Journal of Circuits, Systems, and Computers, Vol. 11, No. 5 (2002) 1–18

CODE COVERAGE-BASED POWER ESTIMATION TECHNIQUES FOR MICROPROCESSORS

GANG QU

Electrical and Computer Engineering Department and Institute of Advanced Computer Study, University of Maryland, College Park, Maryland, 20742, USA

NAOYUKI KAWABE and KIMIYOSHI USAMI

Design Methodology Department II, System LSI Design Division, Toshiba Corporation Semiconductor Company, 580-1, Horikawa-cho, Saiwai-ku, Kawasaki, 210, Japan

MIODRAG POTKONJAK

Computer Science Department, University of California, Los Angeles, California 90095, USA

Received
Revised
Accepted

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 decompose the machine code into library functions and user-defined functions. We then use program profiling/tracing tools to get the execution information of the target software. Next, we evaluate the total energy consumption and execution time based on the "power data bank", and their ratio is taken 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%. Using 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.

Keywords:

1. Introduction

With the emergence of wireless and battery-operated devices, low-power design is becoming increasingly important in the system design process. To meet the low-power requirement for such designs, researchers have put intensive efforts on power estimation, modeling, and optimization. As a result, numerous techniques have been proposed and tools been implemented in the past decade to help system designers

get an early view of the system's power behavior.[1,2] These power estimates will provide system designers guidelines to apply appropriate power minimization techniques to ensure that system's power budget is met. For instance, power has been measured as proportional to the product of effective capacitance, clock frequency, and the square of supply voltage. Consequently, various methods have been developed to reduce the effective capacitance,[3–5] slow down the system clock,[6,7] and/or reduce the supply voltage.[8–11]

Although it is well-understood 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 application programs, whose exact execution behavior is hard to predict until the underlying hardware configuration is determined. Tiwari *et al.*[12] conducted 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.[12,13] More recently, Lajolo *et al.* presented a power estimation framework for hardware/software system designs.[14]

In this paper, we present a function-level power model and a power estimation technique built on top of this model, to predict the power consumption of embedded software on any specific microprocessor core at the compilation time. The basic idea is to cover the machine code by built-in library functions and user-defined functions and thus view the source code as a sequence of function calls. (i) 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". (ii) To estimate the power consumption of an embedded software, we use program profiling and tracing tools to gather the program's execution information such as how many times each function is executed, cache miss rate, pipeline stalls and etc. The average power is estimated based on the "power data bank", without using the time consuming power simulator again.

1.1. *Key contributions*

We develop a new methodology for power estimation from software side at the function level, 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 at compilation time. More importantly, vendors need not 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" because only the power information for the built-in library functions and instructions will be collected by simulation.

- 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.

1.2. *Paper organization*

The rest of the paper is organized as follows. In the next section, we briefly survey the previous work on power estimation and program profiling/tracing. In Sec. 3, we lay out the foundation of our function-level power model and the power estimation method; both have been verified by applying to a commercial microprocessor core. The detailed experimental platform and validation results are reported in Sec. 4. We conclude by a summary of our current and ongoing works.

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,[15] gate level,[16,17] RT level,[18,19] to system level.[20–23] Power is measured directly or by means of circuit activities like effective capacitances[18] and average currents.[12,13,24] Most of them are simulation oriented, in which the system's power behavior is monitored during the simulation on the input vectors. Statistical,[16,25] regression-based,[19,26] and information-theoretic[27,28] 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. These tools can provide detailed information about the behavior of a program that can be used to analyze and predict the behavior of a particular system component. A number of widely used architectural and operating systems profiling tools have been develop, including one reported.[29–31] A profiler counts the occurrences of an event during a program's execution. Such events can be the beginning 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 suffers a cache miss. However, program profilers ignore the data references. In contrast,

a program tracer provides a complete record of instructions executed and data reference.[31,32] Comprehensive tracers have been often used for both optimization[33] and validation.[34] Recently, new domains such as the Internet and wireless networks have also been analyzed using trace and profiling techniques.[35–38]

