Available online at www.sciencedirect.com

SCIENCE DIRECT®

Microprocessors and Microsystems 30 (2006) 189–198

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

Operating system power minimization through run-time processor resource adaptation

Tao Li [a,*], Lizy Kurian John [b]

[a] Department of Electrical and Computer Engineering, University of Florida, 223 Larsen Hall, P.O. Box 116200, Gainesville, FL 32611, USA
[b] Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712, USA

Available online 6 January 2006

Abstract

The increasingly constrained power budget of today's microprocessor has resulted in a situation where power savings of all components in a system have to be taken into consideration. The Operating System (OS), which manages both software and hardware resources, dissipates a significant portion of power in the execution of many modern applications. This paper profiles run-time OS power/performance characteristics and advocates a routine based OS-aware microprocessor resource adaptation mechanism to save run-time OS power. This approach permits precise hardware reconfigurations for the OS with low overhead and allows fine-grained performance/power tuning at the microarchitectural level. Simulation results show that compared to existing sampling-based adaptation schemes, this novel methodology yields a more attractive power and performance trade-off on OS execution. To our knowledge, this work is the first to address the power saving issue of the OS itself, an increasingly important area that has been largely overlooked in previous studies.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Adaptive computing; Design; High-performance; Low-power-design; Performance trade-offs; Reconfigurable-systems; System-level

1. Introduction

Today's high-performance microprocessor constitutes millions of transistors clocked at Gigahertz frequency, which results in significant power dissipation [24]. Its performance-driven market and increasingly constrained power budget necessitates an effort to save power in all of its components, from circuits to the software it runs [3–5,11–14,23,27,29,36]. For example, dynamic thermal management [20,35] may have to be used to slow down the chip's execution when its temperature exceeds a given threshold or in a battery-driven computing environment, an application may have to be executed with a degraded QoS setting when the available battery energy is low [2,29].

The increasing concern on the power issue drives the need for the above performance/energy trade-offs for all components of a system [14]. The Operating System (OS) which manages both hardware and software resources, constitutes a major software component of today's complex systems implemented with high-end and general-purpose microprocessors, memory hierarchy and heterogeneous I/O devices. Many modern and emerging workloads (e.g. database, web servers and file/e-mail applications) exercise the OS significantly [1,6,21]. OS execution not only occupies a large fraction of machine cycles, but also accounts for a significant amount of power dissipation. Using an energy-aware full system simulation framework, we profiled OS power characteristics on various system workloads. We observed that the OS draws 32% of the total energy (CPU, cache and main memory) on the studied workloads. Note too, that the proportion of OS power consumption is projected to increase due to increasing demands for system management activities, such as thermal sensor reading, energy accounting and power control for memory and I/O devices [2,26,35,36]. Clearly, in a power constrained environment, OS power saving needs to be addressed. However, previous studies [3–5,11,20,26,27,29,35] focused entirely on reducing power for user-only applications. To our knowledge, power saving and optimization for the OS itself has received little attention.

In this paper, we explore the adaptation of processor resources to reduce OS power consumption on today's high performance superscalar processors. These processors exploit aggressive hardware design to maximize performance across a wide range of targeted applications. It has been observed that a program's computational requirement, generally measured by the instruction per cycle (IPC), varies during its execution [25].

* Corresponding author. Tel.: +1 352 392 9510; fax: +1 352 846 1416.
E-mail addresses: taoli@ece.ufl.edu (T. Li), ljohn@ece.utexas.edu (L.K. John).

0141-9331/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2005.12.001

By appropriately tuning processor resources to the actual needs of the program, significant power savings can be achieved with minimal impact on performance. To reduce power, hardware can be dynamically adapted to provide appropriate resources to the program's computational demand. The OS IPC does not scale well with the increasing superscalar capability, making it an ideal candidate for resource adaptation. Given the assumption that OS execution can be timely and accurately detected, significant power savings can be achieved (with tolerable performance penalty) by tailoring appropriate processor computational resources to match the OS requirements.

