SOFTWARE PERFORMANCE EVALUATION

The specification of a software system includes describing what software is supposed to do, that is, its functionality. Its performance, size, modularity, reliability, cost, and design time are other metrics that affect its design process. Power consumption is a newer and an increasingly important software metric. Evaluating or analyzing these metrics is an integral part of the software design process. This article focuses on evaluating the performance and power metrics of software.

Software performance refers to execution time of the given program. Software power consumption refers to the power consumption by the hardware on which the given piece of software is executed. Both performance and power depend on the number of instructions executed during the program and the sequence in which the instructions are executed. Performance further depends on the execution time of each executed instruction, and similarly, power depends on the power cost of each executed instruction. However, for the purpose of exposition, it is better to treat these metrics separately. The first part of the article deals with software performance evaluation, and the second part deals with software power evaluation and the relationship between the two metrics.

COMPONENTS OF SOFTWARE PERFORMANCE EVALUATION

Software performance evaluation is the estimation and analysis of the running time of programs. The scope of this article is limited to single software components or processes. The analysis of multiple processes belongs to the larger field of system-level performance analysis. However, the concepts and techniques discussed here are fundamental to both single and multiprocess performance evaluation:

Three components affect a program's running time:

1. *System.* The underlying hardware and the system software. Following are main elements of the hardware that affect program performance:
   (a) Instruction execution time: This is the amount of time needed to execute each basic instruction. It depends on the processor architecture and is typically reported in terms of the number of processor clock cycles needed.
   (b) Memory access time: The memory system usually has a hierarchical organization. On-chip caches are the lowest level, followed by off-chip caches, main memory, and finally disk storage (in the case of virtual memory systems). Data elements needed most frequently are stored at the lowest levels. The time

needed to access memory and the storage size increases up the hierarchy. Thus, the execution time for an instruction that involves memory access depends on the level of hierarchy at which the access occurs. It is one or two cycles for on-chip cache, several cycles for off-chip caches, tens of cycles for main memory, and thousands of cycles for disk accesses.

(c) System software: The system software (e.g., the operating system, device drivers, etc.) is the level of software just above the hardware. It coordinates and provides services to the user-level programs. Thus it is a factor in the performance of programs. For example, in systems with multiple processes, the operating system schedules the use of the processor for individual processes. The total execution time of the program thus depends on how often and for how long it is scheduled to run. The operating system also provides predefined software routines for performing specific tasks, such as communicating with input/output devices. Programs use these service routines, and thus their performance also depends on the performance of these routines.

2. *Inputs.* The input data values and external stimuli, such as interrupts, are external factors affecting a program's performance. The behavior of a program, or the path that it takes to execute, often depends on the value of input data items. The performance of a program thus cannot be looked at independently of the input data.
3. *Program.* The following factors internal to software also affect performance:

(a) Algorithm: The algorithm defines the fundamental "work" to solve the problem, that is, the basic steps and the sequence of steps needed to achieve the given function. The choice of algorithm determines the performance of the program at the highest level. For example, for sorting a given set of data values in an increasing or decreasing order, a program implementing the *bubble sort* algorithm is slower than one implementing *quick sort,* as the number of data items to be sorted increases.

Performance analysis of algorithms is usually in the form of *asymptotic* or *order* analysis (1). The idea is to specify a mathematical function which describes the order of magnitude of the algorithm's execution time. It describes the asymptotic increase in the running time of a program as its input sets gets larger. For example, for sorting n elements, the bubble sort algorithm is of the order of n^2 [$O(n^2)$], and quick sort is of the order of $n \cdot \log(n)$ $\{O[n \cdot \log(n)]\}$. Thus, as n increases, quick sort outperforms bubble sort. [More rigorously: a function f is of an order of function g, i.e., $f = O(g)$, if there are positive constants r and c, such that $f(n) \leq r \cdot g(n)$ for all n bigger than c.]

(b) Program implementation: In most practical situations, the algorithm is already known and the performance of the algorithm's implementation has to be evaluated. This can be done at different levels. The higher level of abstraction is the source code of the program, as written in a higher-level language, such as FORTRAN, LISP, or C. Different numbers of statements, different ways of using program constructs, such as "for," and "while" loops, different organizations of program data in the form of data structures, all lead to different program performance. The process of translating the high-level source code into machine instructions, that is, the process of *compilation* also leads to varying performance, because there are many ways to map a high-level program into a machine-level program.

CLASSIFICATION OF SOFTWARE PERFORMANCE EVALUATION

Software performance evaluation is classified under two broad categories:

Average Case

Average case performance analysis loosely refers to the performance of the program in the most common case, that is, the most likely input values and external system behavior. The performance of the program based on a small set of test runs is evaluated, and it is used to represent its overall performance. Few or no guarantees are made on the variance of the performance. A typical example is a text compression program, whose average speed is often stated in bytes compressed per second.

Extreme Case

Systems generally interact with the outside world. This involves measuring sensors and controlling actuators, communicating with other systems, or interacting with users. These tasks have to be performed at precise times. A system with such timing constraints is called a *real-time system.* A typical example is an automotive engine control unit which gathers data from sensors and computes the proper air/fuel mixture and ignition timing for the engine within a single rotation. Software is increasingly used to implement the functionality of such systems (in which case they are called *embedded computer systems*). In such systems, the response time of the software must comply with the specified timing constraints under all possible conditions. Thus, the performance metric of interest here is the extreme case performance of the software.

APPLICATIONS OF SOFTWARE PERFORMANCE EVALUATION

The main applications of the categories of software performance evaluation previously mentioned are as follows.

Average Case

- *Performance Tuning.* Performance analysis helps to identify performance bottlenecks in programs. A small section of badly written code could be responsible for slowing down the entire program, and searching for this piece of code is an activity with potentially high returns. Then optimization efforts are focused on this piece of code, maximizing their efficiency.
- *System Design Exploration.* Performance analysis of programs also indicates the performance bottlenecks in the system. For example, it can point to the excessive re-

sponse time of an I/O subsystem or the performance degradation of the network interface. Without careful analysis, these trouble spots go undetected.

- *Compiler Optimizations.* Compilers are software tools that translate high-level programs into machine instructions. Performance analysis provides quantitative data about the quality of the code generated. These data guide the development and refinement of compiler optimizations to generate better codes.

