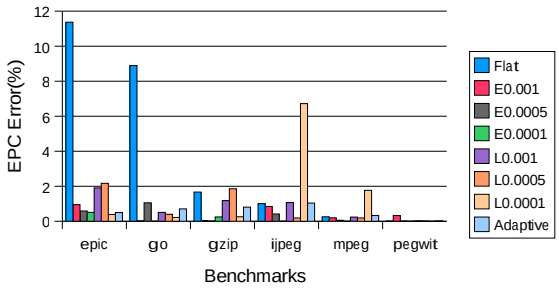
[3]We compare only the top five branches of the hotspot as we have observed that the frequencies of the candidate branches are very close in the sampling period and the least frequent two branches may change with different execution contexts of the hotspot.



(a) Speedup

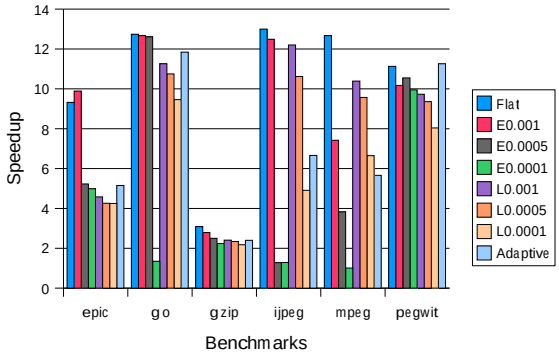


(b) IPC Error

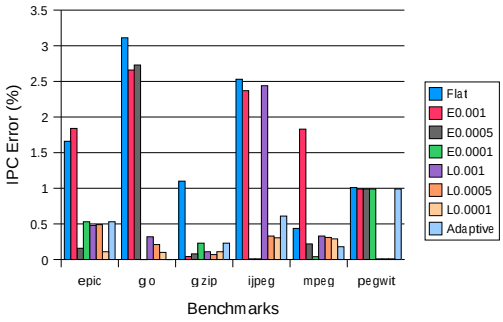


(c) EPC Error

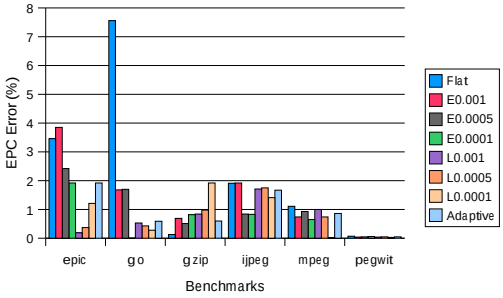
Figure 4. Results for 4-way, out-of-order processors



(a) Speedup



(b) IPC Error



(c) EPC Error

Figure 5. Results for 1-way, in-order processors

schemes are similar and only differ in the way they check for convergence of metrics.

5 Experimental Results

We have tested our hybrid simulation strategy on various benchmarks for both single instruction issue, in-order processors and for 4-way out-of-order superscalar processor. We have compared the performance of the four schemes of hybrid simulation described in this paper (flat sampling, linear and exponential window sampling, and adaptive sampling with a monitor table). Both configurations for the in-order, single instruction issue and 4-way, out-of-order superscalar processors assume a 512K direct mapped I-cache and a 1024K 4-way set-associative D-cache. In case of the superscalar machine, the width of the pipeline and the number of ALUs is assumed to be four. In addition, a 32-entry Register Update Unit (RUU) and a 16-entry Load Store Queue (LSQ) are considered in this case. In both cases, we have used benchmarks from SpecInt95 (*ijpeg*, *go*), SpecInt2000 (*gzip*) and MediaBench (*mpeg*, *pegwit*, and *epic*).

To assess the viability of our proposed hybrid simulation approach, the following set of experiments have been considered:

- The speed-up and accuracy for energy (EPC) and performance (IPC) metrics for the set of benchmarks under consideration.
- A comparative, detailed study of the flat and adaptive sampling schemes in terms of per module accuracy.
- A comparative study of all sampling schemes in terms of predicting the run-time profile of a given benchmark.

We show in Figures 4 and 5 our results for the accuracy and speed-up achieved for all sampling schemes proposed. The linear and exponential convergence schemes have been considered for three different precision thresholds: 0.001, 0.0005, 0.0001 (E0.001, E0.0005, E0.0001 for exponential and similar for the linear case). As it can be seen, the flat sampling case offers the highest speed-up, but at the expense of a very large error in some cases (e.g., 20% for IPC estimation in case of *go* or 11% error for EPC in case of *epic* when run on a 4-way, superscalar processor). While

Component	*go* flat	*go* adp	*epic* flat	*epic* adp
Rename	N/A	N/A	12.40%	0.64%
Bpred	18.25%	0.00%	6.05%	0.31%
Window	N/A	N/A	12.34%	0.81%
LSQ	N/A	N/A	7.91%	0.44%
Regfile	6.93%	0.00%	8.56%	0.58%
Icache	2.75%	0.00%	9.39%	0.53%
Dcache	3.21%	0.00%	11.58%	0.49%
Dcache2	2.36%	0.00%	2.44%	0.07%
ALU	0.05%	0.00%	7.63%	0.50%
Resultbus	9.80%	0.00%	12.00%	0.83%
Clock	18.25%	1.52%	14.25%	0.41%

