Function-Level Power Estimation Methodology for Microprocessors

Gang Qu†, Naoyuki Kawabe‡, Kimiyoshi Usami‡, and Miodrag Potkonjak†
† Computer Science Department, University of California, Los Angeles, CA 90095, USA
‡ Design Methodology Dept., Semiconductor Company, Toshiba Corporation, Kawasaki, 210, Japan

Abstract

We have developed a function-level power estimation methodology for predicting the power dissipation of embedded software. For a given microprocessor core, we empirically build the "power data bank", which stores the power information of the built-in library functions and basic instructions. To estimate the average power of an embedded software on this core, we first get the execution information of the target software from program profiling/tracing tools. Then we evaluate the total energy consumption and execution time based on the "power data bank", and take their ratio as the average power. High efficiency is achieved because no power simulator is used once the "power data bank" is built. We apply this method to a commercial microprocessor core and get power estimates with an average error of 3%. With this method, microprocessor vendors can provide users the "power data bank" without releasing details of the core to help users get early power estimates and eventually guide power optimization.

1 Introduction

With the emergence of wireless and battery-operated devices, low-power design is becoming increasingly popular. To meet the low-power requirement for such designs, researchers have put intensive efforts on power estimation, modeling, and optimization in the past decade. As a result, numerous techniques have been proposed and tools been implemented to help system designers get an early view of the system's power behavior (see [8, 11] for a survey).

Although it is generally accepted that the processor's power consumption highly depends on the program to be executed, most of these efforts are on the hardware side and little attention has been paid on software. Models at the software level are difficult to build because of the great variety and complexity of the application programs, whose exact execution behavior is hard to predict until the underlying hardware configuration is determined. Tiwari et al. [14] conduct the pioneer work on this front. In their proposed instruction-level power model, each instruction and instruction pair are assigned a fixed (base) energy cost and the sum is taken as the program's total energy consumption. This also leads to several software-level power minimization techniques [6, 14].

In this paper, we first present a function-level power model which is set up by the construction of "power data bank" for a given microprocessor core. For each built-in library function and instruction, we use a power simulator to empirically collect its power and execution time information while considering the effects of cache misses and pipeline stalls. Such information is stored in the "power data bank". We then develop a power estimation technique built on top of this model to predict the power consumption of embedded software. We get the program's execution information from program profiling and tracing tools, which includes for example how many times each function is executed, cache miss rate and pipeline stalls. The average power is estimated based on the "power data bank", without using the time consuming power simulator again.

Key Contributions

Both microprocessor vendors and users benefit from our function-level power model:

- Vendors can packet the "power data bank" with their microprocessor and provide users the ability to conduct power estimation and optimization of their embedded software. More importantly, vendors need not to release the details of their core, such as the RTL or gate-level netlist, for this purpose.
- Vendors can efficiently build the "power data bank", which is restricted to the built-in library functions and instructions.
- Vendors can build the "power data bank" with power simulators at any level. This enables users to get highly accurate power estimation without any degradation of power estimation's efficiency.
- Vendors can further make power estimation process fully automated with the help of program profiling and tracing tools.
- Users will enjoy higher efficiency to estimate their programs' power on the given microprocessor. This is because functions may consist of tens to hundreds of basic instructions. Potentially, there could be a speedup of one or two orders of magnitude over the instruction-level tools.
- Users are also assured of higher accuracy due to the fact that function-level model captures the inter-instruction effects of a sequence of instructions.

2 Related Work

Power estimation and modeling attract a lot of attention as power becomes one of the critical constraints for system design. Extensive research efforts have been put to develop efficient and accurate algorithms and tools at all levels of the design process, from circuit level [4], gate level [9, 10], RT level [7, 3], to system level [13, 12]. Power is measured directly or by means of circuit activities like effective capacitances [7] and average currents[1, 6, 14]. Most of them are simulation oriented, in which the system's power behavior is monitored during the simulation on the input vectors. Statistical , regression-based , and information-theoretic techniques have been proposed to reduce simulation time without sacrificing much accuracy. In general, as the simulation platform moves from low level to high level, the tools become more efficient but less accurate.

Program profiling and tracing tools are widely used in the analysis, design, and tuning of hardware and software systems. A profiler counts the occurrences of an event during a program's execution. Such events can be the beginnings of new paths, in which case the profile counts the number of times each path executes; or hardware events such as data cache misses, in which case the profile counts the number of times each path suffered a cache miss. However, the data references are ignored. In contrast, a program tracer provides a complete record of instructions executed and data reference [2, 5].

