Run-time Modeling and Estimation of Operating System Power Consumption

Tao Li
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas, 78712
tli3@ece.utexas.edu

Lizy Kurian John
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas, 78712
ljohn@ece.utexas.edu

ABSTRACT

The increasing constraints on power consumption in many computing systems point to the need for power modeling and estimation for all components of a system. The Operating System (OS) constitutes a major software component and dissipates a significant portion of total power in many modern application executions. Therefore, modeling OS power is imperative for accurate software power evaluation, as well as power management (e.g. dynamic thermal control and equal energy scheduling) in the light of OS-intensive workloads. This paper characterizes the power behavior of a commercial OS across a wide spectrum of applications to understand OS energy profiles and then proposes various models to cost-effectively estimate its run-time energy dissipation. The proposed models rely on a few simple parameters and have various degrees of complexity and accuracy. Experiments show that compared with cycle-accurate full-system simulation, the model can predict cumulative OS energy to within 1% accuracy for a set of benchmark programs evaluated on a high-end superscalar microprocessor. When applied to track run-time OS energy profiles, the proposed routine level OS power model offers superior accuracy than a simpler, flat OS power model, yielding per-routine estimation error of less than 6%. The most striking observation is the strong correlation between power consumption and the instructions per cycle (IPC) during OS routine executions. Since tools and methodology to measure IPC exist on modern microprocessors, the proposed models can estimate OS power for run-time dynamic thermal and energy management.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development - *modeling methodologies*

General Terms

Measurement, Performance, Design.

Keywords

Power estimation, operating system, low power.

SIGMETRICS'03, June 10-14, 2003, San Diego, California, USA.

1. INTRODUCTION

The increasing concern on power issues in many computing systems points to the need for the power modeling and estimation for all components of a system. Software, which presents in forms of both the operating system (OS) and the user applications, constitutes a major component of today's systems implemented with high-end and general-purpose microprocessors. Software execution drives the activities of the underlying hardware and the manner in which software uses hardware can have a substantial impact on the power dissipation of a system [3]. For example, a 2 GHz Intel Pentium-4 microprocessor fabricated with 0.13 μm technology can consume 60-Watt power on an integer multiplication, whereas the power dissipated to execute a HALT instruction can be as low as a few Watts [11]. Previous studies [29, 24] also observed that the choice of algorithm and other higher-level decisions during the design of software components could measurably affect system power. Therefore, it is becoming crucial to model power consumption from the perspective of the software.

Many modern and emerging applications (e.g. database, file/e-mail servers) constitute an important software domain and exercise operating system significantly. The OS not only occupies a significant portion of machine cycles but also can consume the dominant part of total energy. For instance, Figure 1 shows the percentage of total energy (microprocessor and memory subsystems) dissipated by the OS on the experimented applications (see Section 2 for details). Overlooking the OS effect can cause significant software energy estimation error. The proportion of the OS energy overhead is continuously increasing due the emerging system administrative activities, such as thermal sensor reading [12], energy accounting [3] and low power mode control for memory and I/O devices [13, 34].

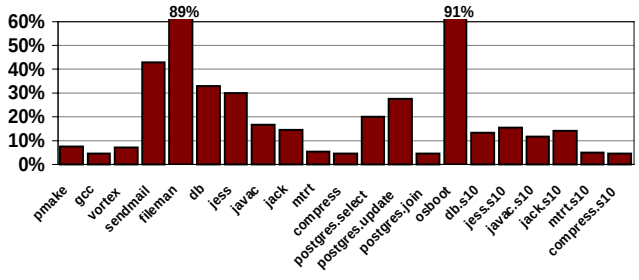


Figure 1. % of Energy Dissipated by the OS on the Experimented Applications

Therefore, accounting OS power is imperative for accurate software power modeling in the light of OS-intensive workloads. In the power-constrained systems, such information is useful for programmers to explore the design space for energy-efficient software and to assure that overall software (system and application) power meets the specified budget. Moreover, OS power estimation can help to quantify how one optimization affects the complete system behavior from the perspective of energy consideration. For example, software engineers may be interested in knowing whether compiler techniques such as code and data re-layout [32] can help with saving run-time OS power caused by handling TLB misses and paging.

Many high-level power management and optimizations require the just-in-time, accurate and complete (including the OS) energy profile to finely tune the performance/power knob during program execution. As shown in [6, 10], hardware and software strategies can be used to implement dynamic thermal management (DTM) to reduce the chip's temperature operated by a particular workload. An on-line and accurate power estimator is imperative to make it feasible and to further alleviate the negative impact of the DTM. In mobile computing environments where energy is precious, to extend battery life, applications can run in degraded QoS modes [34], provided the available battery energy is below a given threshold. In this scenario, software power estimator can be used to account the available battery energy quota for a given application and to adjust its QoS modes accordingly. Without the reliable run-time OS power models, the fidelity of such adaptations is difficult to guarantee.

In the past, the energy behavior of embedded and real-time operating systems has been evaluated [8, 2, 27]. In [9], a full-system energy-aware simulator is developed and the necessity of simulating OS energy is quantified. Nevertheless, the issue of just-in-time OS power estimation on high-performance and general-purpose microprocessors has not been addressed so far. These motivate the need for the accurate and efficient run-time OS power estimation - something that traditional software power modeling techniques are inadequate or not quite suited for.

Several software power estimation methodologies [29, 26, 12, 5, 9] have been proposed in the past literature. In the following subsections, we briefly describe various styles for software power estimation and discuss the challenges of modeling OS power that motivates the new approaches.

1.1 Software Power Estimation Techniques

In microprocessor-based systems, one can model power dissipation as a function of the software (instructions) being executed on the underlying hardware platforms. Software power estimation techniques from past literature can be sorted into the following four categories:

1.1.1 Instruction Level Power Modeling

The instruction level power modeling [29] has been proposed to evaluate the power dissipation of a given piece of software. The basic idea is to explicitly associate the consumed power with individual instruction execution. An instruction level software power model can be generally described as:

$$E = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k S_k \quad (1),$$

where B_i is the base energy cost to process the individual instruction i. $O_{i,j}$ reflects the dissipated power due to the circuit switching between each pair of consecutively executed instructions (i, j). The term S_k accounts for other energy overhead due to the *k*-types of inter-instruction effects, such as write buffer stalls and cache misses. For a given program, its overall energy cost, E, can then be calculated by multiplying the B_i and the $O_{i,j}$ with the dynamic instances of the individual instruction (N_i) and the instruction pair ($N_{i,j}$) correspondingly.

To get B_i and $O_{i,j}$, an exhaustive power characterization of the entire ISA (Instruction Set Architecture) and an inter-instruction effects measurement for any possible instruction pairs have to be conducted. For example, for the Intel IA-32 ISA [11] with 331 unique instructions, the number of possible instruction pairs need to be measured are 109,561 (331^2), which makes the instruction level power characterization effort non-trivial.