Table 1. Per Component Power Errors

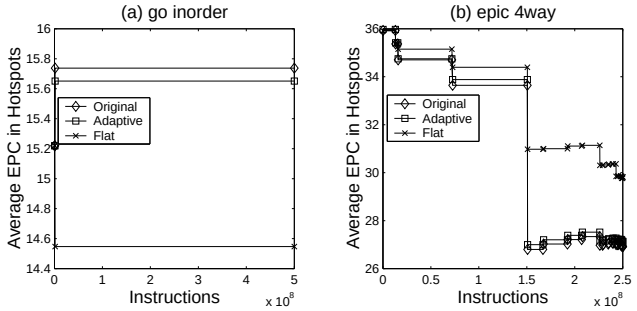


Figure 6. Comparison of the two schemes

the exponential and linear convergence cases outperform the adaptive sampling scheme in some cases in terms of accuracy, they have unpredictable behavior, depending on the benchmark. The adaptive sampling scheme is the only one that offers consistent low error (less than 2% for both energy and performance estimates), irrespective of the type of processor being considered. The same cannot be said about the linear or exponential sampling schemes, showing their lack of adaptability to the application being run.

For the worst case achieved in the case of flat sampling scheme (i.e., benchmark *go* in case of single issue, in-order processors and *epic* in case of 4-way superscalar processors), we also show in Table 1 the estimation error for the energy cost per module. As it can be seen, adaptive sampling case (denoted by adp in Table 1) consistently produces results that are less than 2% away from the exact, cycle-accurate values, while the flat sampling (denoted by flat in Table 1) scheme can generate up to 18.25% error in some cases.

Finally, we have analyzed how the sampling schemes proposed in this paper track the actual run-time profile of the application being run. We show in Figure 6 the estimated EPC values for the adaptive and flat sampling cases, compared to the original results provided by Wattch. As it can be seen, the adaptive case preserves the actual run-time profile much better (within 2% of the original), while the flat sampling results can be off by as much as 12%.

6 Conclusion

In this paper we have presented a hybrid simulation strategy, which can accelerate the microarchitectural simulation without trading off accuracy. The strategy is based on the detection of hotspots and determining the exact sampling window size, which can be used for estimation of various metrics. We also presented how these features can be exploited for simulation acceleration along with using a monitor table for reusing know information. By using these techniques we have been able to obtain substantial speedup, with negligible errors in IPC and EPC values. The proposed adaptive sampling scheme not only offers superior accuracy when compared to a simpler, flat sampling scheme, but also provides per module estimation error of less than 2% and faithfully tracks the run-time profile of the application under consideration.

References

[1] C. X. Huang, B. Zhang, A. C. Deng and B. Swirski, 'The Design and Implementation of Powermill,' in *Proc. Intl. Workshop on Low Power Design*, pp. 105-110, April 1995.

[2] D. Brooks, V. Tiwari and M. Martonosi, 'Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,' in *Proc. Intl. Symp. Computer Architecture*, pp. 83-94, Vancouver, BC, Canada, June 2000.

[3] W. Ye, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, 'The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool' in *Proc. ACM/IEEE Design Automation Conference*, Los Angeles, CA, USA, June 2000.

[4] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, 'An instruction-level functionality-based energy estimation model for 32-bit microprocessors,' in *Proc. Design Automation Conf.*, pp. 346-351, June 2000.

[5] J. Russell and M. Jacome, 'Software power estimation and optimization for high-performance 32-bit embedded processors,' in *Proc. Int. Conf. Computer Design*, pages 328-333, October 1998.

[6] A. Sama, M. Balakrishnan and J. F. M. Theeuwen, 'Speeding up Power Estimation of Embedded Software,' in *Proc. Int. Symp. Low Power Electronics and Design*, Rapallo, Italy, 2000.

[7] V. Tiwari, S. Malik and A. Wolfe, 'Power analyasis of embedded software: A first step towards software power minimization,' in *IEEE Tran. VLSI Systems*, 2(4):437-445, December 1994.

[8] T. K. Tan, A. Raghunathan, G. Lakshminarayana and N. K. Jha, 'High-level Software Energy Macro-modelling,' in *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, USA, June 2001.

[9] M. Lajolo, A. Raghunathan and S. Dey, 'Efficient Power Co-Estimation Techniques for System-on-Chip Design,' in *Proc. Design & Test Europe*, pp.27-34, March 2000.

[10] G. Qu, N. Kawabe, K. Usami and M. Potkonjak, 'Function-Level Power Estimation Methodology for Microprocessors,' in *Proc. ACM/IEEE Design Automation Conference*, Los Angeles, CA, USA, June 2000.

[11] R. Marculescu, D. Marculescu and M. Pedram, 'Sequence Compaction for Power Estimation: Theory and Practice,' in *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):973-993, July 1999.

[12] D. Burger and T. M. Austin, 'The Simplescalar Tool Set, Version 2.0,' in *Computer Architecture News*, pp. 13-25, June 1997.

[13] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu 'A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization,' in *Proc. Int. Symp. Computer Architecture*, pp. 136-147, May, 1999.

[14] M. Sami, D. Sciuto, C. Silvano and V. Zaccaria, 'Instruction -level power estimation for embedded VLIW cores,' in *Proc. Intl. Wrkshp. Hardware/Software Codesign*, pp. 34-37, March 2000.

[15] A. Sinha and A. P. Chandrakasan, 'JouleTrack - A Web Based Tool for Software Energy Profiling,' in *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, USA, June 2001.

[16] D. Marculescu and A. Iyer, 'Application-Driven Processor Design Exploration for Power-Performance Trade-off Analysis,' in *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, San Jose, USA, Nov. 2001.