The most relevant work on power modeling and estimation includes the profile-driven program synthesis techniques by Hsieh *et al.*[39] and the instruction-level power modeling technique by *Tiwari et al.*[12] We now detail these two approaches and explain the difference between them and our approach. Hsieh *et al.* propose the profile-driven program synthesis technique for power estimation at RT-level. In this approach, the characteristic profile of the application programs is first extracted by instruction tracing using an architectural simulator, then a new program is synthesized with the same performance and power behavior, but smaller size. The synthesized program will be simulated on a RT-level description of the target microprocessor to provide the power value. In their experimentation, the synthesized programs are smaller than the original programs by $3 \sim 5$ order of magnitude. However, these synthesized programs give the average energy per instruction with an error of only 2.9%. We conclude that this approach requires power simulation, albeit on the much smaller synthesized programs, and therefore still belongs to the category of simulation based methods.

As the first successful effort on power estimation and optimization from software side, Tiwari *et al.*[12] and Lee *et al.*[13] developed 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. To be more specific, the total energy dissipation of a program is divided into three parts:

(i) Base instruction energy, which is measured by the product of each instruction's energy dissipation and the number of times this instruction has been executed.
(ii) Circuit state overhead, which captures the additional energy cost of a pair of instructions over the sum of the base cost of the pair. This component is again evaluated as the product of each instruction pair's power overhead and the number of times this pair has been executed.
(iii) Other energy dissipation caused by cache misses and/or pipeline stalls during the program execution.

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 *empirically determine* the basic energy cost for all the instructions and instruction pairs.

3. Function-Level Power Modeling and Estimation

In this section, we first explain how we measure the (average) power and the idea of "power data bank". Then we propose the function-level power model with detailed

explanation on how to construct the "power data bank". Finally we discuss the code coverage-based power estimation technique.

3.1. *Power measurement*

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

Clearly the challenge is how to measure T and E accurately and efficiently. It will be expensive and impractical to use the simulation based tools. Such tools are either time consuming (at low level) or lack accuracy (at high level). The profile-driven program synthesis method[39] relies on how accurate the instruction tracing tools can extract the program's run-time characteristics and how similar (in terms of performance and power behavior) the smaller synthesized sample program is to the original program. The instruction-level power model requires the base 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.[12]

Through extensive experiments, we observe that the machine code mainly consists of blocks of basic instructions, which implement library functions and user-defined procedures, linked by a small portion of codes in the main program itself. For instance, there are 16 349 instructions in the compiled MPEG2 Decode code (source codes obtained from Ref. 40), only 161 of them (less than 1%) are in the *main()* body. This suggests us to measure the behavior of such functions (both library and user-defined) instead of the basic instruction set for power estimation. In the instruction-level power estimation, the authors have noticed that, the power cost of two instructions is usually different from the average of power costs of the two individual instructions. Therefore, 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. Here we propose to measure the power information at function level, where each function may consist of tens of instructions, to capture the inter-instruction power effect better and thus provide better accuracy. 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:

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

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. Execution time can be measured either in

seconds or simulator ticks; energy data are expected from the power simulator, either directly if the tool provides such feature or indirectly as the product of power and execution time.

3.2. *Power data bank*

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, or some other structures. 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 directly for each library function and basic instruction and store the results in the so-called "power data bank".

"Power data bank" consists of the power information, such as energy consumption and execution time, for library functions and basic instructions. These data are obtained from power simulations on a set of well-designed test codes, each test code targets one or more library functions and basic instructions. The power simulator can be at any level from transistor level to RT level. Low-level power simulators can provide us high accurate data with long simulation time. The microprocessor core vendor has the freedom to choose the power simulator to achieve a desired level of accuracy.

"Power data bank" has 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/instruction. While constructing the "power data bank", we account these factors by designing test codes that will suffer constant cache misses or pipeline stalls (based on the given system configuration) and repeatedly measuring under different circumstances.

With the "power data bank" of a given microprocessor, it becomes possible to estimate the average power dissipation to execute an embedded software by our code coverage-based technique. We decompose the machine code into library functions, user-defined functions, *main()* body, and other parts. The power information for the library functions and instructions in the *main()* body are provided by the "power data bank". For user-defined functions, in particular for those being frequently called in the program execution, we first evaluate their power information using the most detailed data from the "power data bank" and then treat these user-defined functions as built-in library functions.