Extreme Case

- *System Verification.* Extreme case analysis is used to answer the question: Does the system do what it is supposed to? In systems with specified performance constraints, meeting the constraints is necessary for correct functionality. For example, consider a factory control system program that must respond to a pressure sensor signal by releasing a valve within 50 ms. If it does release the valve but does so after 60 ms, it has not met its desired functionality. In certain situations, performance violations result in catastrophic situations, for example, in the previous case or in fly-by-wire aircraft systems. Verification requires that the specified performance constraints are met under all input conditions and all possible starting states.
- *System Hardware Selection.* As a flip side of the previous point, it is also important to verify that the system does not overly exceed the performance constraints. Pessimistic estimates of the extreme case performance of the software lead to overdesigned systems, for example, choosing a faster processor or a more efficient operating system. Overdesign typically translates into more expensive systems. Even a small increase in system cost is undesirable, especially for high volume products.
- *Design Space Exploration.* In embedded systems, the design process involves deciding how the functionality is partitioned between the hardware and the software. This is called the hardware-software partitioning problem. Software is typically a more economical way of implementing functionality, as long as it runs fast enough to meet the system performance constraints. Thus, a careful choice of hardware-software functionality requires accurate performance analysis of software.
- *Real-Time Schedulers.* Real-time systems must respond to environmental stimuli in real time. The factory control system mentioned previously is an example. Real-time systems typically must respond to multiple stimuli within specified time limits for each. The software for such systems is designed in the form of multiple processes, and system software, that is, the scheduler, has to schedule the processes to guarantee that the deadlines for each process are met. The schedulers thus need accurate information about the performance bounds for each process. Loose estimates lead to inability to guarantee deadlines or the poor utilization of system resources.

METHODS FOR SOFTWARE PERFORMANCE EVALUATION

Average case performance analysis has two main components:

1. Modeling: estimating the average execution time for a *given* sequence of instructions when they are executed on the given system.
2. Path analysis: finding the "typical" paths from among all possible ones through the program or the paths that execute most often. A path is an instruction trace or a sequence of consecutively executed instructions.

Extreme case performance analysis has similar components:

1. Modeling: estimating the extreme case execution time for a given sequence of instructions when they are executed on the given system.
2. Path analysis: finding the extreme case paths through the program.

Modeling is directly related to the system and to the inputs. Path analysis is directly related to the program and to the inputs. Both must be solved for accurate performance analysis. Methods to solve them are discussed here.

Modeling

For modeling, simple methods are often used to predict the performance of a given sequence of instructions. The idea is to add up the execution time (the number of execution cycles) of each instruction in the sequence. The problem with this is that, in modern processors, the execution time of an instruction in a given sequence depends on the surrounding instructions and on the processor state. In addition to the processor pipeline, the caches, the system memory, and the I/O subsystem also must be modeled because they directly affect performance. It is no longer sufficient to specify a single number as the execution time for an instruction. For example, the Motorola MC68040 processor manual contains 37 pages of formulas to describe instruction timing.

Using pessimistic bounds for the execution time of each instruction makes the task easier, but the final estimate has too much approximation error to be useful. As an illustration, consider the following instruction sequence from the MIPS R4000 processor. To account for the variability of execution time for each instruction, bounds for the number of execution cycles for each instruction are also shown:

```
mul.d  f4,f6,f8    1-26 cycles
and    r1,r2,r3    1-15 cycles
lw     r4,o(r1)    1-29 cycles
mul.d  f5,f6,f8    1-26 cycles
```

Separately adding the upper and lower bounds for each instruction gives the following bounds for the performance of the sequence: 4-96 cycles. Although these bounds are guaranteed, their large difference limits the usefulness of the estimate.

A more detailed analysis of the previous sequence can lead to tighter bounds. For example, the second instruction in the sequence is guaranteed to finish in just one cycle, because it does not interact with the previous instruction. This improvement in the estimate requires additional information about the processor. Similarly, a model for the cache behavior improves the bounds for the load instruction `lw`, because then it is possible to determine if the data operand is available in the cache or not.

Thus, for more accurate performance analysis, a more refined modeling of all potential interactions among instructions and interactions with the external system is desirable. Generally this is very difficult. Analytical models have been built, however, to solve the modeling problem in some cases, and these are discussed in greater detail later.

Path Analysis

For straight line code, that is, code with no jumps or branches, there is exactly one execution path to consider. Complexity creeps in only in the presence of control flow constructs, such as branches and loops. Each branch doubles the number of possible paths, leading to an exponential blowup of the number of possible execution paths. This makes path analysis a very hard problem to solve generally. [In particular, extreme case analysis is *undecidable,* because it is equivalent to the halting problem (2)]. Different techniques with varying levels of effectiveness are used for path analysis.

General Heuristics. General heuristics derived from basic program statistics are used for informal path analysis. This is more appropriate for average case analysis where crude performance estimates often suffices. These estimates are based on code size and prior experience or on heuristics, for example, the observation that most backward branches are taken and most forward branches are not taken. Prior experience, designer insight, and crude estimates are also used for extreme case analysis. Generally, these are less applicable in that case, because of the desire for guaranteed tight bounds.

Profiling. This technique incorporates elements of both modeling and path analysis. It is a *dynamic* technique, because it requires actual execution of the program under consideration. The term profiling, as used here, also includes techniques known as *program tracing.* It involves collecting run-time statistics for a specific run of the given program.

The common mechanism is *instrumenting* the program code (3). The original program [see Fig. 1(a)] is modified to create an instrumented version of the program [see Fig. 1(b)]. This involves adding instructions and data spaces to the code to keep count of the number of times each basic block is executed. A basic block is the largest contiguous sequence of instructions that contains a single entry point (at the start of the sequence) and a single exit point (at the end of the sequence). So each instruction in the basic block is executed as many times as the basic block. Instructions are also added for specialized bookkeeping, for example, to keep track of the number of times conditional branches are taken. Information such as this can help the program writer fine-tune the performance of the program.

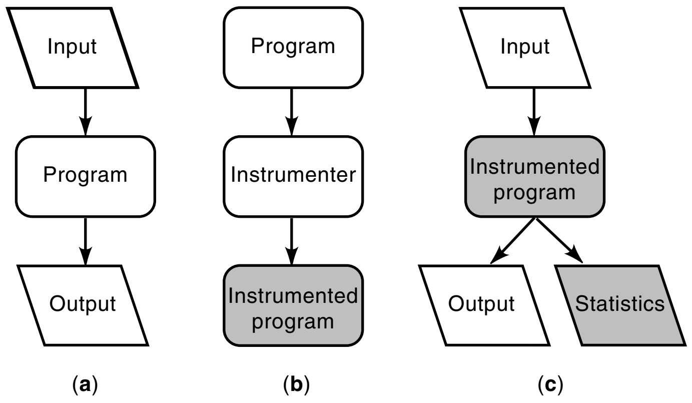


Figure 1. Profiling through code instrumentation.

Then the instrumented code is executed, and the desired statistics are generated as a by-product of the execution [see Fig. 1(c)]. The statistics are then put out to the user in a variety of forms, such as tables or graphs, depending on the user's needs. Profilers come in different shapes and forms and provide a variety of capabilities. Representative examples include pixie (4), a profiling tool for MIPS processors, and VTune (5), a profiling tool for Intel processors.