To compute power dissipation, the above methodology favors an off-line analysis of the complete trace of the program. Although it is feasible to produce and store complete instruction traces for the simple and embedded software, the volumes of complete instruction traces from large applications would easily overwhelm the disk space. It should be noticed that the on-the-fly traces could be generated and analyzed by employing ad hoc methodology. For example, a special linker modifies the compiler-generated code to insert calls to the tracing and power analyzing routines at each instruction and at the entry of each basic block. However, programs traced in this way are much larger than normal and take longer to execute than normal. Additionally, without significantly merging, approximation and therefore paying the cost of accuracy lost, it is infeasible to fit all the B_i and $O_{i,j}$ into a small (hardware) table for a live, just-in-time power estimation, a feature which is imperative to support many run-time power management. One solution is to store the B_i and $O_{i,j}$ into a software-based table and uses a dedicated software trap to trigger table lookup and then compute power consumption. Unfortunately, this scheme can also significantly dilate the execution time of an estimated program, due to the overhead of the software trap handler and its invocations at individual instruction (or instruction sequence) granularity. Therefore, run-time instruction level power modeling is intrusive and computation intensive.

1.1.2 Characterization-based Macro-modeling

Instead of evaluating power at instruction level, software function level macro-modeling techniques [26, 21] treat application functions or sub-routines as "black boxes" and construct macro-models that correlate power with a set of characteristics of interest. Such power characteristics of interest can be obtained and collected by using a low-level energy simulation framework [28]. Under this philosophy, a software function or sub-routine's power template can be represented by a linear formula with respective to the *n* power interest metrics $[c_1, c_2, ..., c_n]$ as:

$$P = \sum_j w_j \times c_j \quad (2),$$

where $[w_1, w_2, ..., w_n]$ are the macro-modeling coefficients to be determined. Regression analysis is then applied to identify the optimal $[w_1, w_2, ..., w_n]$ with the least mean square fitting error based on a set of known input and output pairs.

The key issue on the above macro-modeling is how to choose $[c_1, c_2, ..., c_n]$, which can effectively capture the power characteristics of a given software sub-routine under various circumstances. In [26], Tan et al. suggested the use of algorithm complexity and trace-based basic-block correlation information as the power metrics. These techniques are proposed for embedded software and targeted for embedded processors. It should be noticed that while embedded software like the DSP kernels have more intensive and regular looping patterns, the operating systems which are designed to manage both software and hardware systems can lead to far more complicated and unpredictable control flow [14, 15] that can not be easily captured by a naive metric such as algorithm complexity. The trace-based basic-block correlation analysis is more suitable for processors that execute instruction in order [17]. The data dependency and speculative execution effects have a more significant impact and greater variation in the case of wide-issue and deeply pipelined superscalar processors. For example, even for exactly the same input data set, speculative execution along the wrong path followed by a mispredicted branch will cause more energy dissipation compared with the scenario that has the correctly predicted control flow [16].

On the other hand, the use of basic-block correlation metric relies on storing complete control flow graph (CFG) for each software sub-routine and counting the number of each correlated path whenever that sub-routine is invoked. Like instruction level power modeling, this macro-modeling technique necessitates off-line trace analysis because finding basic-blocks and counting correlated paths will be computation intensive and intrusive to the estimated software execution when they are applied to the on-line power estimation. The feature of just-in-time power modeling necessitates the use of simpler metrics.

1.1.3 Performance Counter-based Run-time Power Estimation

Run-time software power estimation [12, 3] derives an estimate of live power dissipation by leveraging the existing processor hardware and an analytical power model of the target microprocessor. The idea is that the amount of power dissipated on software execution is appropriate to the amount of accesses and switching activities within processor units. Most modern microprocessors have already embedded programmable event counters [4] to monitor microarchitectural events for the performance measurement purpose. Heuristics can be chosen from the available counters to infer power relevant events and further feed to an analytical processor power model to calculate the power.

Joseph et al. [12] showed that the performance counters can be quite useful in providing good power estimation for programs as they run. Considering about 12 performance measures, they estimated power within 2% of the actual power. However, in general and for a given processor, the availability of heuristics is limited by the types of the performance counters and the number of events that can be measured simultaneously. For example, the Alpha 21264 has only 9 performance counters and the Intel Pentium III processor can only simultaneously observe 2 out of the 77 total events. Operating systems and many large software are non-deterministic in nature and their behavior can vary significantly over time and different runs [1]. Therefore, random sampling of counters with different configured event types does not apply to the on-line OS energy profiling. On the other hand, due to the "black box" power modeling approaches taken in [12, 3], fine-grained (e.g. function level) power distribution, which provides insight into the software power behavior, is not available. Meanwhile, due to the observed drastic phase changes during application execution [23], the accuracy of using a simpler, flat model to track the run-time software power behavior is largely unknown.

1.1.4 Cycle-accurate Architectural Level Simulation

It has been widely accepted that circuit and gate level simulations are infeasible to evaluate power consumption of large software executing on complex computing systems. A complementary set of approaches is based on the use of cycle-accurate architectural level power simulators [5, 33, 9]. Architectural level power simulations have been shown to be applicable to modern superscalar processor (with deep pipelines, out-of-order and speculative execution). However, cycle-accurate simulation causes simulation speed to be extremely slow, preventing the efficiency of the design space searching. This is especially true when simulating large and complex applications using detailed processor models. Because of that, simulation based power model can not be used to support run-time software power estimation.

Moreover, most of the existing architectural level power simulators (e.g. Wattch [5] and SimplePower [33]) do not include the effect of the OS in their software power analysis. The OS execution can either be invoked explicitly (e.g. system calls) or implicitly (e.g. paging and faults handling) and the occurrence of the OS execution can be either synchronous (e.g. timer interrupt) or asynchronous (e.g. scheduling). Therefore, the power dissipation of OS due to its run-time, exception-driven and non-deterministic nature can not be completely captured without using a power-aware, timing-accurate and full-system simulation framework. In [9, 27, 7], such full-system energy simulators are developed and the necessity of simulating OS energy is quantified. Detailed and full-system simulation further suffers from potentially long run times when simulating complete system activities using complicated processor, memory and I/O device modules.

1.2 Challenges in OS Power Modeling

For an OS power estimation technique to be applicable to run-time thermal/power management, it must have the following properties:

- *High fidelity and fast speed*: The model should be able to estimate the OS energy dissipation accurately. Power estimation should avoid the extremely slow cycle by cycle full-system simulation as much as possible.

- *Run-time estimation capability, non-intrusive and low overhead*: The model should support on-the-fly OS power estimation. The run-time power estimation overhead should be low to avoid disturbing the normal OS execution.

- *Simplicity, availability and generality*: The model should only rely on a few power metrics of interest that is widely available across different hardware platforms.

The goal of this work is to develop a methodology that possesses the above merits.

1.3 Paper Overview

This paper explores techniques to efficiently estimate OS power dissipation while providing the valuable features discussed in Section 1.2. The observation is that in a given computing system, OS is a commonly used software layer exercised by all applications. OS power dissipation is usually dominated by a set of limited but heavily invoked kernel service routines. Just as instructions are the fundamental units of software execution, the OS service routines can be though as the fundamental unit of OS execution. Provided that the most frequently invoked OS service routines have the similar or predictable power dissipation behavior across various benchmarks, we can evaluate the power characteristics of these OS routines and use such information to derive the aggregrate OS power consumption across various applications. Furthermore, we discover a strong correlation between power and the Instruction Per Cycle (IPC) metric during execution. We develop a simple model that exploits this correlation.

OS routine based power characterization and estimation thus avoid the computationally expensive full-system simulation for each estimated application. Combined with existing performance estimation mechanisms (e.g. microprocessor hardware performance counters), the proposed OS routine level power model can lead to highly efficient and accurate run-time power modeling for the OS.