3.3. *Function-level power modeling*

We discuss the approach to build function-level power model, in particular, how to evaluate each term in Eq. (1) by a power simulation tool. Implementation details are

explained in the next section through the example of Toshiba TX39 microprocessor core.

3.3.1. *Built-in library functions*

A built-in library function consists of a sequence of basic instructions from the core's instruction set. At the program's execution time, library functions are normally called by jump or branch instructions. The program counter will return to the main flow upon the completion of these library functions. Notice that the execution of a library function may not be necessarily sequential, because jump/branch instructions may also be used in the implementation of such library function. However, taking all these jump/branch instructions into consideration will make it hard to build the "power data bank". Moreover, this effect can be ignored due to the following reasons: (i) these jump/branch instruction jumps over a small portion of, and never goes outside of, the instruction sequence, and (ii) such jumps depend on values of parameters to the function, which are generally not available until run time.

Now 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.

In the "power data bank", we have multiple entries for each function and instruction to capture the impact of cache miss/hit, pipeline stalls, and so on. Many reasons can cause cache misses and pipeline stalls. Instead of treating these effects explicitly and separately,[13,12] 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.

3.3.2. *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-defined 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 with special care. 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.

In Fig. 1, procedure *test()* calls two other user-defined functions *test1()* and *test2()*. No user-defined procedures are called in the omitted part of the codes. We refer *test2()* as a *basic user-defined function* because it does not depend on any other user-defined functions.

Collecting the power data for function *test()* requires those data for functions *test1()* and *test2()*, and a request for *test2()* is also made from *test1()*. The power data for *test2()* is first calculated from the "power data bank" with special care being paid to *test2()*'s localities, for example, all its basic instructions will be considered and the most detailed power data for the library functions in *test2()* will be used. This complementary method considers the locality of user-defined functions

```
void  test()          void  test1()         void  test2()
{                     {                     {
  ...                   ...                   ...
  test1();              test2();            }
  ...                   ...
  test2();            }
  ...
  test1();
  ...
}
```

Fig. 1. Three user-defined functions. The omitted portions of the codes do not make calls to user-defined procedures.

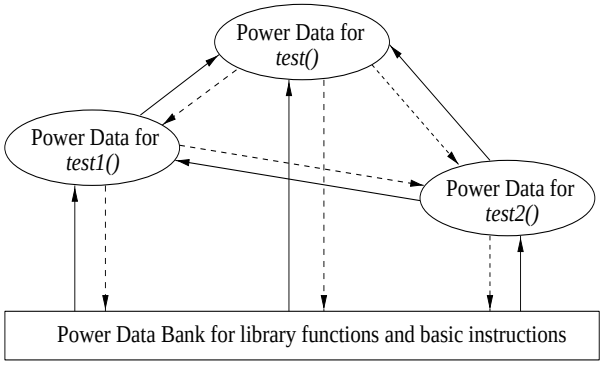

Fig. 2. Recursively measuring power data for user-defined functions. (Dotted edges show power data requests, solid edges shows the answers to such requests.)

and thus makes a more accurate estimation. Similarly, the power data for *test1()* and finally *test()* are computed in turn as illustrated in Fig. 2.

This recursion can be avoided by using a function-dependency graph. Such graph takes all user-defined functions (including *main()*) as nodes, and there is a direct edge from node A to node B if B calls A. We start from any basic user-defined function and traverse the graph to determine each node's power data following a topological order. The node associated with *main()* is the unique sink of the graph.

3.3.3. *main()*

The *main()* part of the program that we refer here is the remainder of the machine code after excluding all the library functions and user-defined procedures. This part of the machine code, during the execution, is usually separated by the blocks of code that represent the library and user-defined functions. Consequently, to measure the power dissipation on this part, we are facing either individual or short sequence of basic instructions. It takes the microprocessor little time and energy to execute these small pieces of basic instructions. This makes it very difficult to obtain accurate estimation. However, as we discussed earlier, *main()* itself contributes very little to the total energy consumption and execution time. Therefore, we can ignore the *main()* part without significant loss of accuracy. If the *main()* part is not negligible, (in cases such as intensive computation involving only the instruction set, which is performed inside *main()*), then we should treat it in the same way as a user-defined function and measure its power behavior directly from the "power data bank".