The limitations of profiler tools are that they modify the code that they are supposed to profile. So the collected statistics do not fully represent the original code. Minimizing the impact of the code instrumentation typically results in less detail in the collected data. Thus, there is a trade-off involved here. In addition, profilers also cannot account for the full complexities of today's processors, such as long, complex pipelines and elaborate memory hierarchies involving multiple levels of caches and virtual memory systems. These complexities introduce a great amount of variation in the number of cycles needed to execute different instances of the same instruction. Profilers must rely on approximate models to account for the impact of these effects on performance. For example, pixie models pipeline interlocks pessimistically (assume that an interlock happens each time it can), and assume a perfect memory system, that is, one where all the data are available in the same constant amount of time.

The following are other profiling techniques:

- *Sampling.* The technique of sampling is an alternative to instrumentation. In the time-based version of this technique, the processor is periodically interrupted, and the addresses of the instructions under execution are recorded. The recorded addresses are matched against routines called from a given program, and a database of the sampled data is created. The ratio of samples in which a particular routine was being executed is an estimate of the ratio of time spent in executing that routine. VTune is such a profiling tool. It provides graphical interfaces to allow the user to extract a variety of information about the program's behavior. The UNIX profiling utility prof is also based on sampling.
- *Hardware Monitoring.* Certain modern microprocessors monitor the behavior of the processor as it executes given programs in real time. Counters are built into the processor to count events, such as cache misses, branch mispredictions. These counters are readable by user-level programs, and software tools are built to use this information. This is a very powerful idea, and will see more use in the future.
- *Simulation.* Programs are profiled by simulating their execution on models of the system, instead of running them on the real system. Simulation is orders of magnitude slower than real execution. However, the simulation models are built to provide arbitrarily high levels of detail. This level of detail is impossible to obtain through instrumenting profilers and very expensive to monitor

through hardware. Simulation is most commonly used to evaluate the performance impact of caches. Information, such as the distribution of the kind of cache misses and the relationship of cache behavior to program data structures, etc. is obtained. Representative examples of such tools include memspy (6) and CProf (7).

Profilers are very useful tools, despite their limitations because of the following:

- Modeling the complexities of today's systems is a very hard task, as previously discussed. However, profilers provide an avenue for performance evaluation while dealing with this complexity. Although profilers are not always perfect representations of real systems, they can be tailored to be accurate for the statistics of most interest to a given set of users. This may not be possible through analytical modeling techniques, which are still an area of research and are currently limited in their application.
- Profilers enable the program developer to identify "trouble spots" within the code. User-friendly data that process back-ends of the tools correlate the collected profiling data with the original source code and make this task easier. Then program performance improvement efforts are focused on these trouble spots, improving the programmer's productivity.

Analytical Techniques. Analytical path analysis techniques do not require actual execution of the given program. This is a relatively new approach, and systematic and rigorous analytical techniques are an area of active research. These techniques are mostly applicable to extreme case performance analysis and are discussed later.

AVERAGE CASE PERFORMANCE ANALYSIS

Average case performance is analyzed with profiling techniques. Although general heuristics are acceptable in some cases, generally they do not provide the level of accuracy needed. The biggest problem in average case analysis is the question: What is the average case for the given program? Given a fixed system and a program, this translates into determining the average case input set. There are no generally applicable standards or guidelines for this. Instead the performance of the system is evaluated from a sample input data set that is considered representative of the typical application environment for the program. Then profiling techniques are used with the sample input data set. Profiling implicitly accounts for both modeling and accurate path analysis for the given input set.

It should be apparent from the above discussion that the usefulness of average case performance analysis is tied to the ability to characterize the "typical" input set. This is a nontrivial problem, especially for complex functions, for example, How can one determine a representative set of images for a JPEG compression program?

An approach called benchmarking is used specifically to compare different software implementations for a given application. The idea is to select an input set that will be used as a standard benchmark. The chosen benchmark set must be acceptable to all concerned parties and is supposed to be equitable and not have an inherent bias toward any implementation. For example, to compare different JPEG compression software packages that may be available in the market, a common set of images is used for comparative analysis. Although standard benchmarks are often controversial, they practically solve the problem of input characterization. A general solution to the problem, however, is still an open area of research.

EXTREME CASE PERFORMANCE ANALYSIS

Extreme case performance analysis is typically performed by analytical techniques because dynamic techniques, such as profiling, require actually executing the given program. For large programs with many conditional branches, only a fraction of all possible paths are exercised during program execution. (The total number of paths is exponential in terms of the number of branches. Backward branches (i.e., loops), make the problem even harder.) Extreme case performance analysis, however, requires determining the worst case (or best case) path from all possible paths. Exercising all possible paths by profiling is not a practical solution. On the other hand, analytical techniques try to traverse the search space more efficiently. Extreme case analysis is generally undecidable. Thus, for analytical techniques to work, the program must meet certain restrictions (2):

- All loop statements must have bounded iterations, that is, they cannot loop forever.
- There are no recursive function calls.
- There are no dynamic function calls.

Analytical techniques for extreme case performance analysis are a relatively recent area of research. Some representative ideas are briefly described here.

Extreme Case Selection

In worst case (best case) analysis, a straightforward approach is to always assume that the worst case (best case) choice is made for each branch and loop. For example, in the *timing schema* approach (8), for an if-then-else statement, the execution times of the true and false statements are compared and the larger one taken for worst case estimation. Although this works well in simple cases, generally different parts of the program are related. Consider the example shown in Fig. 2. S_1 and S_3 are always executed together, and so are S_2 and S_4. If extreme case selection is used, statements S_1 and S_4 are selected for worst case analysis. However, these two state-

```
      if (ok)
S1      i = i*i + 1;   /* i is non-zero! */
      else
S2      i = 0;
      /* ... */
      if (i)
S3      j++;
      else
S4      j = j*j;
```

Figure 2. Different parts of the code are sometimes related.

ments are never executed together in practice, and the technique results in loose estimation. Other researchers have shown that this technique can be extended to allow the programmer to provide simple execution count information for certain statements. This permits nonpessimistic choices locally and leads to better estimation.

Path Enumeration

To capture the relationship between different parts of the program, another approach is to statically enumerate some of the paths. For extreme case analysis, this partial enumeration must be pessimistic, that is, it must include paths that bound the extreme case behavior, even if they are never actually exercised. It has been observed (9) that all statically feasible execution paths can be expressed by regular expressions. For example, the following equations show the regular expression for the if-then-else statement and that for the while loop statement with loop bound n, respectively:

$$\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \quad : \quad B \cdot (S_1 + S_2)$$

$$\texttt{while } B \texttt{ do } S \quad : \quad B \cdot (S \cdot B)^n$$

Let the set of statically feasible execution paths be represented by a regular expression A_p. The user can provide path information by using a script language called *information description language* (IDL) (9), which is subsequently translated into another regular expression denoted as I_p. The intersection of A_p and I_p, denoted as $A_p \cap I_p$, represents all feasible execution paths of the program. Then the best case and worst case execution paths and their corresponding execution times are determined from the regular expression $A_p \cap I_p$.