This paper is organized as follows: Section 2 describes the experimental framework, methodology and benchmarks. Section 3 provides routine level OS power characterization. Section 4 proposes the routine based OS power models and evaluates their estimation accuracies. Section 5 discusses the issues of applying the proposed model to run-time power estimation. Finally, Section 6 concludes with some final remarks and comments.

2. EXPERIMENTAL METHODOLOGY

To characterize OS power behavior at routine level, we use power-aware and full-system simulation driven by a wide range of OS-intensive applications. This section describes the simulation framework, the machine architecture modeled and the workloads executed.

2.1 Framework and System Configuration

We use the complete system power simulator SoftWatt [9] that models the power dissipation of the CPU, memory hierarchy and a low-power disk subsystem to investigate the power behavior of OS. The SoftWatt tool, built on top of the SimOS infrastructure [22], uses validated energy models similar to other low-level power simulators like Wattch [5]. By leveraging the SimOS cycle-accurate and full-system simulation capability, SoftWatt captures power dissipation of both applications and OS running on a detailed system model. The simulated OS is a commercial version of the SGI IRIX 5.3.

Table 1 gives the target system configuration of SoftWatt that is used for our experiments. The simulated processor is a 8-way issue, out-of-order superscalar with function unit latency like MIPS R10000. The CPU model runs at 900 MHz on 2.0 V supply voltage and uses 0.18 micron processing technology. The memory hierarchy includes separate L1 data and instruction caches, unified L2 cache and multiple-banked main memory. The disk model is a SCSI HP97560 incorporated with low power feature.

Table 1. Baseline Machine

Processor Core	
Technology/V_{dd}/Frequency	0.18 um/2.0V/900 Mhz
Fetch/Issue/Retire Width	8
Instruction Window Size	128
Reorder Buffer Size	128
Number and Latency of Function Units	MIPS R10000 Like
Branch Target Buffer	2048-entry, 4-way
Return Address Stack	32-entry w/ misprediction repair
Branch Prediction/Penalty	8K-entry OS-aware Gshare/10 Cycles[15]
Load Store Queue Size	64
Memory Hierarchy	
MMU	Fully associative TLB, 48-entries, 4KB page size
L1 I-Cache	32KB, 2-way(LRU), 64B blocks, 4MSHRs, 2 ports, 1 cycle latency
L1 D-Cache	32KB, 2-way(LRU), 32B blocks, 4MSHRs, 2 ports, 1 cycle latency
L2 Cache	512KB, 2-way(LRU), 128B blocks, 4MSHRs, 2 ports, 9 cycle latency
Memory	256MB, 4 banks, 180 cycle access
I/O	
SCSI Disk	Scaled HP97560 incorporated with low power feature

2.2 Benchmarks

Table 2. Benchmarks

	Name	Description
Test	*sendmail*	UNIX electronic mail transport agent
	fileman	File management
	db	Performs multiple database functions
	jess	Java expert shell system
	postgres.select	DBMS PostgreSQL executes a select query
	postgres.update	DBMS PostgreSQL executes an update query
	osboot	A complete OS boot sequence
Profiling	*pmake*	Two parallel compilation processes
	gcc	Compiles pre-processed source
	vortex	A full object oriented database
	javac	The JDK 1.0.2 Java compiler
	jack	Parser generator with lexical analysis
	mtrt	Dual-threaded raytracer
	compress	Compress and decompress large file
	postgres.join	DBMS PostgreSQL executes a join query
	db.s10	*db* executes S10 dataset
	jess.s10	*jess* executes S10 dataset
	javac.s10	*javac* executes S10 dataset
	jack.s10	*jack* executes S10 dataset
	mtrt.s10	*mtrt* executes S10 dataset
	compress.s10	*compress* executes S10 dataset

We use 21 applications (see Table 2) that have different characteristics. This section provides a brief overview of the selected applications. *Db*, *jess*, *javac*, *jack*, *mtrt* and *compress* are Java programs from the SPECjvm98 suite [25] executed with s1 dataset on a SGI ported Sun Java virtual machine. The above 6 applications are also run with different dataset (s10) to provide more profiling information. *Vortex* and *gcc* are two programs from the SPECint95 benchmarks. *Pmake* is a parallel program

development workload [18]. The *sendmail* benchmark forwards emails using the Simple Mail Transport Protocol (SMTP). We also use three benchmarks that run on a relational database management system (DBMS) engine - PostgreSQL [20]. The database is populated with relational tables for the TPC-C benchmark [30]. The *postgres.select* performs a sequential table scan of a table with 1 million rows and a selectivity of 3%. The *postgres.update* updates to a field of a 300,000 row table and the *postgres.join* executes a nested loop join query involving two tables of sizes 11MB and 24KB. The *osboot* executes a complete OS booting sequence from a root disk image and then generates a shell for the root user. The *fileman* performs file management activities, such as copy, remove, tar -cvf and tar -xvf.

For the OS power modeling and estimation, the benchmarks are partitioned into two groups, namely, *profiling* and *test*, as shown in Table 2. The *profiling* group is used to generate data needed to build the models. The *test* group is used to examine the accuracy of the proposed models. The *test* group was selected to contain some of the programs that contain significant OS activity.

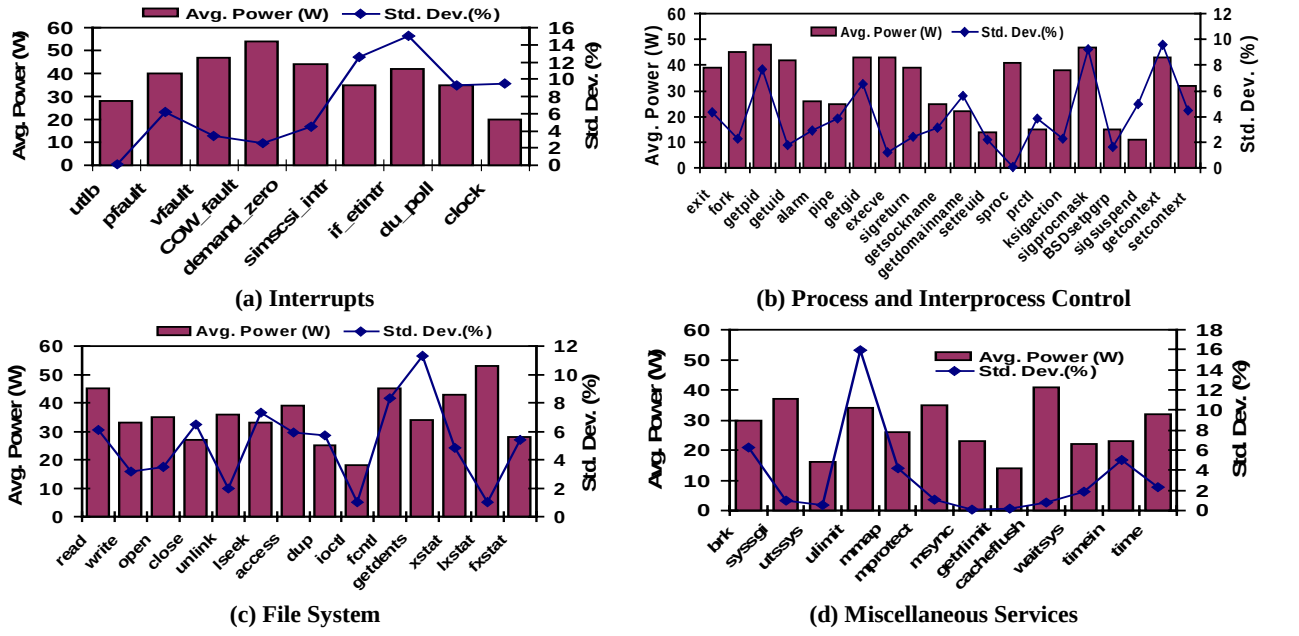
3. ROUTINE LEVEL OS POWER CHARACTERIZATION

Modern operating systems are complex software managing heterogeneous system resources. The complexities of OS are hidden behind a relatively simple interface - the OS kernel services. Thus to model the energy consumption of this complex system, it seems intuitive to consider individual OS service routine. Each OS service execution consumes a certain amount of power. The power consumed by the OS can be thought of as the aggregation of the power cost of each OS routine that is executed in the OS. This section presents a characterization of OS power behavior at kernel service routine level.