3.3.4. *Others*

We have considered the parts of the machine code that implement built-in library functions and user-defined procedures, as well as the *main()* body of the program that integrates these function calls. The remaining part of the machine code supports the program's execution and consume power too. 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; and instructions that save the information when the program starts and restore them when *main()* is completed. In our experience, the energy and execution time on this part depend on the program's execution, but the variations are so small that we can model them as constants.

3.4. *Code coverage-based power estimation*

Our goal is to estimate the power consumption of an embedded software on a given microprocessor core. We achieve this by decomposing the machine code and applying the function-level power model. Figure 3 illustrates the global flow of our code coverage-based power estimation method.

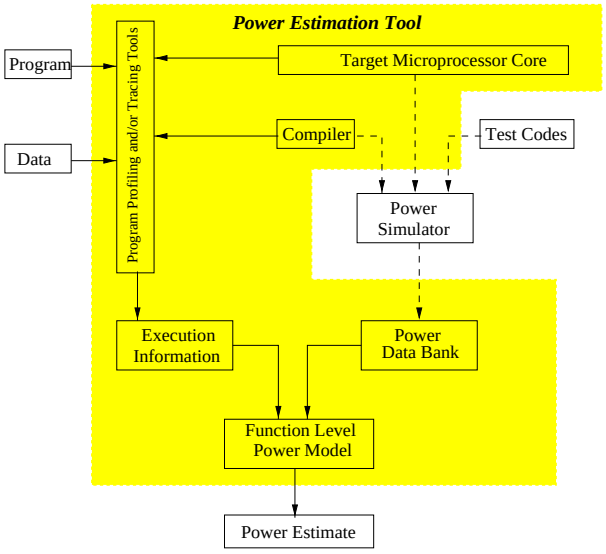

Fig. 3. Global flow for the code coverage-based power estimation.

The components connected by the dotted edges (including the target microprocessor core, compiler, test codes, and a power simulation tool) depict the procedure of building the "power data bank". For a fixed hardware configuration (microprocessor core, I-cache and D-cache set up, hardware reset program, etc.), we design a set of test codes. Each test code targets one or more built-in function or basic instruction supported by the microprocessor core. To have a better measurement of the power data with cache misses and pipeline stalls, we also have test codes that have frequent cache misses and/or pipeline stalls. For instance, when the cache size, associativity, cache line size, and replace algorithm are specified, we can use the well-documented cache memory design techniques to make the cache misses exactly predictable. Each test code will be compiled and its execution on the target microprocessor will be simulated by a power simulator. From the simulation results, we extract the energy consumption and execution time for each built-in library function and basic instruction while considering parameters such as cache misses and pipeline stalls. This information is deposited in the "power data bank". Note that although this simulation may take a long time, it will be performed only once for a fixed hardware, independent of the application programs to be executed on the core.

The code coverage-based 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 4 illustrates the three major steps of the estimation process.

In phase I, parameters for the program tracer and power model are collected. The source program is compiled and all the built-in library functions and user-

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 profiling and 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;

Fig. 4. Pseudo-code for the function-level power estimation.

defined functions are identified from the compiler's symbol table. Note that not all these functions will be called during the execution of the source program with the given input data. However, the information of each function, such as its starting and finishing points, will be passed to the program profiling/tracing tools in the next step to determine whether and how many times a function will be used. Also in this preparation step, a function-dependency graph is built and a topological order is found for the power data calculation in phase III.

We pass the required information to the program profiling and tracing tools. This may involve the compilation of the source codes and some other pre-processing steps that depend on the profiling/tracing tools being used. The detailed behavior of the program during execution is collected by profiling/tracing tools, such as the number of times each function has been executed, the cache miss rate, and pipeline stalls.

The power data for each user-defined function is computed according to the topological order in phase I. Recall that *main()* is the sink of the function-dependency graph, i.e., the last node according to any topological order. Therefore we get the power estimate of the input program at the end of this step. Each user-defined function's power data is calculated by the function-level power model, based on the "power data bank" and the other functions being called. As we have mentioned earlier, the "power data bank" has multiple entries for the library functions and instructions with different levels of precision, where the most accurate one with the available profiling/tracing information will be selected.