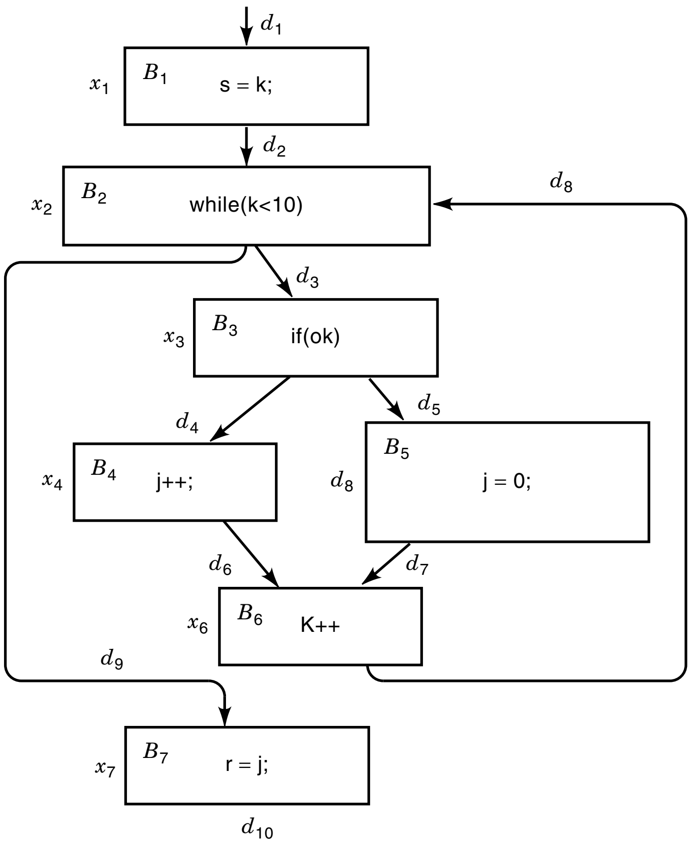
The use of IDL is a vast improvement over earlier methods because it allows expressing path relationships directly. However, the main drawback of this approach is that the intersection of A_p and I_p is a complicated and expensive operation. To speed it up, pessimistic approximations are used in the intersection operation, which limit the accuracy of the method.

Bounding Techniques

An alternative approach is used in the Cinderella project (10). The basic idea is to use integral-linear programming to determine the execution counts for all blocks in the extreme case execution of the program. Let x_i be the execution count of a basic block B_i, and c_i be the execution time of the basic block. If there are N basic blocks in the program, then the total execution time of the program is given by:

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i \tag{1}$$

Determining the values of x_i's for extreme case execution is the path analysis component of the problem, and estimating the values for c_i's is the modeling component. The possible values of x_i's are constrained by the program structure and the possible values of the program variables. These are expressed as linear constraints divided into two parts: (1) structural constraints, which are derived automatically from the program's control flow, and (2) functionality constraints, which are provided by the user to specify loop bounds and other path information. As an example, consider Fig. 3, in which a conditional statement is nested inside a while loop. Figure 3(b) shows the control flow graph (CGF). Each edge in the CFG is labeled with a variable d_i which is both a label for that edge and a count of the number of times that the program control passes through that edge. Structural constraints are derived from the CFG because, for each node B_i, its execution count is equal to the number of times that the control enters the node (inflow) and is also equal to the number of times that the control exits the node (outflow): $x_i = \sum$ inflow $= \sum$ outflow.



(**b**) Control flow graph

Figure 3. An example showing how structural and functionality constraints are constructed.

The structural constraints cannot provide any loop-bound information. This information is provided by the user as a functionality constraint. In this example, note that because k is positive before it enters the loop, the loop body is executed between 0 and 10 times each time the loop is entered. The constraints to specify this information are $0x_1 \leq x_3 \leq 10x_1$. Further, observe that the else statement (B_5) is executed at most once inside the loop. This information is specified as: $x_5 \leq 1x_1$. These constraints form the input to an integral linear programming formulation. This mechanism is more powerful than IDL (9) in describing path information (10).

FURTHER DETAILS ON MODELING

Significant variations in program execution time result from varying uses of system resources. The way the program references memory or occupies pipeline resources, etc., all have significant impact on program performance. Analytical modeling of this source of performance variation is a nascent field of research. The techniques currently available work under simplifying assumptions, which often limit their scope. Examples of these techniques are briefly discussed here. Relevant references can be obtained from (11).

Microarchitectural Resources

Today some of the key performance factors in systems revolve around the program's utilization of the processor's microarchitectural resources. Early works in this area assumed a very simple microarchitecture, such as the Motorola M68000 microprocessor, where the instruction execution times are assumed constant and independent of each other. With fairly complex superscalar pipelines becoming more common even in embedded processors, simple processor models often no longer suffice for estimating or bounding program performance. To be useful, processor performance models must consider utilizing individual functional units and the issue rate down the processor pipelines. Pipelines are relatively easy to model, and they have been studied extensively. Most pipeline modeling methods model the pipeline states within a short sequence of straight line instructions, such as a basic block. Although some works consider pipeline effects across the branches and loops, it is generally felt that analysis using limited code length sequences is fairly accurate.

Memory Behavior

With processor speeds improving at a much faster rate than memory speeds, the relative importance of memory behavior to program performance has increased significantly in recent years. In response to this, researchers have explored several ways of producing estimates, or bounds, on program memory performance. Some of these are briefly discussed here:

- Renewal theory models: The use of simulation for evaluating program memory behavior was discussed earlier. Although simulation provides arbitrarily high levels of detail, it suffers from the drawback that it is quite slow. To avoid this, the alternative is to estimate performance statistics based on samples of memory reference traces. The cache state is unknown at the beginning of each sample. To deal with this, renewal theory models have been developed to analyze memory behavior based on this partial information.
- Locality analysis: Locality analysis is widely used by compiler writers to offer good estimates of memory behavior in loop-oriented scientific codes. The analysis relies on computing a set of reuse vectors that summarize how the loop's array accesses reference (and rereference) memory locations and cache lines. This information is used to guide memory prefetching algorithms.
- CM equations: Cache-miss (CM) equations build on the ideas of locality analysis. These equations represent the detail the cache misses in loop-oriented scientific code. Mathematical equations are generated to summarize each loop's memory behavior. Mathematical techniques for manipulating these equations allow computing a number of possible solutions. Each solution corresponds to a potential cache miss. These equations provide a general framework for guiding software code optimizations to improve cache performance.
- Bounding techniques: The cinderella project presents an alternative approach to cache modeling. It integrates the impact of program execution on the cache with the bounding techniques used in path analysis (10).