3.1 Power Behavior of OS Routines

We measure the average power and its standard deviation for each OS routine across different benchmarks. As shown in Figure 2, these OS routines are classified into interrupts, process and interprocess control, file system and miscellaneous services (see Appendix 1 for more information).

One can see that there can be a great variance in power consumption between different OS routines. For example, while the power dissipation on the OS copy-on-write fault handler *COW_fault* is as high as 54W, the *setreuid* routine (set real and effective user id) only consumes 14W of power. This implies that estimating the energy cost of various OS calls without resorting to detailed simulation will cause measurable error. Each OS service involves specific instruction processing across various units of the processor, which results in circuit activity that is characteristic of each OS service and can vary with OS services. Memory access intensive OS routines, such as *vfault*, *COW_fault*, *demand_zero*, *cacheflush* show higher power consumption than computation intensive services, such as *utlb* and *clock*. Some I/O interrupts (*simscsi_intr* and *if_etintr*), process scheduling (*getcontext*), file I/O (*fcntl*, *lseek* and *getdents*) show higher standard derivation in power consumption because their execution is largely dependent on system status. On the other hand, OS routines such as *utlb*, *utssys* and *cacheflush* perform certain amount of work in each invocation, resulting in negligible power consumption variation.



(a) Interrupts

(b) Process and Interprocess Control

(c) File System

(d) Miscellaneous Services

Figure 2. Average and Standard Deviations of OS Routines Power (Standard deviations indicated on the right side y-axis in each graph)

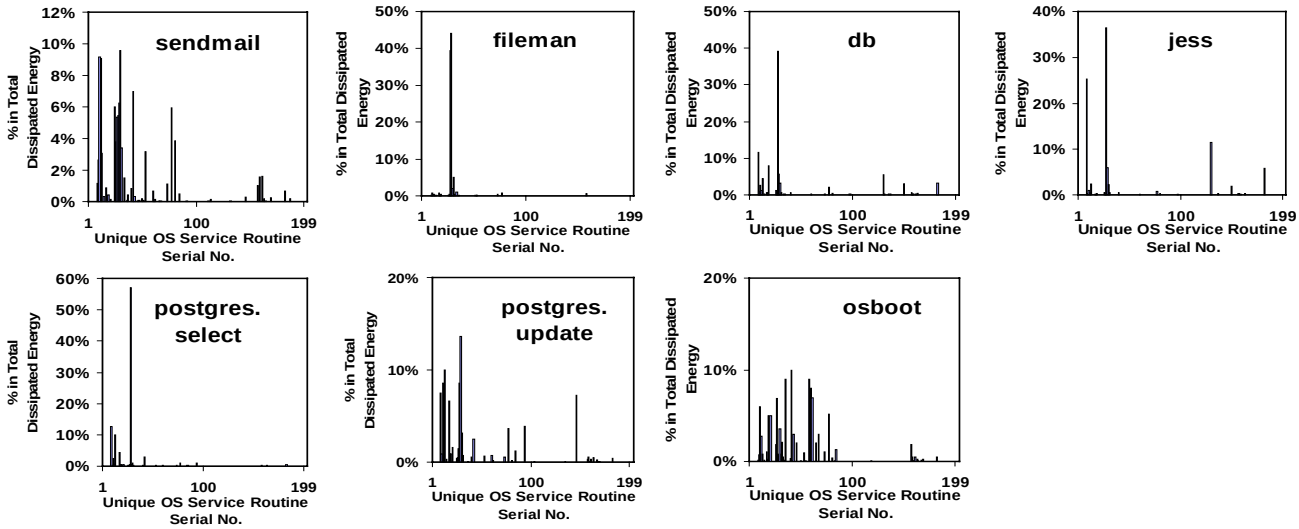


Figure 3. Routine Level Energy Distributions in OS

Figure 3 further reveals the run-time routine-level OS energy distribution across different benchmarks. The x-axis indicates the serial numbers of unique OS service routines and the y-axis shows the percentage of run-time OS energy dissipated by that specific OS routine. In this study, we identify a total number of 186 OS service routines. Figure 3 shows that different benchmarks invoke different OS services and hence show different energy distribution patterns. For example, on benchmarks *filename*, *db*, *jess* and *postgres.select*, the OS energy dissipation is dominated by a small fraction of highly invoked service routines while on benchmarks *sendmail*, *postgres.update* and *osboot*, OS energy consumption is contributed by a wide range of service routines. The above observation, combined with the fact that individual routine shows different power behavior, implies that: (1) overall, the OS power behavior can vary from one application to another; (2) the use of single "average OS power" number across various applications will lead to significant estimation errors.

3.2 Energy-Performance Correlation

Figure 4 further shows how a set of OS routine's power varies on different profiling benchmarks. In the cases of *utlb* and *cacheflush*, the OS power varies in a very restricted range. However, on *simscsi_intr*, the OS routine power can span with in a range from 8W to 59W. Interestingly, we observe that OS routine's power is strongly correlated with its performance. We investigate the use of IPC (Instructions per Cycle) as the metric to characterize the performance of modern processors, as pointed out in [19]. Valluri [31] and Chen [7] also had observed a similar correlation.

The explanation for this correlation lies in the fact that in a complex, high performance superscalar processor, a dominant portion of the power is consumed by circuits used to exploit the ILP (instruction level parallelism). The pie chart in Figure 5 shows how various components in the CPU and memory systems contribute to the total OS routine power. Data-path and pipeline structures, which support multiple issue and out-of-order execution, are found to consume 50% of total power on the examined OS routines. Figure 5 shows that clock is the second largest power consuming component: the capacitive load to the clock network switches on every clock tick, causing significant power consumption.