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 code coverage-based estimation technique is also more efficient than the instruction-level power model in collecting power data and evaluating a program's power. First, power data is collected only for built-in functions and basic instructions instead of individual and pairs of basic instructions. Second, the occurrence of library functions and user-defined procedures are used in the power estimation instead of individual and pairs of basic instructions. Moreover, we expect better accuracy from this power estimation technique because it captures the inter-instruction impact, not only between pairs of instructions, but also among library functions which may consist of dozens of instructions in implementation. This has been demonstrated by extensive experiments and some part of the results are reported in the next section.

4. Experimental Results

In this section, we report the simulation results when we apply the proposed function-level power model and code coverage-based power estimation technique to a commercial RISC microprocessor. We first briefly explain the target Toshiba TX39 microprocessor core and our simulation platform. Then we show the experimental results to support our claims on the efficiency and accuracy of the power modeling and estimation.

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 5 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,

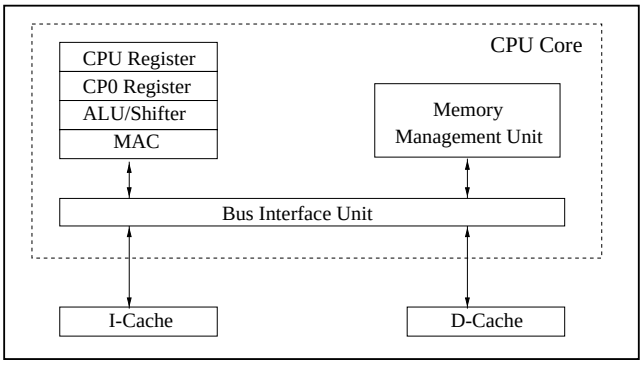

Fig. 5. Block diagram of the TX39 processor core.[41]

the unit MAC for multiply/add, memory management unit, and bus interface unit.

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.[42] We use the power simulation and analysis tools provided by Senté, Inc.[43]

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, any integer sorting program will need to call the library function *dcmp* to compare the magnitudes of two integers frequently; and a program that calculates the sum of a set of mixed integers and floating-points will intensively make function calls to convert integer to floating-point and do floating-point additions.

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 run-time 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".

4.3. *Model validation*

After building the "power data bank", we have tested our function-level power model and code coverage-based estimation technique on a large set of programs. For each 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. We then compare these two power estimates to validate our model.

The selected test programs are representative of numerically intensive applications, such as scientific computing and more importantly multimedia application. In particular they have structure and operation composition that is similar to a number of applications from the MediaBench benchmark.[44] Our test programs include

three sorting algorithms (bubble sort, quick sort, and insertion sort), several fractions of the MPEG2 codec source code, and artificial programs for test purpose. The average power estimation error is within 3% and we have the following interesting observation: *our power model gives more accurate power estimates on programs that call more built-in library functions* (e.g, the power estimate on quick sort is better than bubble sort.). We believe that this is due to the self-correction of the model. The power consumption depends on the input data and by no means the "power data bank" can provide an accurate one-fits-all power information for a built-in library function without knowing its operand(s). Consequently, the power data, which gives a function's average power consumption, will either over-estimate or under-estimate the function's actual power consumption. However, these errors will compensate each other and result in rather precise power estimates in the long run.

We select six representative test programs and report the detailed validation results. They have frequently called the following six built-in 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). Table 1 describes their functionalities which are mainly integer/floating-point additions/subtractions. The operands are extracted from MPEG2 applications (such as multiples of the values of $\cos(\frac{(2i+1)u\pi}{16}) \cdot \cos(\frac{(2j+1)v\pi}{16})$ in the DCT formula) and stored in arrays. (The sorting algorithms only call the **dcmp** function frequently and the MPEG2 codec calls many functions and thus not suitable for a demonstration here although our power model does give accurate power estimates on all the test code fragments.).

We first use the power simulator to simulate the full execution of each program. The obtained average power and execution time are used as reference. Then we use program profiler and tracer to trace the program's execution in order to collect the number of times each function has been called, and the number of cache misses. These results are shown in Table 2. We consider the first call of each function as an I-cache miss, and the rest calls to the same function as I-cache hits due to the fact that the 4KB I-cache is sufficient to hold the instructions for the above small test programs. Now, for each of the built-in library function blocks that has been used, we fetch its power information from the pre-calculated "power data bank". These include the simulated energy consumption and execution time (in simulator

Table 1. Description of the test programs.