SOFTWARE POWER EVALUATION

The increasing popularity of power-constrained mobile computers and embedded computing applications drives the need for analyzing and optimizing power in all the components of a system. This has forced an examination of the power consumption characteristics of all modules, ranging from disk drives and displays to individual chips and interconnects. Focusing solely on the hardware components of a design ignores the impact of the software on the overall power consumption of the system. Software constitutes a major component of systems where power is a constraint. Its presence is very visible in a mobile computer, as system software and application programs running on the main processor. But software also plays an even greater role in general digital applications. An ever growing fraction of these applications are now being implemented as embedded systems. As mentioned earlier, the functionality in these systems is partitioned between hardware and software components. The software component usually consists of application-specific software running on a dedicated processor, whereas the hardware component usually consists of application-specific circuits. Given these trends, there is a clear need to consider the power consumption by the software component of systems.

COMPONENTS OF SOFTWARE POWER EVALUATION

Software affects the system power consumption at various levels. At the highest level, this is determined by the way functionality is partitioned between hardware and software. The choice of the algorithm and other higher level decisions about the design of the software component affect system power consumption in a big way. The design of the system software, the actual application source code, and the process of translation into machine instructions, all of these determine the power cost of the software component.

To systematically analyze and quantify this cost, however, it is important to start at the most fundamental level, the individual instructions executing on the processor. Just as logic gates are the fundamental units of computation in digital hardware circuits, instructions are the fundamental unit of software. Accurate modeling and analysis at this level, therefore, is essential. Instructional level models can then be used to quantify the power costs of the higher constructs of software (application programs, system software, algorithm, etc.). This aspect of software power evaluation is akin to the modeling component of software performance evaluation (see first section).

The other component of software power evaluation consists of methods to collect information about the instructions carried out during program execution, in particular, methods to determine the number of times each instruction in the program was executed and the exact sequence in which the instructions were executed (i.e., the program path). This aspect of software power evaluation is therefore the same as the *path analysis* component of software performance evaluation (see first section).

The focus of discussion in this part of the article is on instructional level power modeling, because the path analysis component is the same as that for performance.

APPLICATIONS OF SOFTWARE POWER EVALUATION

The following are the main applications for software power evaluation:

- Software power analysis is needed to assign an accurate power cost to the software component of a system. For power-constrained embedded systems, this helps to verify that the overall system is meeting its specified power budget.
- The most common way of specifying power consumption in processors is through a single number, the average power consumption. Because power consumption varies with instructions, a single number is not enough to capture the power consumption characteristics of processors. This also makes it difficult to compare processors. Thus, there is a need to develop programs for use as power benchmarks. Instructional level analysis provides a finer level of resolution about the processor's power consumption. This additional resolution can guide the careful development of software power benchmarks and more meaningful comparisons between processors.
- For systems in which part of the functionality is implemented in software, it is natural to expect a potential for power reduction by modifying software. Software power analysis is needed to effectively exploit this potential. It helps in identifying the reasons for variation in power from one program to another. This enables the search for low power alternatives for each program. In particular, the information provided by software power analysis can guide higher level design decisions like hardware-software partitioning and choice of algorithm. It can also be directly used by automated tools like compilers, code generators, and code schedulers to generate codes targeted toward low power.

  Power reduction by software modification is cheap. In contrast to hardware-based power reduction techniques, it entails no increase in system cost or complexity. Also note that software power analysis is a new field and has gained attention only in the recent past. Until recently, the potential for power reduction through software modification was poorly understood and thus was not exploited.
- Power analysis also provides insight into where and how power is consumed within the system's hardware. This additional insight provides directions for modifications in system hardware design that lead to the most effective overall power reduction. For example, instructions can be evaluated in terms of their power cost and frequency of occurrence in typical compiler or even hand-generated code. This combined information can be used during processor design to rank the instructions that should be implemented for less power expense.

```
MOV  DX,  [BX]
MOV  AX,  CX
ADD   AX,  DX
```

Power = 1.15 Watts
Energy = 8.6×10^{-8} Joules

(a)

```
NOP
MOV  DX,  [BX]
NOP
NOP
MOV  AX,  CX
NOP
NOP
ADD   AX,  DX
NOP
```

Power = 0.99 Watts
Energy = 22.3×10^{-8} Joules

(b)

Figure 4. Illustration of the difference between power and energy.

POWER AND ENERGY

At this point, it is helpful to clarify the distinction between the term *power* that has been used so far in this article and the term *energy* as they relate to software. The average power consumed by a processor while running a given program is given by $\mathscr{P} = I \times V_{dd}$, where $\mathscr{P}$ is the average power, I is the average current, and V_{dd} is the supply voltage. Power is also defined as the rate at which energy is consumed. Therefore, the energy consumed by a program is given by $E = \mathscr{P} \times T$, where T is the execution time of the program. This in turn is given by $T = N \times \tau$, where N is the number of clock cycles taken by the program and τ is the clock period. Energy is thus given by $E = I \times N \times V_{dd} \times \tau$. Because V_{dd} and τ are constant for a given system, E is proportional to $I \times N$, whereas $\mathscr{P}$ is proportional to I.

The difference between the two metrics is highlighted by two instructional sequences (for the Intel 486DX2 processor) shown in Fig. 4. The two sequences have the same functionality, but sequence (b) has some additional NOP instructions. The NOPs are a little cheaper in terms of power than the other instructions, and thus the average power cost for sequence (b) is about 14% lower than that of sequence (a). Because of its increased running time, however, sequence (b) consumes 158% more energy. Energy consumption is the primary concern for mobile systems, which run on the limited energy in a battery. Power consumption, on its own, is important in applications where cooling and packaging costs are a concern, because power consumption leads to heat dissipation. Energy consumption is the primary focus of attention in this article. Although an attempt is made to maintain a distinction between the two terms, the term power is sometimes used to refer to energy, in adherence to common usage. Nevertheless, power and energy are closely related, and the energy cost of a program is simply the product of its average power cost and its running time.

METHODS FOR SOFTWARE POWER EVALUATION

As can be seen from the above discussion, the ability to measure the current drawn by a processor during the execution of a program is essential for evaluating the power/energy cost of software. Different methods for measuring processor current are discussed below. The main concepts described in this article are independent of the method used to measure average current.

Current Estimating Through Simulation

The most commonly used method for power analysis of circuits is through specialized power analysis tools that operate on abstract models of the given circuits. The models can be at different levels of the design process, such as the circuit, gate, or architectural level. These tools can be used for software power evaluation, too. A model of the given processor and a suitable power analysis tool are required. The idea is to simulate the execution of the given program on the model. During simulation, the power analysis tool estimates the power (current) drawn by the circuit using predefined power estimating formulas and algorithms.

However, this method has some drawbacks. It requires models that capture the internal implementation details of the processor. This is proprietary information, to which most software designers do not have access. Even if the models are available, there is an accuracy versus efficiency tradeoff. The most accurate power analysis tools work at the lower levels of the design, switch level or circuit level (12,13). These tools, however, are slow and impractical for analyzing the total power consumption of a processor as it executes *entire* programs. More efficient tools work at the higher levels, register transfer or architectural, but these are limited in the accuracy of the estimates.