DAC 2000, Los Angeles, California

As the first successful effort on power estimation and optimization from software side, Tiwari et al. [14] and Lee et al. [6] develop the instruction-level power modeling technique, in which the energy cost for each instruction and each instruction pair is empirically constructed and the sum is taken as the program's total energy consumption. One problem for this approach is the size of instruction set. A modern microprocessor core like the MIPS R3000A based Toshiba TX39 supports about 100 basic instructions. It takes tremendous amount of work to determine the energy cost for all the instruction pairs. The authors also notice that the cost for two instructions is usually different from their average and they propose to measure the overhead for each instruction pair. It is obvious that, for the same reasons, the accuracy can be improved by measuring three or more consecutive instructions at the expense of increasing the cost for model building.

3 Function-Level Power Modeling and Estimation

3.1 Power Measurement

Average power is the energy consumption per unit time and we attempt to model power directly from this fundamental formula. Suppose during the program's execution, we observe an execution time T and core's energy consumption E, then the average power is given by $P = \frac{E}{T}$.

However, the challenge is how to measure E and T. It is expensive and impractical to apply the simulation based tools. Such tools are either time consuming (at low level) or lack accuracy (at high level). The instruction-level power model [14] relies on the energy cost of each instruction and instruction pair. A typical software package may involve tens of thousands of instructions, including jump/branch instructions that make it very difficult to apply the instruction-level power model [14].

Through extensive experiments, we observe that majority of the machine code is for the implementation of the library functions and user-defined procedures, linked by a small portion of codes in the main program itself. For example, there are 16,349 instructions in the compiled MPEG2 Decode code (source codes obtained from http://www.mpeg.org/MSSG), only 161 (less than 1%) are for the main body. In particular, we identify four key energy/time consumers during the program's execution: built-in library function calls, user-defined functions calls, the main body, and others (e.g., the hardware reset program). This leads to the following power formulation:

$$P = \frac{E}{T} = \frac{\sum_i E_{BF_i} + \sum_j E_{UF_j} + E_{main} + E_{others}}{\sum_i T_{BF_i} + \sum_j T_{UF_j} + T_{main} + T_{others}} \quad (1)$$

where for example, E_{BF_i} is the energy consumed by library function i, T_{UF_j} is the time used to execute user-defined function j, and the sums are taken over all library and user-defined functions.

3.2 "Power Data Bank" Construction

One may view the application programs as a sequence of basic blocks. Such blocks can be the instruction set, built-in library functions, user-defined procedures, etc. At the lowest level, we can take the basic blocks as the instruction set supported by the microprocessor core. However, this does not capture the inter-instruction effects. On the other hand, user-defined functions differ from program to program and usually are not available until the software package is known. Therefore, we propose to measure the power data for each library function and basic instruction.

"Power data bank" consists of the power information, such as energy consumption and estimated execution time, for library functions and basic instructions. These data are obtained from power simulations on a set of well-designed test codes. The power simulator can be at any level from transistor level to RT level. A low-level power simulator is able to provide high accurate data with long simulation time. The vendor has the freedom to choose power simulator to achieve desired level of accuracy.

"Power data bank" contains multiple entries for each function and instruction. We know that I-cache miss, D-cache miss, or pipeline stalls will cause different power behavior for a function. When construct the "power data bank", we take into account these factors by repeatedly measuring under different circumstances.

3.3 Function-Level Power Modeling

We describe how to evaluate each term in Equation (1) by a power simulator.

Built-in library functions

A built-in library function consists of a sequence of basic instructions from the core's instruction set. However, its execution is not necessarily sequential because of the jump/branch instructions. These jumps and branches in general cannot be determined until the run time. Taking all the possibilities into consideration will make it hard to build the "power data bank". We observe that these jump/branch instructions usually jump over a small portion of, and never go outside of, the instruction sequence.

Therefore we measure each function's power behavior for a number of times, extract the statistical information and store them in the "power data bank" while factors like cache misses and pipeline stalls are considered. For instance, both the energy consumption and execution time of a function without cache misses can be very different from the case when a cache miss occurs. The cache miss rate usually depends on the cache specifications (cache size, associativities, replace algorithm, cache line size, etc.). We can use the well-documented cache memory design techniques to make cache misses exactly predictable, and thus their impact measurable.