Current adaptation techniques [3–5,11] rely on periodic sampling to match the program computational requirements with processor resources. However, we show in this paper that resource adaptation based on a sampling window becomes less efficient when applied to the exception-driven and short-lived OS execution [13]. Moreover, for large and sophisticated programs like the OS, a naïve sampling scheme does not guarantee the optimal solution when both energy and performance are under consideration. Therefore, we advocate a routine based OS-aware microprocessor resource adaptation scheme. The proposed innovative technique ensures that processor resources match to the computational demands of the OS in a timely and optimal fashion yet with low overhead. Compared with existing techniques, the proposed scheme has the following advantages: (1) OS-aware resource adaptation guarantees the timely and fine-grained resolution required to capture the exception-driven, short-lived OS activity. (2) Adapting processor resources only at OS routine boundaries largely eliminates reconfiguration latency. (3) Routine based adaptation selects the optimal configuration for each individual routine, yielding a more attractive power and performance trade-off. (4) Aggressive optimizations can be safely applied to certain OS routines to further save energy without degrading performance.

This work makes the following contributions: (1) employing a full-system power simulation framework, we profile the power behavior of a commercial operating system across a wide range of applications. (2) We analyze performance/power trade-offs at the OS service routine level to identify power saving opportunities in the OS. (3) We propose a novel mechanism that allows the microprocessor to adapt its resources to OS execution for power saving without incurring significant overhead.

The rest of this paper is organized as follows: Section 2 describes our experimental methodology, including SoftWatt simulator, machine configuration and studied workloads. Section 3 presents OS power characterization and performance/power trade-off analysis. Section 4 introduces the sampling-adaptation scheme and demonstrates the challenges in sampling OS activity. Section 5 proposes the routine based OS-aware microarchitecture adaptation scheme and discusses its benefits. Section 6 presents simulation results. Section 7 discusses related work. In Section 8, we conclude with some final remarks.

2. Experimental methodology

We use the complete system power simulator SoftWatt [7] that models the power dissipation of the CPU, memory hierarchy and a low-power disk subsystem to investigate the power behavior of the OS. The SoftWatt tool, built on top of the SimOS infrastructure [8], uses validated energy models similar to other low-level power simulators like Wattch and SimplePower [9,28]. 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 an eight-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 μm 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 features.

We use 13 applications that have different characteristics. *Pmake* [39] is a parallel program development workload. *Vortex* and *gcc* are two benchmarks from the SPECint95. The *sendmail* benchmark [22] forwards emails using the Simple Mail Transport Protocol (SMTP). *Db*, *jess*, *javac* and *jack* are Java programs from the SPECjvm98 [19] suite executed on a SGI-ported Sun Java Virtual Machine (JVM). We also use two benchmarks that run on a relational database management system (DBMS) engine-PostgreSQL [10]. The database is populated with relational tables for the TPC-C [31] benchmark. *Postgres.select* performs a sequential table scan of a table with 1 million rows and a selectivity of 3%. *Postgres.update* updates a field of a 300,000 row table. *Fileman* performs popular file management activities, such as copy, remove, tar -cvf and tar -xvf operations. *Osboot* executes a complete OS booting

Table 1
Baseline machine

Processor core	
Technology/V_{dd}/frequency	0.18 μm/2.0 V/900 MHz
Fetch/issue/retire width	8
Physical register file	64
Instruction window size	128
Reorder buffer size	256
Function units	MIPS R10000 Like
Branch target buffer (BTB)	2048-entry, four-way
Return address stack	32-entry w/misprediction repair
Branch prediction/penalty	24K-entry hybrid/10 cycles
Load store queue size	64
Memory hierarchy	
MMU	Fully associative TLB, 48-entries, 4 kb page size
L1 I-cache	32 kb, two-way, 64 b blocks, 1 cycle
L1 D-cache	32 kb, two-way, 32 b blocks, 1 cycle
L2 Cache	512 kb, two-way, 128 b blocks, 9 cycle
Memory	256 Mb, 180 cycle access