Current Measurement

The previous problems are overcome if the current drawn by the processor during the execution of a program is physically measured. The instantaneous current drawn by the processor is a time-dependent quantity that varies rapidly. The current waveform peaks following a clock edge and then goes down until the next clock edge. To capture this waveform, the current measurement system has to sample the current many times during a cycle. For a cycle time of a few nanoseconds, very elaborate and expensive real-time data acquisition systems are required. However, as can be seen from the definition of average power, the average current drawn by the processor must be measured. For this, a much simpler and cheaper alternative method can be used.

The basic idea is illustrated in Fig. 5. The power supply connection to the processor is isolated from the rest of the system. Then an ammeter (a current measuring device) is inserted in series with the power supply and the processor. The ammeter is a standard off-the-shelf, digital ammeter.

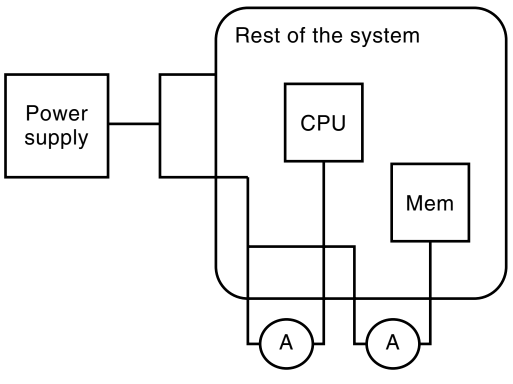


Figure 5. Experimental setup for current measurement.

Now if the given program is executed in a short time, a current reading cannot be visually obtained from the ammeter. To overcome this, the programs being considered are put in infinite loops. The current waveform is now periodic. Digital ammeters average current over a window of time (e.g., 100 ms), called the integrating window. If the period of the current waveform is much smaller than the integrating window, almost the same number of loop iterations occur within consecutive windows. Thus, the average current reading displayed by the ammeter will be stable.

The limitation of this approach is that it cannot be used directly for large programs, because the average current value continuously varies from one window to the next. However, this is not a limitation, because the main use of the measurement technique is for an instructional level power analysis. As discussed in the next section, short loops are adequate for this. This inexpensive current measurement approach works very well here and has been validated on three commercial processors (14). It should be stressed again, however, that the main concepts in this article are independent of the method for measuring average current.

INSTRUCTIONAL LEVEL POWER MODELS

The previous section presented feasible methods for measuring the power cost of entire programs. But these methods can also be used to obtain the finer level of resolution needed to analyze the power impact of individual instructions. By measuring the current drawn by the processor as it repeatedly executes certain instructions or certain short instruction sequences, it is possible to obtain most of the information needed to evaluate the power cost of a program for that processor.

The intuition behind this idea is as follows. Execution of each instruction in the instruction set involves specific processing across various units of the processor, which is characteristic of that instruction. If the processor is executing the same instruction over and over again, it seems intuitive that the entire activity in the processor can be attributed to the basic processing required to execute that instruction. The average current drawn by the processor, then, is the basic current cost for executing that instruction. In real programs, there may be other effects that affect the processor's power consumption. These include pipeline stalls, cache misses, and the effect of change in circuit state when consecutive instructions differ from one another. These effects involve more than one instruction, but executing programs where these effects occur repeatedly also provides a way of isolating the power costs of these effects.

This idea has been empirically verified for three commercial processors (14). The power costs of individual instructions and interinstructional effects are obtained by experiments which involve creating specific programs and measuring the current drawn during their execution. These costs are the basic parameters that define the instructional level power model of a given processor, as described following.

Instructional Base Costs

The primary component of the power models is the set of base costs of instructions. The experimental procedure for determining these costs requires a program containing a loop consisting of several instances of the given instruction. The average current drawn during the execution of this loop is measured. The product of this current and V_{dd} is the base power cost of the instruction. The base power cost multiplied by the number of nonoverlapped cycles needed to execute the instruction is proportional to its base energy cost. Table 1 presents a sample of the base costs of some instructions for the Intel 486DX2 and the Fujitsu SPARClite '934. The measured average current I, number of cycles N, and the base energy costs are also shown. The base energy costs are derived from the formula in the previous section.

The following are some points to be noted for assigning base costs to instructions:

- The definition of base costs follows the convention that the base costs of instructions should not reflect the power contribution of effects like stalls and cache misses. The programs for determining the base costs must be designed to avoid these effects. The power costs of these effects are modeled separately.
- Generally, instructions with similar functionality have similar base costs. Thus, similar instructions can be arranged in classes, and a single average cost can be assigned to each class.
- The base cost of an instruction varies with the value and address of the operands used. Although appropriate measurement experiments give the exact cost if the operand and address values are known, in real applications these values are often unknown until run time. The alternative is to assign a single average cost as the base cost for an instruction type. This is justified, because it has been observed that the variation in operands leads only to a limited variation in base costs.

Effect of Circuit State

The switching activity and, hence, the power consumption in a circuit is a function of the change in circuit state resulting from changes in two consecutive sets of inputs. Now, during the determination of base costs, the same instruction executes each time. Thus, it is expected that the change in circuit state between instructions would be less here than in an instructional sequence in which consecutive instructions differ from one another. This is confirmed by the fact that, for a sequence consisting of a mix of instructions, the actual cost is never less than the estimate obtained by using the base costs. For example, consider a loop of the following pair of instructions for the 486DX2:

```
XOR BX,1
ADD AX,DX
```

The base costs of the XOR and ADD instructions are 319.2 and 313.6 mA respectively. The expected base cost of the pair, using the individual base costs, is their average, that is, 316.4 mA, whereas the actual cost is 323.2 mA, greater by 6.8 mA.

The concept of circuit state overhead for a pair of instructions deals with this effect. Given any two instructions, the current for a loop consisting of an alternating sequence of these instructions is measured. The difference between the measured current and the average base costs of the two instructions is defined as the circuit state overhead for the pair. Adding in the average circuit state overhead for each pair of consecutive instructions in an instruction sequence leads to a much closer power estimate than using base costs alone.

Although this effect was observed for all of the processors studied so far, limited impact is reported on the 486DX2 and the Fujitsu '934 (14). The explanation for the limited impact may lie in the fact that, in large complex processors, a major part of the circuit activity is common to all instructions, for example, the clocks, instruction prefetch, memory management, pipeline control. Circuit state certainly results in significant variation within certain control and data path modules. But the impact of the variation on the net power consumption of the processor is masked by the much larger common cost. It should also follow from the above that, if instruction control and the data path constitute a larger fraction of silicon, the impact of the circuit state should be more visible. This indeed happens for the digital signal processor (DSP), a smaller, more basic processor.