Many reasons can cause cache misses and pipeline stalls. Instead of treating these effects explicitly and separately[6, 14], we embed them into the power data for each library function and basic instruction. In particular, we provide data of different level of precision in the "power data bank". Accurate data can be used if sufficient information from the program profiler/tracer is available. To get such more detailed information about the execution, the profiler/tracer needs additional specification and this again leads to the trade-off between efficiency and accuracy.

User-defined function

It is possible to decompose a user-defined function/procedure to a combination of library functions and basic instructions. Then we can use the associated data from "power data bank" to approximate the user-defined procedure. The problem for this approach is that the error for each individual procedure may be small, but the accumulated effect could have a large impact on the final result. For example, in the case when a procedure makes many calls to other user-defines function calls. We suggest to measure each user-defined function, at least those being frequently used, separately based on the structure of the function and the "power data bank". For a heavily used procedure, we calculate its power data while paying special attention, for example, all the basic instructions being used will be considered, and the most detailed power data for the library functions from the "power data bank" will be used. This complementary method considers the locality of the user-defined function and thus makes a more accurate estimation.

main()

The *main()* part of the program that we refer here is the remainder of the program after excluding all the library and user-defined function implementations. This part of the machine code is usually separated by the library and user-defined function calls. As we discussed earlier, it contributes very little to the total energy consumption and execution time. However, if the *main()* part is not negligible, (in cases like intensive computation involving only the instruction set, which is performed inside *main()*), then we treat it

in as a user-defined function and measure its power behavior directly from the "power data bank".

Others

Besides the above three key components, there exist other parts that support the program's execution and consume power as well. For instance, the boot program, which initializes the registers, assigns addresses for the user program and stack pointer, and sets flags upon the program's termination; instructions that save the information when the program starts and restore them when *main()* is completed. In our experience, these depend on the execution of the program, but the variations are so small that we can model them as constants.

3.4 Function-Level Power Estimation

Figure 1 shows the global flow of power estimation at function-level for a given microprocessor core.

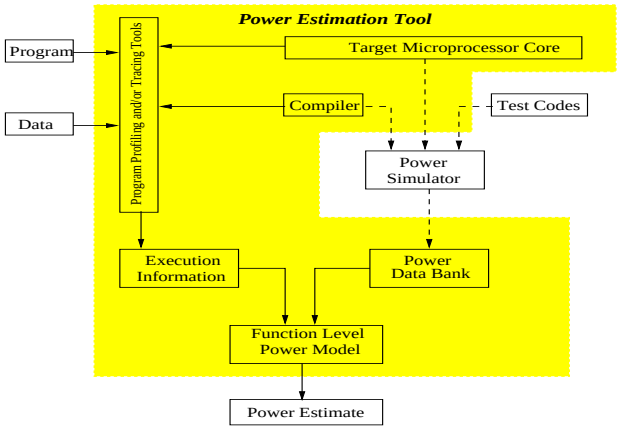

Figure 1: Global flow for the function-level power estimation.

The components connected by the dotted edges depict the procedure of building the "power data bank". This will be executed only once for a fixed hardware configuration (microprocessor core, I-cache and D-cache set up, hardware reset program, etc.) without any information about the application programs to be executed on the core. The power estimation tool is shown in the shaded area. It takes the user's program and test data as input, and predicts the power behavior of the execution of such program on the given microprocessor core. Figure 2 illustrates the three major steps of this estimation process.

Notice that the time consuming power simulator is not involved in the power estimation tool, so this method conceptually outperforms all the simulation-based approaches in terms of efficiency. This estimation technique is also more efficient than the instruction-level power model for collecting power data and evaluating a program's power for obvious reasons. The accuracy has been demonstrated by extensive experiments and part of the results are reported in the next section.

4 Experimental Results

4.1 Toshiba TX39 Microprocessor Core

The TX39 processor core is a high-performance 32-bit microprocessor core based on the R3000A RISC microprocessor. Toshiba develops application specific standard products using the TX39 core and provides the TX39 as a processor core in embedded array or cell-based ICs. Figure 3 shows the block diagram of the TX39 processor core, which includes the CPU core, an instruction cache and a data cache. The CPU core comprises CPU registers, CP0 registers, the computational unit ALU/Shifter, the unit MAC for multiply/add, memory management unit, and bus interface unit.

Input: a program with input data
Output: power estimate for the execution of the input program on a specific microprocessor core.