Table 2
Benchmarks and execution statistics

Benchmarks	Total instructions (M)	Description
pmake	1117	Two parallel compilation processes compile the modified Andrew benchmark
gcc	1036	Compiles pre-processed source into optimized SPARC assembly code
vortex	1811	A full object oriented database
sendmail	1494	UNIX electronic mail transport agent
fileman	177	File management application
db	201	Performs multiple database functions on a memory resident database
jess	467	Java expert shell system based on NASA's CLIPS expert system
javac	366	The JDK 1.0.2 Java compiler compiling 225,000 lines of code
jack	1782	Parser generator with lexical analysis
postgres.select	1516	Object -relational DBMS PostgreSQL executes a select query
postgres.update	1438	Object-relational DBMS PostgreSQL executes an update query
postgres.join	1849	Object-relational DBMS PostgreSQL executes a join query
osboot	48	A complete OS boot sequence

sequence from a root disk image and then generates a shell for the user. Table 2 summarizes the benchmarks and statistics.

3. OS power characterizatcion and performance/power trade-offs

Employing the SoftWatt framework, we profiled the power behavior of the OS running on the above-described applications. We analyzed the performance/power trade-offs at the OS service routine level to identify power saving opportunities in the OS.

Fig. 1 shows the percentage of total energy (microprocessor and memory subsystems) dissipated by the OS on the simulated experimented applications. As can be seen, on the average, the processor spends 41% of its time on the OS and the OS draws 32% of the total energy, making it a major power consumer. This suggests implies that overlooking the OS effect can cause significant software energy estimation error.

Modern operating systems are large, sophisticated software and their complexities are hidden behind a relatively simple interface—the OS kernel service routines, which provide a commonly used interface for all applications to exercise the OS. Just as instructions are the fundamental units of software execution, the OS service routines can be thought of as the fundamental unit of OS execution. 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. Fig. 2 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 investigate a total number of 186 OS service routines. Fig. 2 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 frequently invoked service routines while on benchmarks *sendmail*, *postgres.update* and *osboot*, the OS energy consumption is contributed by a wide range of service routines.

Today's high-performance microprocessor designs attempt to push the performance envelope by employing aggressive out-of-order execution mechanisms [30]. As a result, in a complex, high performance superscalar processor, circuits used to exploit ILP consume a dominant portion of the power [4,32]. We found that data-path and pipeline structures, which support multiple issue and out-of-order execution, consume 50% of the total power on the examined OS routines. The inherent instruction level parallelism (ILP) in the OS has been found to be much lower than in user applications [1,6,21,38]. The nature of OS code limits the available instruction level parallelism. For example, the OS usually uses serializing instructions to synchronize I/O operations. A serializing instruction requires that all other instructions in the pipeline complete before it executes. Moreover, much architecture treat privilege instructions, such as move to/from special register, TLB management, explicit cache operations, and interrupt/exception return, as serialization instructions. To handle precise exceptions, the processor pipeline must drain before OS code execution can begin. Serializing instructions, interrupts and privilege level changes, may spend considerable cycles in execution, forcing the decoder to wait and thus increase the resource stalls, limiting the available ILP. Fig. 3 compares the IPC of the user and the OS running the twelve studied benchmarks on an eight-issue machine. The OS IPC is 1.2×–2.4× lower than the user IPC, suggesting that the OS does not exploit the superscalar capabilities provided by the wide-issue, aggressive processor as efficiently as user code does.