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

ticks) per execution. Finally we apply our function-level power model to calculate the average power and execution time. These are reported in the last two rows in Table 2.

Table 2. Number 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.)

	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%

Power estimate with high accuracy (an error within 3.5% in average) is achieved as one can see from Table 2. 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()* body that connect the built-in library function blocks and others. When we model this part and add its effect to the execution time, we are able to reduce the error in execution time T to less than 4%. However, it has little impact on average power.

We further validate our assumption on the I-cache miss/hit by simulating the repetitive execution of each individual built-in library function. The results demonstrate that, the first execution of each function block takes more time than the rest runs while the rest runs are executed at roughly 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 Ref. 12, the impact of different input data has been studied in terms of the number of 1's in the data's binary representation. They conclude that such impact is small (less than 5%). Under our experimental platform, this impact is even harder to observe because we estimate average power at function-level instead of instruction-level. We measure the average power for a function as the ratio of the energy dissipation and the execution time to execute the function. The impact of different input data to the power dissipation of a single instruction is shadowed by the relatively long execution time of the entire instruction sequence.

5. Conclusions and Future Work

Power is one of the major concerns when users select hardware to execute 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. Existing tools, in particular simulation-based tools, usually require a detailed description of the core to get a reasonably accurate power estimation. However, most vendors are reluctant to release such information. To bridge this gap, we propose a power model and estimation methodology at function level. In our approach, the core vendor builds a "power data bank" and makes it public to users. The "power data bank" consists of the power information of the built-in library functions and basic instructions. The users can decompose the machine code of their program and use the "power data bank" to evaluate the average power dissipation during the program execution without running any power simulators. 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. M. Pedram, "Power minimization in IC design: principles and applications", *ACM Trans. Design Automation of Electronic Syst.* **1**, 1 (1996) 3–56.
2. E. Macii, M. Pedram, and F. Somenzi, "High-level power modeling, estimation, and optimization", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **17**, 11 (1998) 1061–1079.
3. M. Borah, R. Owens, and M. J. Irwin, "Transistor sizing for minimizing power consumption of CMOS circuits under delay constraint", *Int. Symp. Low Power Design*, 1995, pp. 167–172.
4. D. S. Chen and M. Sarrafzadeh, "An exact algorithm for low power library-specific gate re-sizing", *Design Automation Conf.*, 1996, pp. 783–788.
5. P. Girard, C. Landrault, S. Pravossoudovitch, and D. Severac, "A gate resizing technique for high reduction in power consumption", *Int. Symp. Low Power Design*, 1997, pp. 281–286.
6. L. Benini and G. De Micheli, "Automatic synthesis of low-power gated-clock finite-state machines", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **15**, 6 (1996) 630–643.
7. A. Hemani, T. Meincke, and S. Kumar *et al.*, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style", *Design Automation Conf.*, 1999, pp. 873–878.
8. I. Hong, D. Kirovski, G. Qu, and M. Potkonjak, "Power optimization of variable-voltage core-based systems", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **18**, 12 (1999) 1702–1714.
9. G. Qu and M. Potkonjak, "Techniques for energy minimization of communication pipelines", *Int. Conf. Computer-Aided Design*, 1998, pp. 597–600.
10. T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling", *Int. Symp. Low Power Electronics and Design*, 2000, pp. 9–14.
11. I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processor", *IEEE Real-Time Systems Symp.*, 1998, pp. 178–187.
12. V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization", *IEEE Trans. VLSI Syst.* **2**, 4 (1994) 437–445.