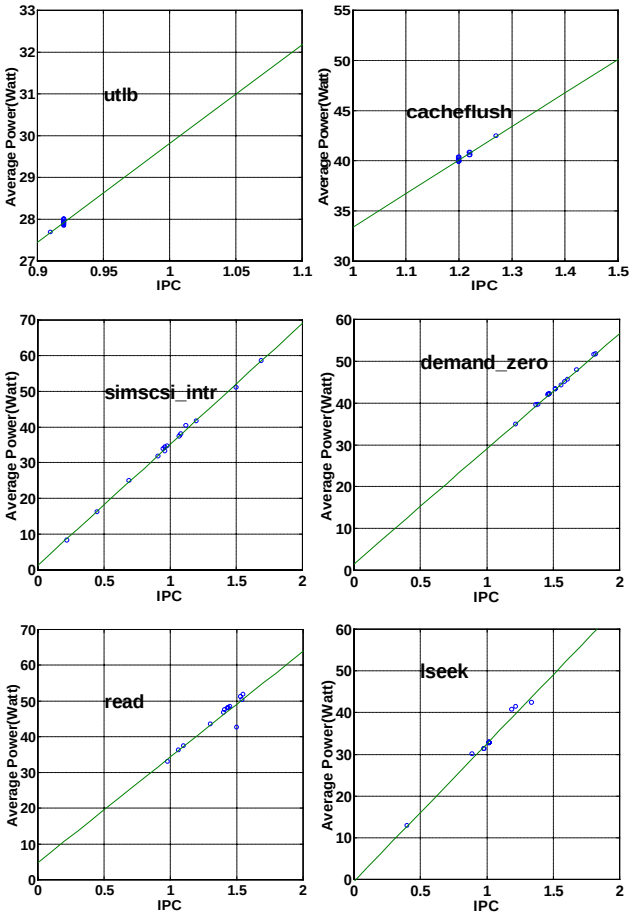


Figure 4. Correlation between OS Routines Power and IPC

The energy consumed in data-path during execution usually depends on the number of instructions that flow through. The ILP performance measured by IPC, certainly impacts circuit switching activities in those microprocessor components and can result in significant variation in power. High IPC reflects the scenario in which most of the processor structures are busy. On the other hand, main pipeline stalls or bubbles, which lead to low IPC and can be easily clock gated, will drastically reduce power dissipation. For a given piece of code, similar IPC usually indicates similar circuit switching activities and therefore, similar power consumption.

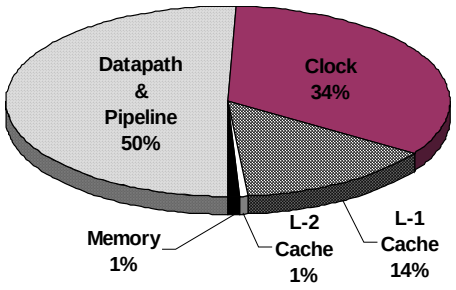


Figure 5. Breakdown of Power Dissipation of OS Routines shown in Figure 4

The above correlation implies that one can use a simple linear regression model

$$P = k_1 \times IPC + k_0 \quad (3),$$

to track the OS routine power showing different performance. Appendix 1 lists the regression model parameters (k_1, k_0) and the regression model fitting errors for the examined OS routines.

4. ROUTINE LEVEL OS POWER MODEL

In this section, we present routine level profiling based energy estimation models. The objective is to provide simple and easily computable techniques that can be used for run-time energy estimation of operating system software.

Energy consumption of a given piece of software can be estimated as: $E = P \times T$, where P is the average power and T is the execution time of that program. If average power of different OS routines can be determined, it can be used to compute the operating system energy. A routine level OS energy estimation model can be represented as:

$$E_{OS} = \sum_i (P_{os\_routine,i} \times T_{os\_routine,i}) \quad (4),$$

where $P_{os\_routine,i}$ is the power of the i_{th} OS routine invocation and $T_{os\_routine,i}$ is the execution time of that invocation.

The $P_{os\_routine,i}$ can be computed in many ways. It can be an average power based on all invocations of that routine in the programs (as shown in Figure 2). Figure 6 illustrates the accuracy of this estimation model. The profiling based average power values at the routine level are found to yield estimation errors within 5% in 6 out of the 7 *test* benchmarks. On benchmark *fileman*, however, this scheme can underestimate the OS power by as much as 32%.

Exploiting the interesting observation we presented in Section 3.2 on the correlation between IPC and OS routine average power, we investigate the potential of this correlation in estimating energy consumption of programs based on IPC. This approach is similar to the one used in [12], where approximately a dozen performance counters are used to estimate power. However, we only utilize 2 pieces of information here, namely, instruction count and cycles. Also, we use a profiling approach by which information based on some benchmarks can be used to predict the energy of a different application. To investigate the usefulness of this approach, we use per-routine based OS power models built on *profiling* benchmarks (Appendix 1) to estimate OS power on the *test* benchmarks. The accuracy of the energy estimation is within 1% (as illustrated in Figure 7).

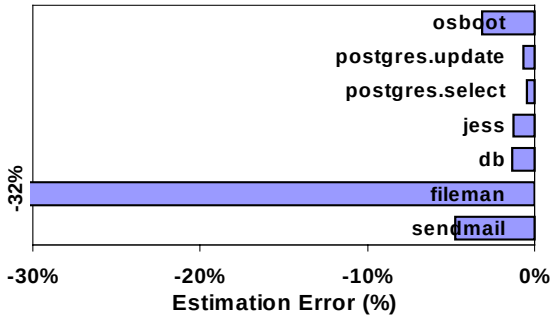


Figure 6. Model Estimation Accuracy (Routine Average Power)

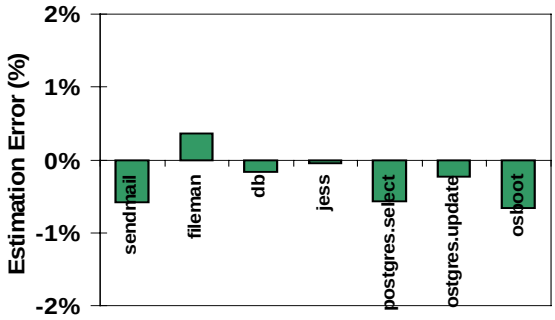


Figure 7. Model Estimation Accuracy (IPC Correlated Routine Average Power)

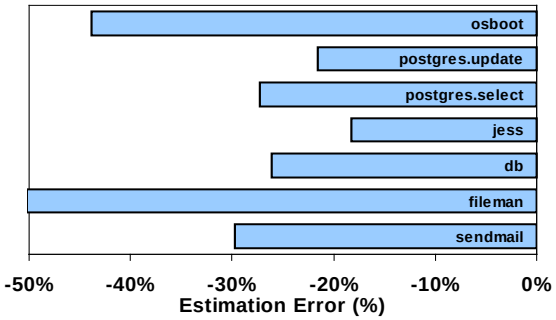


Figure 8. Model Estimation Accuracy (OS Average Power)

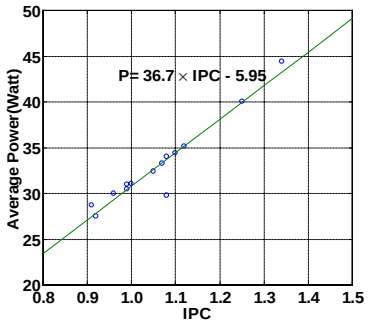
If instead of routine-based estimation, a flat average is used, the errors are high. We use this approach and estimate energy of OS execution on the *test* programs listed in Table 2. Not surprisingly, Figure 8 illustrates that there is 20% to 50% error if energy is estimated with a flat average OS power for all programs. Therefore, the paradigm of blindly treating the OS as monolithic software is unlikely to yield highly accurate estimation.

5. RUN-TIME OS POWER MODELING

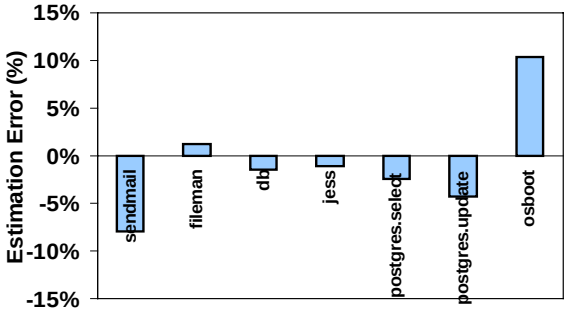
As discussed in Section 2, live power estimation is valuable for run-time power management and optimizations. The proposed routine level power estimation technique characterizes the power behavior of each OS routine at profiling stage and uses that information to compute the run-time power dissipation. The overhead of estimation is the computation needed for a first order linear processing of the IPC at OS routine boundaries, which is low.