The energy consumed in the data-path during execution usually depends on the number of instructions that flow

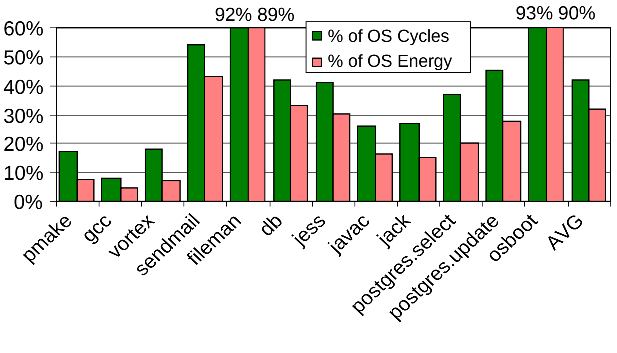

Fig. 1. Percentage of energy dissipated by the OS.

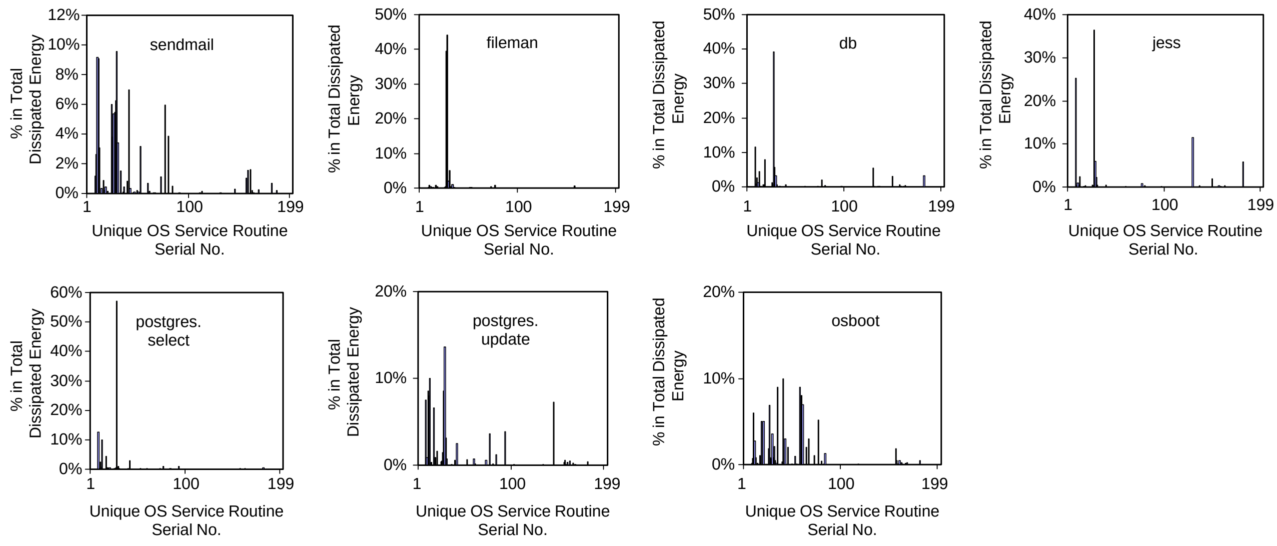

Fig. 2. Routine level energy distributions in OS.

through it. 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.

To understand performance/power trade-offs on the OS, we measure OS power and IPC on machines with different configurations. As described in Table 1, the baseline machine we consider is an aggressive, eight-issue superscalar processor. To reduce its power consumption, the processor can be reconfigured to six-issue, four-issue, two-issue and one-issue modes by reducing its computational capacity. Previous studies [3–5] observe that power consumption of a high-performance superscalar machine is largely determined by the instruction issue width and the scale of major microarchitectural structures, such as: instruction window (IW), reorder buffer (ROB) and load store queue (LSQ). Therefore, in six-issue mode, we limit the instruction fetch, decode, issue and retire width to be six and disable 1/4 of the IW, ROB and LSQ entries. In four-issue, two-issue and one-issue modes, we restrict the issue width to be 4, 2, and 1 and disable 1/2, 3/4, and 7/8 of the above resources (i.e. IW, ROB and LSQ) respectively. Table 3 shows the OS IPC and power consumption (average over all benchmarks) on eight-issue, six-issue, four-issue, two-issue, and one-issue machines respectively. It can be seen that by reducing processor resources, the four-issue machine saves 49% of power with a performance loss of only 5%.