13. M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded DSP software", *IEEE Trans. VLSI Syst.* **5**, 1 (1997) 123–135.
14. M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli, "Efficient power estimation techniques for HW/SW systems", *IEEE Alessandro Volta Memorial Workshop Low-Power Design*, 1999, pp. 191–199.
15. C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski, "The design and implementation of PowerMill", *Int. Symp. Low Power Design*, 1995, pp. 105–108.
16. R. Marculescu, D. Marculescu, and M. Pedram, "Adaptive models for input data compaction for power simulations", *Asia South Pacific Design Automation Conf.*, 1997, pp. 391–396.
17. F. N. Najm, "Transition density: a new measure of activity in digital circuits", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **14**, 1 (1993) 310–323.
18. D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips", *IEEE J. Solid-State Circuits* **29**, 6 (1994) 663–670.
19. C. T. Hsieh, Q. Wu, C. S. Ding, and M. Pedram, "Statistical sampling and regression analysis for RT-level power evaluation", *Int. Conf. Computer-Aided Design*, 1996, pp. 583–588.
20. T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago, "Evaluation of architecture-level power estimation for CMOS RISC processors", *Int. Symp. Low Power Design*, 1995, pp. 44–45.
21. J. M. Rabaey, "Exploring the power dimension", *IEEE Custom Integrated Circuits Conference*, 1996, pp. 215–220.
22. A. Nandi and R. Marculescu, "System-level power/performance analysis for embedded system design", *Design Automation Conf.*, 2001, pp. 599–604.
23. C. Akturan and M. F. Jacome, "CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors", *Int. Conf. Computer-Aided Design*, 2001, pp. 112–118.
24. D. T. Blaauw, A. Dharchoudhury, R. Panda, S. Sirichotiyakul, C. Oh, and T. Edwards, "Emerging power management tools for processor design", *Int. Symp. Low Power Electronics and Design*, 1998, pp. 143–148.
25. L. P. Yuan, C. C. Teng, and S. M. Kang, "Statistical estimation of average power dissipation in CMOS VLSI circuits using nonparametric techniques", *Int. Symp. Low Power Electronics and Design*, 1996, pp. 73–78.
26. A. Bogliolo, L. Benini, M. Favalli, and G. De Micheli, "Regression models for behavioral power estimation", *Integrated Computer-Aided Engineering* **5**, 2 (1998) 95–106.
27. D. Marculescu, R. Marculescu, and M. Pedram, "Information theoretic measure for power analysis", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **15**, 6 (1996) 599–610.
28. M. Nemani and F. N. Najm, "Towards a high-level power estimation capability", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.* **15**, 6 (1996) 588–598.
29. J. M. Anderson, L. M. Berc, and J. Dean *et al.*, "Continuous profiling: Where have all the cycles gone?" *ACM Symp. Operating Syst. Principles*, 1997, pp. 1–14.
30. T. Ball and J. R. Larus, "Optimally profiling and tracing programs", *ACM Symp. Principles of Programming Languages*, 1992, pp. 59–70.
31. B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling", *ACM SIGMETRICS*, *Conf. Measurement and Modeling of Computer Syst.*, 1994, pp. 128–137.
32. J. R. Larus, "Efficient program tracing", *IEEE Computer* **26**, 5 (1993) 52–61.
33. J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A

trace-driven analysis of the Unix 4.2 BSD file system", *ACM Symp. Operating System Principles*, 1985, pp. 15–24.
34. D. L. Dill, "Trace theory for automatic hierarchical verification", The MIT Press, Cambridge, Mass., 1988.
35. G. H. Kuenning, G. J. Popek, and P. L. Reiher, "An analysis of trace data for predictive file caching in mobile computing", *USENIX Summer Conf.*, 1994, pp. 291–303.
36. K. Claffy, H.-W. Braun, and G. Polyzos, "A parameterizable methodology for internet traffic flow profiling", *IEEE J. Selected Areas in Communications* **13**, 8 (1995) 1481–1494.
37. H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz, "TCP behavior of a busy internet server: Analysis and improvements", *IEEE INFOCOM, Conf. Computer Communications*, 1998, pp. 252–262.
38. S. D. Gribble and E. A. Brewer, "System design issues for internet middleware services: Deductions from a large client trace", *USENIX Symp. Internet Technol. and Syst.*, 1997, pp. 207–218.
39. C. T. Hsieh, M. Pedram, G. Mehta, and F. Rastgar, "Profile-driven program synthesis for evaluation of system power dissipation", *Design Automation Conf.*, 1997, pp. 576–581.
40. http://www.mpeg.org/MSSG.
41. 32-Bit TX System RISC TX39 Family Architecture, Toshiba Corp., 1999.
42. http://www.ghs.com.
43. http://www.senteinc.com.
44. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems", *MICRO-30*, 1997, pp. 330–335.