Phase I: preparation
- compile the source program
  - denote the built-in library functions by $BF_1, BF_2, \cdots, BF_n$;
  - denote the user-defined functions by $UF_1, UF_2, \cdots, UF_m$;
- build the function-denpendency graph G
  - for each function UF_i, introduce a new node v_i;
  - for each node v_i
    for each function UF_j being called by UF_i
    add a direct edge from v_j to v_i;
- find a topological order of G: $\{v'_1, v'_2, \cdots, v'_m\}$;

Phase II: program tracing
- run the program profiler and tracer;
- collect the execution information;

Phase III: power estimation
- for $i = 1 \cdots m$
  calculate the power data for node v'_i from the "power data bank" and the power data of nodes $v'_1, v'_2, \cdots, v'_{i-1}$;
- return power data of node v'_m;

Figure 2: Pseudo-code for the function-level power estimation.

4.2 Experimental Platform

We illustrate how the "power data bank" is built for the Toshiba TX39 microprocessor core through a cross-architecture compiler for the MIPS microprocessor architecture from the Green Hills Software, Inc.(http://www.ghs.com). We use power simulation and analysis tools provided by Senté, Inc.(http://www.senteinc.com).

We have developed, as the first step, a set of C/C++ codes each targeting one or more specific built-in library functions of the microprocessor core. For example, the function that performs integer comparison will be frequently used in any integer sorting program.

The source codes are then compiled by the cross-architecture compiler and all the library functions that might be used during the execution can be easily identified from the symbol table at compilation time. However, not all such functions will be actually called at the run time. Whether a function will be used or not, and how many times, depend on the real input data. We estimate this runtime information by tracing the software's execution using program profiling/tracing tools.

Next is the key procedure to build the "power data bank". Our goal is to get the power data (e.g. the average power and execution time) for each library function. This requires effective and efficient methodologies to monitor the execution of such functions. In practice, simulated execution time is obtained by inserting breakpoints at the beginning and the end of each library function while running the simulator. Then we use Senté's Watt Watcher to conduct the power analysis for the particular function by specifying the start/finish time and the average power consumption is gathered.

Finally, the previous three steps are repeated and the statistical power information for each library function is stored in the "power data bank".

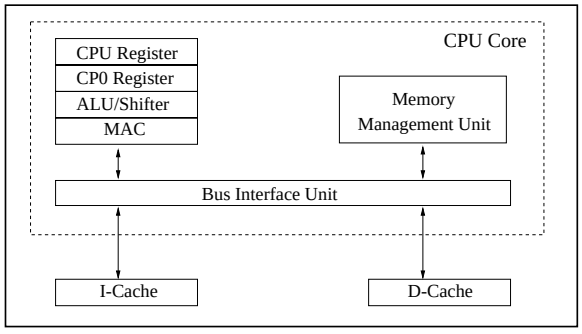

Figure 3: Block Diagram of the TX39 Processor Core[15].

4.3 Model Validation

For a test program, we conduct full power simulation to get its power consumption; we also trace the program's execution and calculate its power from the function-level power model. These two power estimates are then compared for the model validation.

test1	program with 200 floating-point additions
test2	program with 200 floating-point subtractions
test3	program with 200 mixed integer/floating-point additions
test4	program with 200 mixed integer/floating-point subtractions
test5	program with 80 integer/floating-point additions/subtractions
test6	program with 196 integer/floating-point additions/subtractions

Table 1: Description of the test programs.

Table 1 describes six test codes. They have called six library functions: **fcmp** (comparison of two floating-points), **dcmp** (comparison of two integers), **ftod** (convert floating-point to double), **fadd** (addition of two floating-points), **fsub** (subtraction of two floating-points), and **itof** (convert integer to floating-point).

We first use power simulator to simulate the full execution of each program and the obtained average power and execution time are used as reference. Then we trace the execution of the program and collect the number of times each function has been called, and the number that cache miss occurs. These results are shown in Table 2, where the first call of each function is considered as an I-cache miss, and the rest as I-cache hits due to the fact that the 4KB I-cache is large enough to hold the instructions for these test programs. Now, for each of the function blocks being used, we fetch its power information from the pre-calculated "power data bank", which includes the simulated energy consumption and execution time (in simulator ticks) per execution, and apply our model to get the average power and execution time as reported in the last two rows in Table 2.

	test1	test2	test3	test4	test5	test6
fcmp	/	200	100	200	/	98
dcmp	201	1	101	1	1	1
ftod	201	1	101	1	1	1
fadd	200	/	200	/	80	158
fsub	/	200	/	200	/	38
itof	/	/	100	100	40	98
P	94.65%	94.48%	96.82%	97.71%	100.36%	98.76%
T	88.99%	85.07%	85.28%	86.24%	72.28%	82.94%

Table 2: Numbers of library function calls, and power estimates. (P and T are the program's average power consumption and estimated execution time. These values are calculated from the power model and have been normalized by the respective results from full simulation.)