The OS IPC does not scale well with the increasing superscalar capability, making it an ideal candidate for resource adaptation. Given the assumption that OS execution can be timely and accurately detected, significant power savings can be achieved (with tolerable performance penalty) by tailoring appropriate processor computational resources to match the OS requirements.

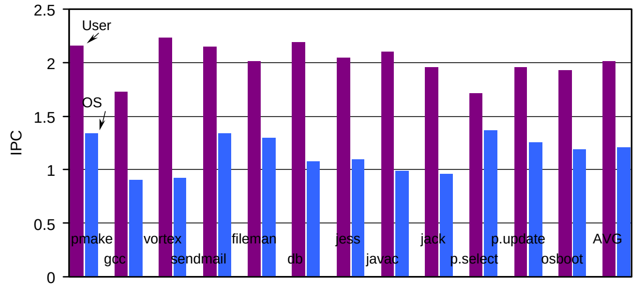

Fig. 3. IPC of user and OS on an eight-issue machine.

4. Sampling based adaptation: challenges for OS

In prior research, the run-time periodic sampling of measurable metrics (e.g. IPC) has ubiquitously been used to estimate program computational demand and to guide adaptations. In sampling based techniques, program execution cycles are partitioned into fixed period intervals as in Fig. 4. The duration of each interval is called a sampling window. A performance metric, such as IPC, is measured within a sampling window to estimate program computation demand for the next execution interval window. At the boundaries of each sampling window, adaptation decisions are made.

Current sampling-adaptation approaches [3,11] use a finite state machine (FSM) to specify the transitions between different configurations. For example, Fig. 5 shows a FSM for transitioning between the normal mode (eight-issue) and the low power modes (six-issue, four-issue, two-issue

Table 3
OS IPC and power on machines with different superscalar capability

	One-issue	Two-issue	Four-issue	Six-issue	Eight-issue
IPC	0.88	1.09	1.15	1.19	1.21
Power (W)	6.4	12.2	21.7	31.1	42.8

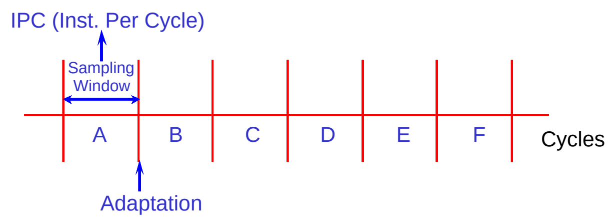

Fig. 4. Sampling window.

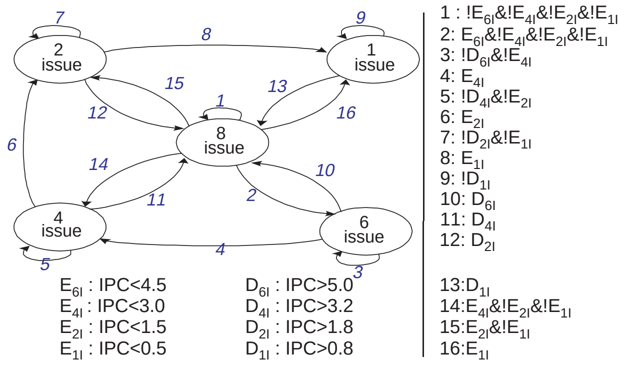

Fig. 5. FSM used in sampling based adaptation (trigger conditions and thresholds are set and extended according to [3]).

and one-issue) described in Section 3. The enabling ($E_{x\mathrm{I}}$) and disabling conditions ($D_{x\mathrm{I}}$) and the IPC thresholds are set and extended according to the one proposed by Bahar et al. [3]. For example, the enabling conditions for entering the four-issue mode are $E_{4\mathrm{I}}$ or $!D_{4\mathrm{I}}\&!E_{2\mathrm{I}}$ or $E_{4\mathrm{I}}\&!E_{2\mathrm{I}}\&!E_{1\mathrm{I}}$, respectively. In this paper, we consider this adaptation technique as the baseline scheme.