Other Interinstructional Effects

The final component of the power model is the power cost of other interinstructional effects that occur in real programs. These include pipeline stalls and cache misses. Base costs of instructions do not reflect the impact of these interinstructional effects. Separate costs must be assigned to these effects through specific current measurement experiments. The basic idea is to write programs where these effects occur repeat-

Table 1. Sample Base Costs for the Intel 486DX2 and the Fujitsu SPARClite '934

	Intel 486DX2				Fujitsu SPARClite '934			
No.	Instruction	I, mA	N	Energy, 10^{-8} J	Instruction	I, mA	N	Energy, 10^{-8} J
1	`nop`	276	1	2.27	`nop`	198	1	3.26
2	`mov dx,[bx]`	428	1	3.53	`ld [%10],%i0`	213	1	3.51
3	`mov dx,bx`	302	1	2.49	`or %g0,%i0,%10`	198	1	3.26
4	`mov [bx],dx`	522	1	4.30	`st %i0,[%10]`	346	2	11.4
5	`add dx,bx`	314	1	2.59	`add %i0,%o0,%10`	199	1	3.28
6	`add dx,[bx]`	400	2	6.60	`mul %g0,%r29,%r27`	198	1	3.26
7	`jmp (taken)`	373	3	9.23	`srl %i0,1,%10`	197	1	3.25

edly. This helps to isolate the power costs of these effects. [For example, for the data cache in the 486DX2 (size 8K, four-way set associative, 16 byte block size), a sequence of five memory accesses to the addresses 0*H*, 800*H*, 1000*H*, 1800*H*, 2000*H* causes a cache miss each time.] Multiplying the power cost of each kind of stall or cache miss by the number of cycles taken for each gives the energy cost of these effects.

SOFTWARE POWER ESTIMATING METHODOLOGY

The basic components of the instructional level power models of typical processors were previously described. These models form the basis for estimating the energy cost of entire programs. For any given program *P*, its overall energy cost E_P is given by

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \qquad (2)$$

The base cost B_i of each instruction *i*, weighted by the number of times N_i, it is executed, is added up to give the base cost of the program. To this, the circuit state overhead $O_{i,j}$ for each pair of consecutive instructions (i, j), weighted by the number of times $N_{i,j}$ the pair is executed, is added. The energy contribution E_k of the other interinstructional effects *k* (stalls and cache misses) that occur during the execution of the program is finally added.

The base cost and overhead values are obtained as described in the previous sections. The other parameters in the previous formula vary from program to program. The execution counts N_i and $N_{i,j}$ depend on the execution path of the program. This is dynamic, run-time information that must be obtained by some of the analytical techniques for software performance described in the first part of this article. In certain cases it can be determined statically but in general it is best obtained from a program profiler. To estimate E_k, the number of pipeline stalls and cache misses must be determined. Again this is dynamic information that is statistically predictable only in certain cases. In general, this information is obtained from a program profiler and cache simulator.

The 486DX2 program shown in Table 2 illustrates the basic elements of the estimating process. The program has three basic blocks shown in the table. The average current and the number of cycles for each instruction are provided in two separate columns. For each basic block, the two columns are multiplied and the products are summed up over all instructions in the basic block. This yields a value proportional to the base energy cost of one instance of the basic block. The cost of the j1 L2 statement is not included in the cost of B2, because its cost is different depending on whether the jump is taken or not. The jump is taken three times and not taken once. B1 is executed once, B2 four times, and B3 once. Multiplying the base cost of each basic block by the number of times it is executed and adding the cost of the unconditional jump j1 L2, a number proportional to the total energy cost of the program is obtained. Dividing it by the estimated number of cycles (72) gives an average current estimate of 369.1 mA. Adding in the circuit state overhead offset value of 15.0 mA (a constant value is used for the circuit state overhead, because of the limited variation in this quantity for the 486DX2) gives 384.0 mA. This program does not have any stalls, and thus, no further additions to the estimated cost are required.

Table 2. Illustration of the Estimating Process

Program	Current, mA	Cycles
`; Block B1`		
`main:`		
`mov bp,sp`	285.0	1
`sub sp,4`	309.0	1
`mov dx,0`	309.8	1
`mov word ptr -4[bp],0`	404.8	2
`;Block B2`		
`L2:`		
`mov si,word ptr -4[bp]`	433.4	1
`add si,si`	309.0	1
`add si,si`	309.0	1
`mov bx,dx`	285.0	1
`mov cx,word ptr _a[si]`	433.4	1
`add bx,cx`	309.0	1
`mov si,word ptr _b[si]`	433.4	1
`add bx,si`	309.0	1
`mov dx,bx`	285.0	1
`mov di,word ptr -4[bp]`	433.4	1
`inc di, 1`	297.0	1
`mov word ptr -4[bp],di`	560.1	1
`cmp di,4`	313.1	1
`j1 L2`	405.7(356.9)	3(1)
`;Block B3`		
`L1:`		
`mov word ptr _sum,dx`	521.7	1
`mov sp,bp`	285.0	1

If some cache misses are expected in the actual execution of this program, their energy overhead has to be added. The actual measured average current is 385.0 mA.

When average values are used for base costs, etc., the accuracy of the energy estimate given by the model described in Eq. (1) is limited to some extent by the range of variation in the average and the actual costs. However, the accuracy of the energy estimate is primarily limited by the accuracy in determining the dynamic information for the program. In certain applications, for example, speech processing, some statistical characteristics of the input data are known (15). Incorporating this knowledge into the power model leads to more accurate power estimates. This is specially beneficial for smaller processors, such as DSPs, which are more sensitive to data-based power variations than large general purpose processors.

Overall Flow

A typical overall flow of the estimating procedure is illustrated in Fig. 6. Given an assembly or machine level program, it is first split up into basic blocks. The base cost of each instance of the basic block is determined by adding up the base costs of the instructions in the block. These costs are provided in a base cost table. The circuit state overhead is determined for each basic block from a table of energy costs for pairs of instructions. (Alternatively, if the circuit state overhead does not show much variation for a processor, a constant value is used.) The energy overhead due to pipeline, write buffer, and other stalls is estimated for each basic block and added to the basic block cost. Next, the number of times each basic block is executed has to be determined. This depends on the path that the program follows and is obtained by dynamic path