The linear regression model parameters can be stored in a smaller look-up table and the operating system can dynamically compute power and energy at run-time. If the routine of interest is not found in the table, a single performance correlated average power number P_{OS} can be used. The maximum error that could occur by using such an approach is shown in Figure 9. Generally, the OS power correlates well with IPC and the cumulative power estimation error using the power model $P_{OS} = k_1 \times IPC_{os} + k_0$ is seen to yield errors less than 10%.

In some cases, cumulative (average) power estimation is insufficient and power has to be modeled and estimated on a fine-grained basis. Generating accurate and fine grained power estimation of an OS on a given system is important to computer architects as well as OS developers who need insight into machine's power efficiency to tune their code.

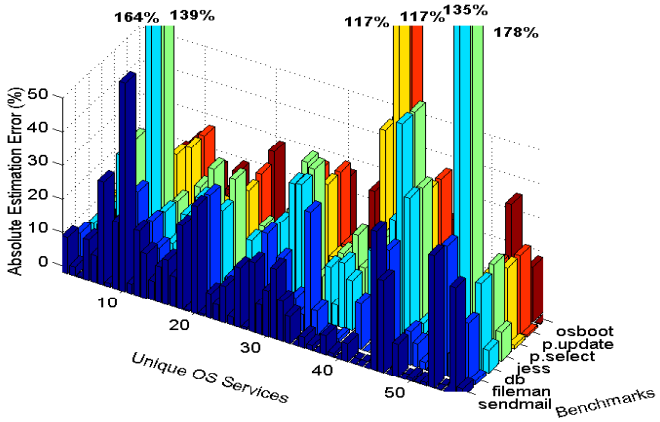


(a) Regression Model: $P_{OS} = k_1 \times IPC_{os} + k_0$

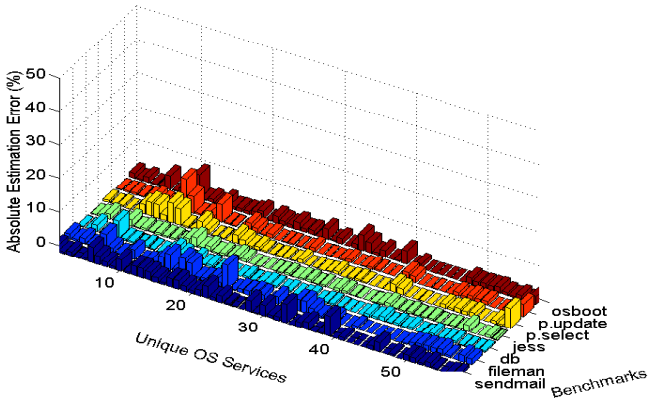


(b) Estimation Errors

Figure 9. OS Power Estimations Using Single Power/IPC Correlation Model



(a) Single Regression Model



(b) Routine based Regression Models

Names of OS Service Routines

1:utlb	2:pfault	3:vfault	4:COW_fault	5:demand_zero	6:timein	7:simscsi_intr	8:if_etintr	9:du_poll	10:clock
11:fchmod	12:exit	13:fork	14:read	15:write	16:open	17:close	18:unlink	19:time	20:brk
21:lseek	22:getpid	23:getuid	24:alarm	25:access	26:syssgi	27:dup	28:pipe	29:getgid	30:ioctl
31:utssys	32:execve	33:fcntl	34:ulimit	35:getdents	36:sigreturn	37:getsockname	38:getdomainname	39:setreuid	40:sproc
41:prctl	42:mmap	43:mprotect	44:msync	45:BSDsetpgrp	46:getrlimit	47:cacheflush	48:xstat	49:lxstat	50:fxstat
51:ksigaction	52:sigprocmask	53:sigsuspend	54:getcontext	55:setcontext	56:waitsys	57:setrlimit			

Figure 10. A Comparison of Run-time Per-routine based Estimation Error

To evaluate the run-time suitability of the proposed routine level power modeling approach, we performed a comparative study of the flat and routine level power modeling schemes in terms of per-module accuracy. As it can be seen, routine level modeling (Figure 10b) consistently produces results that are less than 6% away from the exact, cycle-accurate values, while the flat model (Figure 10a) scheme can generate up to 178% error in some cases. Modeling power behavior at OS service routine level drastically reduces the run-time estimation error, implying the good power tracking ability of this model. On the other hand, building single model for the whole operating system, although achieves acceptable cumulative power estimation accuracy, can lead to measurable estimation error when applied to track the fin-grained run-time power behavior. This fact implies that the "black box" power modeling approaches taken in [12, 3] are unlikely to be effective for run-time power tracking.

As described earlier, many hardware platforms have restrictions on the member of counters that can be configured simultaneously to count events. Therefore, a good power model should rely on minimal number of hardware event counters but must still maintain high accuracy. Table 3 lists energy accounting mechanisms [3] that rely on 2, 3, 5, and 7 types of counters respectively. For example, the 5-CS uses 5 hardware counters, namely, cycles, graduated instructions, L1 data cache accesses, L2 data cache accesses and main memory references to build regression power model and evaluate power.

Table 3. Hardware Counter Schemes

Events	Schemes			
	2-CS	*3-CS*	*5-CS*	*7-CS*
Cycles	+	+	+	+
Graduated Instructions	+	+	+	+
L1-D Cache Accesses		+	+	+
L1-I Cache Accesses				+
L2-D Cache Accesses			+	+
L2-I Cache Accesses				+
Main Memory References			+	+

Figure 11 compares the estimation accuracy of the proposed routine level OS power model that uses 2 counters (RL 2-CS) with flat modeling schemes that rely on more hardware counters. While the 3-CS, 5-CS and 7-CS outperform the 2-CS scheme in some cases in terms of accuracy, they show unpredictable behavior, depending on the benchmarks. The RL 2-CS scheme is the only one that offers consistent low error. One can see that the RL 2-CS model outperforms the flat regression models that use more hardware counters, indicating the benefit of combining hardware and software knowledge in energy modeling.

The proposed technique requires initial energy profiling of OS routines, which necessitate a full-system power-aware simulator such as SoftWatt [9]. However, the models described in this paper are independent of the actual method used to profiling. If sophisticated data acquisition based measurements are available, the measurement method can be used. The OS routine level power characterization is computation intensive. However, the power estimation does not require power simulation once that information is built, making it outperform other simulation-based approaches in terms of efficiency. The scheme also needs run-time measurement of cycles and IPC. All high-end microprocessors provide these counters and hence obtaining the information is not a problem, making it generally applicable to all hardware platforms. The run-time OS power estimation involves a first order linear operation on a single power metric, reducing estimation overhead.