At run-time, the estimated program IPC within the previous sampling window serves as the input of the FSM to choose the configurations for the current interval, as shown in Fig. 4. The basic premise of this sampling algorithm is that past program behavior indicates its future needs. The sampling window period (T_s) determines the finest granularity at which program phase changes can be resolved. Generally, T_s has to be small enough to capture the changes in program behavior.

In practice, accomplishing an adaptation can cause a performance penalty (latency marked as T_a in Fig. 6). In the superscalar processor design, IW, LSQ and ROB are implemented with a partitioned structure [5]. A reconfiguration has to guarantee that there are no instructions left on the partitions that will be deactivated. Additional care must be taken in resizing the ROB and LSQ because of their circular FIFO like structure [4]. Due to these restrictions, whenever an adaptation decision is made, the dispatch unit stops pumping instructions into the IW, LSQ and ROB until all existing instructions are drained out from the partitions to be turned off. This pipeline flushing like action can take a significant amount of time, depending on the number of instructions already in the pipeline and the number of cycles needed for them to complete [11]. Moreover, compared with single mode only execution, adaptations introduce extra latency due to pipeline warm-ups after the reconfigurations. As shown in Fig. 6, reducing the sampling window period ($T_h \ll T_s$) provides the capability of capturing fine-grained phase changes in execution. However, the aggregated adaptation overhead can be prohibitive. This fact prevents the use of a small sampling window without significantly slowing down the program execution. In [4], a sampling window of 2048 cycles is set. In [11], an even larger resizing period is chosen for the entire program hotspot, which could take several million cycles.

At run-time, user and OS execution appear alternately within the sampling windows, as shown in Fig. 6. The OS is activated voluntarily by a system call from the application, or from a call by some other application, or implicitly by some underlying periodic/asynchronous (timer/device interrupt) mechanism. The IPC discrepancy between the user and the OS indicates the different computational requirements when the user/OS context switches. When the program phase shifts (e.g. due to user/OS interactions), the prior interval becomes a poor estimate for the next.

In traditional and performance-centric OS design, highly optimized lightweight routines (e.g. faults and interrupt handlers) are usually implemented in order to keep the number of cycles down. Fig. 7 characterizes the average duration in cycles of individual OS services (note that the y-axis uses logarithmic scale). One can see that many OS service routines show a short-lived execution period. Theoretically, given a sampling interval of T_s, in order to accurately capture the phase shift caused by an OS service and exploit the adapted configuration for at least another sampling interval, the duration of that OS service T_{osd} should be at least $2T_s$ cycles, i.e. $T_{osd} \geq 2T_s$.

Fig. 7 shows that there are only 16 OS routines that satisfy the above condition on the duration ($\geq$4096 cycles) required by the 2048 cycle sampling interval, a window granularity commonly used to avoid the costly reconfiguration overhead. Fig. 8 further illustrates how OS service routines with different duration contribute to the total OS energy dissipation (note that

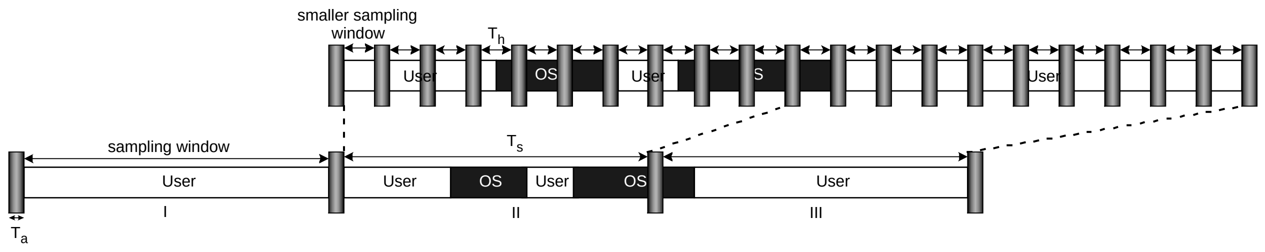

Fig. 6. Implications of sampling window sizes.