A high power accuracy (an error within 3.5% in average)is achieved. We can further improve this by fine tuning the parameters. The error on simulated execution time comes from the ignored part of the program, which includes most of the instructions in *main()* that connect these built-in library function blocks and others. If we model this part and add its effect into the execution time, we are able to restrict the error to within 4%.

We further validate our assumption on the I-cache miss/hit by simulating the single execution of each individual function. The results demonstrate that, for each function block, the first execution takes long while all the rest run much faster and roughly at the same speed. Moreover, the power data for the first execution coincide with those when we disable the I-cache using the cache test function.

In [14], the impact of the different input data has been studied in terms of the numbers of 1's in the data's binary representation, and they conclude that such impact is small (less than 5%). Under our experimental platform, it is hard to observe such phenomena because estimation is at function-level instead of instruction-level, the impact of input data to a single instruction is shadowed by the relatively long execution time of the entire instruction sequence, the divisor when evaluating average power.

5 Conclusions and Future Work

Power is one of the concerns when users select hardware to perform their programs. For the microprocessor core vendors, it helps if they can provide users tools to estimate the program's power consumption on the target core. However, the power estimation usually requires a detail description of the core, which most vendors are reluctant to release. We propose a power model and estimation methodology at function level to bridge this gap. The efficiency and accuracy of this method have been discussed and validated on a commercial microprocessor core. Future work includes fine-tuning the power model, extending it to DSPs, superscalar processors, and our ultimate goal is to use this to provide guidance for power optimization of embedded software at compilation time.

References

[1] D.T. Blaauw, A. Dharchoudhury, R. Panda, S. Sirichotiyakul, C. Oh, and T. Edwards. *Emerging power management tools for processor design.* International Symposium on Low Power Electronics and Design, pp. 143-148, 1998.

[2] B. Cmelik and D. Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling.* SIGMETRIC'94, 1994.

[3] C.T. Hsieh, Q. Wu, C.S. Ding, and M. Pedram. *Statistical Sampling and Regression Analysis for RT-Level Power Evaluation.* ICCAD'96, pp. 583-588, 1996.

[4] C.X. Huang, B. Zhang, An-Chang Deng, and B. Swirski. *The design and implementation of PowerMill.* Proceedings. 1995 International Symposium on Low Power Design, pp. 105-108, 1995.

[5] J.R. Larus. *Efficient Program Tracing.* IEEE Computer, Vol. 26, No. 5, pp. 52-61, May 1993.

[6] M. T.-C. Lee, V. Tiwari, S. malik, and M. Fujita. *Power Analysis and Minimization Techniques for Embedded DSP Software.* IEEE Transactions of Very Large Scale Integration Systems, Vol.5, No.1, pp. 123-135, 1997.

[7] D. Liu and C. Svensson. *Power Consumption Estimation in CMOS VLSI Chips.* IEEE Journal of Solid-State Circuits, Vol.29, No.6, pp. 663-670, 1994.

[8] E. Macii, M. Pedram, and F. Somenzi. *High-Level Power Modeling, Estimation, and Optimization.* IEEE Transactions on Computer-Aided Desigh of Integrated Circuits and Systems, Vol.17, No.11, pp. 1061-1079, 1998.

[9] R. Marculescu, D. Marculescu, and M. Pedram. *Adaptive Models for Input Data Compaction for Power Simulations.* Proceedings of ASP-DAC'97, pp. 391-396, 1997.

[10] F.N. Najm. *Transition Density: A New Measure of Activity in Digital Circuits.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.14, No. 1, pp. 310-323, 1993.

[11] F.N. Najm. *A Survey of Power Estimation Techniques in VLSI Circuits.* IEEE Transactions of Very Large Scale Integration Systems, Vol.2, No.4, pp. 446-455, 1994.

[12] J.M. Rabaey. *Exploring the Power Dimension.* IEEE Custom Integrated Circuits Conference, pp. 215-220, 1996.

[13] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago. *Evaluation of Architecture-Level Power Estimation for CMOS RISC Processors.* ISLPE'95, pp. 44-45, 1995.

[14] V. Tiwari, S. Malik, and A. Wolfe. *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization.* IEEE Transactions of Very Large Scale Integration Systems, Vol.2, No.4, pp. 437-445, 1994.

[15] 32-Bit TX System RISC TX39 Family Architecture, Toshiba Corp. 1999