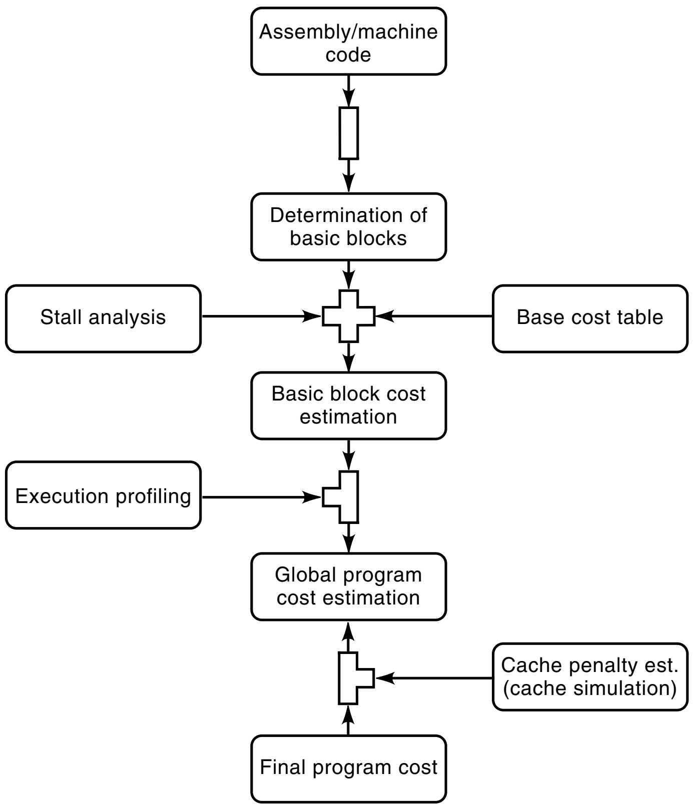

Assembly/machine code
Determination of basic blocks
Stall analysis
Base cost table
Basic block cost estimation
Execution profiling
Global program cost estimation
Cache penalty est. (cache simulation)
Final program cost

Figure 6. Software energy estimating methodology.

analysis techniques, such as program profiling. Given this information, each basic block is multiplied by the number of times it is executed. An estimated cache penalty is added for the final estimate. The cache penalty overhead computation needs an estimate of the miss ratio, obtained through memory performance analysis techniques, such as cache simulation.

Idle Time Evaluation

The previous discussion is for an active processor constantly executing instructions. However, a processor is not always performing useful work during program execution. For example, during the execution of a word processing program, the processor may simply be waiting for keyboard input from the user. If it executes a busy wait, that is, executes a loop of instructions while waiting, it is wasting power. A power reduction technique that is gaining popularity in systems is to put the processor in a low-power state during such “idle” periods.

To account for these low-power periods, the average power cost of a program is thus given by: $\mathscr{P}_{\text{active}} \times \tau_{\text{active}} + \mathscr{P}_{lp} \times \tau_{lp}$, where $\mathscr{P}_{\text{active}}$ is the average power consumption when the processor is active and τ_{active} is the fraction of the time during which the processor is active. $\mathscr{P}_{lp}$ and τ_{lp} are the corresponding parameters for the time when the processor is idle and is put in a low-power state. τ_{active} and τ_{lp} are determined by dynamic performance analysis techniques, such as those described in the first part of this article. The Intel Power Monitor is a commercial tool that provides this information directly for systems based on Intel’s processors (16).

Memory Power Modeling

In addition to the processor power consumption, the software execution also results in power consumption by the memory system. This can be a significant part of the total power consumption of the system. In addition, this consumption is also a function of the software being executed and thus merits similar modeling. The current drawn by the memory system is measured in the same way as for the processor. Then, by executing programs that repeatedly access memory, the memory system power consumption is analyzed. [Details of such a study for a dynamic random access memory (DRAM) chip are presented in Ref. 17.]

FUTURE DIRECTIONS

The concepts in this article can form the basis for further work in software power evaluation. Certain important future directions are presented here for the interested reader.

Software Power Profilers

An interesting extension of the software power estimating methodology described previously is the development of power profilers for given processors. The instructional level power model described here suggests that this is easily done by enhancing existing performance-based profilers with the power costs of instructions and interinstructional effects. Using this data, the profilers generate a cycle-by-cycle profile of the power consumption of given programs.

Impact of Dynamic Power Management

Dynamic power management, the shutting down of modules not needed for a given computation, is gaining popularity as a power reduction technique. An aggressive application of dynamic power management inside a processor has interesting ramifications for the instructional level power analysis of the processor. In particular the following issues deserve further investigation: (1) the base costs of different instructions may show greater variation; (2) variations due to differences in operand data values may increase; (3) the overall reduction in power may also make the effect of circuit state overhead more prominent; and (4) some power management features may be activated depending on the occurrence of specific sequences of instructions, and this may require special handling.

Power Analysis for Newer Architectures

It is important to extend the instructional level power analysis methodology to all major processor architectures. In particular, analysis of processors based on superscalar and very large instruction word (VLIW) architectures is especially important. These seem to be the architectures of choice for high-performance processors in the near future, and, with ever increasing integration and clock frequencies, the power problem will become even more acute for these processors.

BIBLIOGRAPHY

1. G. J. E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms,* New York: Computer Science Press, 1991.
2. P. Puschner and C. Koza, Calculating the maximum execution time of real-time programs, *J. Real-Time Syst.,* **1**: 160–176, 1989.

3. J. Larus and T. Ball, Rewriting executable files to measure program behavior, *Software, Practice and Experience,* **24** (2): 197–218, 1994.
4. MIPS Computer Systems, Inc. *UMIPS-V Reference Manual (pixie and pixstats),* 1990.
5. VTune home page. Intel Corporation, Santa Clara, CA, [online]. Available http://developer.intel.com/design/perftool/vtune/.
6. M. Martonosi, A. Gupta, and T. Anderson, Tuning memory performance in sequential and parallel programs, *IEEE Computer,* **28** (4): 32–40, 1995.
7. A. Lebeck and D. Wood, Cache profiling and the SPEC benchmarks: A case study, *IEEE Computer,* **27** (10): 15–26, 1994.
8. A. Shaw, Reasoning about time in higher-level language software, *IEEE Trans. Softw. Eng.,* **15**: 875–889, 1989.
9. C. Park, Predicting deterministic execution times of real-time programs, Ph.D. Thesis, Univ. of Washington, Seattle, WA, 1992.
10. Y.–T. Li, Performance analysis of embedded software, Ph.D. Thesis, Princeton Univ., Princeton, NJ, 1997.
11. S. Malik, M. Martonosi, and Y.–T. Li, Static timing analysis of embedded software, *Proc. Des. Automation Conf.,* June 1997, pp. 147–152.
12. C. X. Huang et al., The design and implementation of PowerMill, *Proc. Int. Symp. Low Power Des.,* April 1995, pp. 105–110.
13. L. Nagle, SPICE2: A computer program to simulate semiconductor circuits, Tech. Rep. ERL-M520, Univ. of California, Berkeley, 1975.
14. V. Tiwari et al., Instruction level power analysis and optimization of software, *J. VLSI Signal Process. Syst.,* **13**: 223–238, 1996.
15. P. Landman, Low-power architectural design methodologies, Ph.D. Thesis, Univ. of California, Berkeley, 1994.
16. Intel power monitor home page. Intel Corporation, Santa Clara, CA, [online]. Available http://www.intel.com/ial/ipm/.
17. V. Tiwari, Logic and system design for low power consumption, Ph.D. Thesis, Princeton Univ., Princeton, NJ, 1996.

VIVEK TIWARI
Intel Corporation
SHARAD MALIK
Princeton University