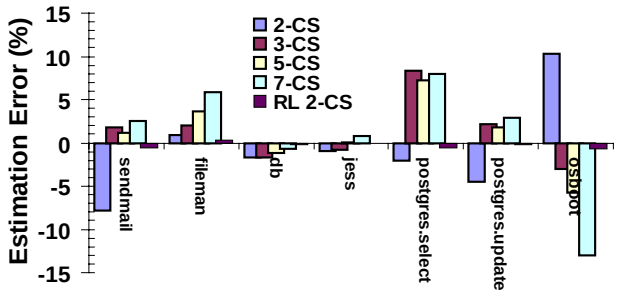


Figure 11. A Comparison of Different Hardware Counter Schemes

6. CONCLUSIONS

Modern computer systems are characterized by the presence of high performance, general-purpose processors and software (operating systems and user applications) running on it. Power modeling is increasingly becoming a critical issue during system designs, as well as run-time power/performance optimizations.

This paper proposes power models for the operating system (OS), a major power consumer in many modern application executions. The proposed models rely on a few metrics of interest for power evaluation. Profiling of several Java, Database, file/e-mail workloads illustrated a strong correlation between IPC and OS routine power. Exploiting this correlation, we built a model to estimate energy consumption of OS activity. Profiling done on one set of programs is used to estimate energy of another set of programs and yields a high accuracy within 1%. The proposed routine level power model not only offers superior accuracy when compared to a simpler, flat OS power model, but also provides per-routine estimation errors of less than 6% when applied to track the run-time OS energy profile.

The integrated OS performance/power characterization not only leads to efficient power estimation for OS-intensive applications but also provides hint to reduce OS power consumption. Having known the routine based power dissipation behavior, hardware can be adapted for power minimization. For example, to save power, the size of a banked instruction window or reorder buffer can be dynamically reconfigured when OS routines with low IPC are detected. In another scenario, dynamic voltage scaling or frequency throttling can be applied to the OS code that performs intensive I/O when the processor ILP dose not really matter. We plan to investigate OS power saving techniques in the future.

7. ACKNOWLEDEGMENT

This research is partially supported by the National Science Foundation under grant number 0113105, and by AMD, Intel, IBM, Tivoli and Microsoft Corporations. We would like to thank Sudhanva Gurumurthi, Anand Sivasubramaniam and Vijaykrishnan Narayanan from Pennsylvania State University for their help in the development of the simulation infrastructure that was used in this paper.

8. REFERENCES

[1] A. R. Alameldeen and D. A. Wood, Variability in Architectural Simulations of Multi-threaded Workloads, In Proceedings of the International Symposium on High Performance Computer Architecture, 2003.

[2] K Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Lohout, C. Smit, T. B. Zhang and B. Jacob, The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems, In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2001.

[3] F. Bellosa, The Benefits of Event-driven Energy Accounting in Power-sensitive Systems, In Proceedings of 9th ACM SIGOPS European Workshop, 2000.

[4] R. Berrendorf and B. Mohr, PCL - The Performance Counter Library Version 2.2, http://www.fz-juelich.de/zam/PCL/, Jan. 2003.

[5] D. Brooks, V. Tiwari and M. Martonosi, Wattch: A Framework for Architectural-level Power Analysis and Optimizations, In Proceedings of the International Symposium on Computer Architecture, 2000.

[6] D. Brooks and M. Martonosi, Dynamic Thermal Management for High-Performance Microprocessors, In Proceedings of the International Symposium on High-Performance Computer Architecture, 2001.

[7] J. W. Chen, M. Dubois and P. Stenström, Integrating Complete-System and User-level Performance/Power Simulators: The SimWattch Approach, In Proceedings of International Symposium on Performance Analysis of Systems and Software, 2003.

[8] R. P. Dick, G. Lakshminarayana, A. Raghunathan and N. K. Jha, Power Analysis of Embedded Operating Systems, In Proceedings of the Design Automation Conference, June 2000.

[9] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li and L. K. John, Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach, In Proceedings of the International Symposium on High Performance Computer Architecture, 2002.

[10] M. Huang, J. Renau, S. M. Yoo and J. Torrellas, A Framework for Dynamic Energy Efficiency and Temperature Management, In Proceedings of the International Symposium on Microarchitecture, 2000.

[11] Intel Pentium 4 Processors - Manuals, Intel Corporation, 2002.

[12] R. Joseph and M. Martonosi, Run-Time Power Estimation in High Performance Microprocessors, In Proceeding of the International Symposium on Low Power Electronic Device, 2001.

[13] A. R. Lebeck, X. B. Fan, H. Zeng and C. S. Ellis, Power Aware Page Allocation, In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[14] T. Li and L. K. John, Understanding Control Flow Transfer and its Predictability in Java Processing, In Proceedings of International Symposium on Performance Analysis of Systems and Software, 2001.

[15] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan and J. Rubio, Understanding and Improving Operating System Effects in Control Flow Prediction, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[16] S. Manne, A. Klauser and D. Grunwald, Pipeline Gating: Speculation Control for Energy Reduction, In Proceedings of the International Symposium on Computer Architecture, 1998.

[17] D. Ofelt and J. L. Hennessy, Efficient Performance Prediction for Modern Microprocessors, In Proceedings of the International Conference on Measurement and Modeling of Computer Systems, 2000.

[18] J. Ousterhout, Why aren't Operating Systems Getting Faster as Fast as Hardware?, In Proceedings of the Summer USENIX Conference, 1990.

[19] S. Palacharla, N. P. Jouppi and J. E. Smith, Quantifying the Complexity of Superscalar Processors, CS-TR-1996-1328, University of Wisconsin, Nov. 1996.

[20] "PostgreSQL", http://www.us.postgresql.org/

[21] G. Qu, N. Kawabe, K. Usami and M. Potkonjak, Function-Level Power Estimation Methodology for Microprocessors, In Proceedings of the Design Automation Conference, 2000.

[22] M. Rosenblum, S. A. Herrod, E. Witchel and A. Gupta, Complete Computer System Simulation: the SimOS Approach, IEEE Parallel and Distributed Technology: Systems and Applications, vol.3, no.4, Winter 1995.

[23] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, Automatically Characterizing Large Scale Program Behavior, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[24] A. Sinha, A. Wang and A. P. Chandrakasan, Algorithmic Transforms for Efficient Energy Scalable Computation, In Proceedings of the International Symposium on Low Power Electronics and Design, 2000.

[25] SPEC JVM98 Benchmarks, http://www.spec.org/jvm98/.

[26] T. K. Tan , A. Raghunathan, G. Lakshminarayana and N. K. Jha, High-level Software Energy Macro-modeling, In Proceedings of the Design Automation Conference, 2001.

[27] T. K. Tan, A. Raghunathan and N. Jha, Embedded Operating System Energy Analysis and Macro-modeling, In Proceedings of the International Conference on Computer Design, 2002.

[28] T. K. Tan, A. Raghunathan and N. Jha, EMSIM: An Energy Simulation Framework for an Embedded Operating System, In the Proceedings of the International Conference on Circuits and Systems, 2002.

[29] V. Tiwari, S. Malik, A. Wolfe and M. T. C. Lee, Instruction Level Power Analysis and Optimization of Software, Journal of VLSI Signal Processing, 1-18, 1996.

[30] Transaction Processing Council, The TPC-C Benchmark, http://www.tpc.org/tpcc/.

[31] M. Valluri and L. K. John, Is Compiling for Performance == Compiling for Power?, In Proceedings of the 5th Annual Workshop on Interaction between Compilers and Computer Architectures, 2001.

[32] C. Xia and J. Torrellas, Comprehensive Hardware and Software Support for Operating Systems to Exploit MP Memory Hierarchies, IEEE Transactions on Computers, May 1999.

[33] W. Ye, N. Vijaykrishnan, M. Kandermir and M. J. Irwin, The Design and Use of SimplePower: A Cycle-accurate Energy Estimation Tool, In Proceedings of Design Automation Conference, 2000.

[34] H. Zeng, X. B. Fan, C. Ellis, A. Lebeck and A. Vahdat, ECOSystem: Managing Energy as a First Class Operating System Resource, In the Proceedings of the International Symposium on Architecture Support for Program Language and Operating System, 2002.

Appendix 1. Power Characterization of OS Routines (ε: Regression Model Fitting Error)

Interrupts

OS Services	Avg. Cycles	IPC		Power		Regression Model $P = k_1 \times IPC + k_0$			Comment
		Avg.	Std. Dev. (%)	Avg. (W)	Std. Dev. (%)	k_1	k_0	ε	
utlb	13	0.92	0	28	0.1	23.6	6.2	0.17%	TLB miss handler
pfault	1,100	1.16	19	40	6.2	32.8	1.9	0.48%	protection fault
vfault	971	1.43	11	47	3.4	23.9	12.9	4.89%	virtual memory fault
COW_fault	2,574	1.65	8	54	2.6	32.1	1.1	0.19%	copy-on-write fault
demand_zero	1,939	1.54	16	44	4.5	27.6	1.5	0.40%	zero fill page faults
simscsi_intr	993	0.98	37	35	12.6	33.9	1.3	1.94%	SCSI disk I/O interrupt
if_etintr	241	1.38	51	42	15.0	29.4	1.1	1.57%	Ethernet interrupt
du_poll	481	0.95	26	35	9.3	35.7	0.8	5.04%	input/output multiplexing
clock	2,457	0.53	26	20	9.5	36.4	0.6	2.68%	clock interrupts

Process and Interprocess Control

OS Services	Avg. Cycles	IPC		Power		Regression Model $P = k_1 \times IPC + k_0$			Comment
		Avg.	Std. Dev. (%)	Avg. (W)	Std. Dev. (%)	k_1	k_0	ε	
exit	63,492	1.08	12	39	4.3	36.0	0.6	0.42%	terminate a process
fork	16,154	1.28	6	45	2.3	36.5	-1.7	0.99%	create a new process
getpid	226	1.51	23	48	7.7	33.6	-2.7	0.75%	return the process ID of the calling process
getuid	248	1.34	5	42	1.8	33.6	-3.1	0.17%	return the real user ID of the calling process
alarm	594	0.77	9	26	2.9	32.8	0.6	0.14%	set a process alarm clock
pipe	4,188	0.71	11	25	3.8	35.4	0.4	0.50%	create an interprocess channel
getgid	240	1.41	21	43	6.5	30.5	0.4	0.10%	return the real group ID of the calling process
execve	64,401	1.23	4	43	1.2	31.0	4.6	0.20%	execute a file
sigreturn	924	1.17	7	39	2.4	34.5	-1.4	0.56%	returns from a signal handler
getsockname	1,137	0.74	10	25	3.1	32.4	1.2	0.57%	get socket name
getdomainname	590	0.70	18	22	5.6	31.2	0.3	0.04%	get name of current NIS domain
setreuid	1,455	0.43	6	14	2.2	34.7	-0.9	0.08%	set real and effective user ID's
sproc	51,775	1.24	4	41	0.1	15.7	21.1	0.12%	create a new share group process
prctl	813	0.48	12	15	3.8	31.8	-0.2	0.89%	operations on a process
ksigaction	624	1.17	7	38	2.3	32.8	0.1	0.70%	used to implement all type signal routines
sigprocmask	364	1.46	29	47	9.2	31.4	0.9	0.03%	alter and return previous state of the blocked signals
BSDsetpgrp	2,565	0.41	4	15	1.6	35.4	0.3	0.55%	set process group ID
sigsuspend	9,901	0.30	15	11	5.0	33.7	0.7	0.94%	release blocked signals and wait for interrupt
getcontext	679	1.38	31	43	9.6	30.6	0.2	0.19%	get current user context
setcontext	1,025	0.97	14	32	4.5	33.1	0.1	0.48%	set current user context

File System

OS Services	Avg. Cycles	IPC		Power		Regression Model $P = k_1 \times IPC + k_0$			Comment
		Avg.	Std. Dev. (%)	Avg. (W)	Std. Dev. (%)	k_1	k_0	ε	
read	2,614	1.36	19	45	6.1	29.6	4.7	4.53%	read file
write	9,344	0.91	9	33	3.2	34.3	1.5	1.27%	write file
open	8,626	0.97	10	35	3.5	34.3	1.2	0.41%	opens a file, serial port or command pipeline
close	2,131	0.77	21	27	6.5	30.4	3.9	2.61%	close an open channel
unlink	8,904	1.00	7	36	2.0	30.0	5.5	0.11%	remove a link to a file
lseek	536	1.01	22	33	7.3	33.1	-0.5	2.49%	move read/write file pointer
access	6,547	1.11	18	39	5.9	33.3	1.7	0.57%	determine accessibility of a file
dup	1,074	0.74	18	25	5.7	32.4	1.2	0.56%	duplicate an open file descriptor
ioctl	5,230	0.51	3	18	1.0	32.5	1.1	0.52%	perform a variety of control functions on devices
fcntl	613	1.39	25	45	8.3	33.2	-0.9	0.95%	file and descriptor control
getdents	5391	1.00	35	34	11.3	32.4	1.8	0.58%	read directory entries and put in a file system independent format
xstat	5,990	1.22	14	43	4.8	35.0	0	0.85%	obtain file attributes
lxstat	3,517	1.52	3	53	1.0	34.9	-0.2	0.20%	obtain symbolic link file attributes
fxstat	1,293	0.85	18	28	5.4	30.5	2.0	2.01%	obtain information about an open file known by the file descriptor

Miscellaneous Services

OS Services	Avg. Cycles	IPC		Power		Regression Model $P = k_1 \times IPC + k_0$			Comment
		Avg.	Std. Dev. (%)	Avg. (W)	Std. Dev. (%)	k_1	k_0	ε	
brk	2,974	0.80	18	30	6.3	35.8	1.1	1.03%	change data segment space allocation
syssgi	2,377	1.06	3	37	1.0	34.4	0.3	0.29%	system interface specific to SGI
utssys	1,833	0.47	2	16	0.5	31.9	0.7	0.22%	set/get system's hostname
ulimit	364	1.08	52	34	15.9	30.4	1.0	0.02%	get and set user limits
mmap	7,311	0.74	12	26	4.2	34.5	0.6	1.08%	map pages of memory
mprotect	1,703	0.99	3	35	1.1	35.3	0.3	0.50%	set protection of memory mapping
msync	23,107	0.61	3	23	0.1	36.8	0	0.36%	synchronize memory with physical storage
getrlimit	1,045	0.42	2	14	0.2	18.0	6.1	0.42%	control maximum system resource consumption
cacheflush	867	1.22	2	41	0.8	33.4	0.1	0.41%	flush contents of instruction and/or data cache
waitsys	3,338	0.63	65	22	1.9	32.7	1.4	0.55%	underlying system call for all wait-like calls
timein	1,185	0.65	15	23	5.0	34.3	0.4	2.89%	set timer
time	478	0.97	7	32	2.3	33.2	-0.6	0.85%	count elapsed time