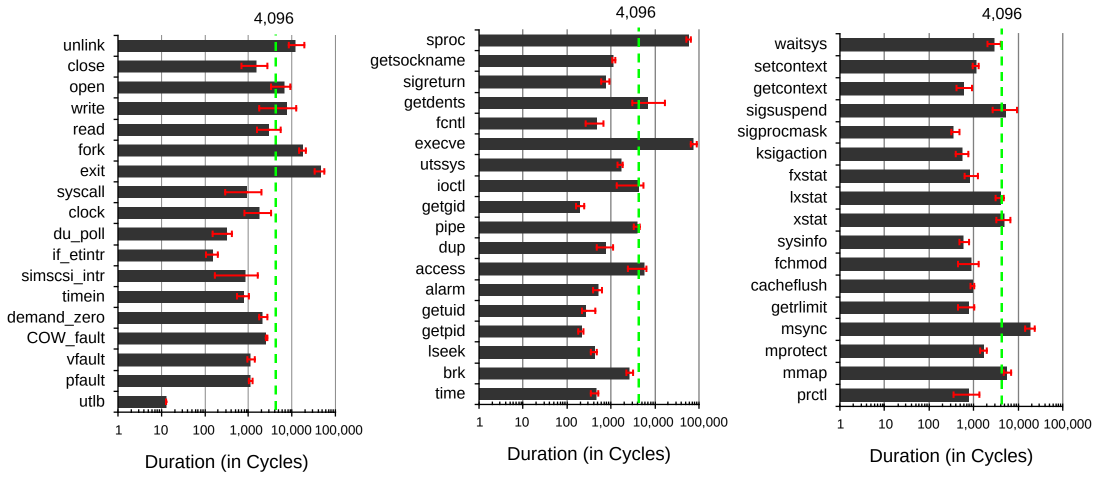

Fig. 7. Average duration of OS services (*y*-error bars show the maximum and minimum cycles).

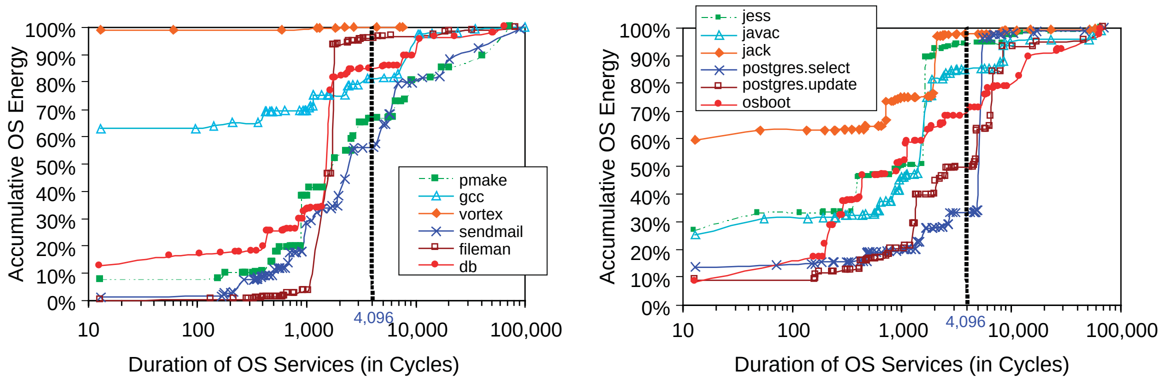

Fig. 8. Accumulative OS energy vs. OS service duration.

the *x*-axis uses logarithmic scale). It is observed that even though some OS services are very efficiently implemented from the execution cycle viewpoint, these lightweight OS services can have significant impact on the total OS energy. For example, on benchmark *postgres.update*, the OS service routines with duration less than 4096 cycles draw 50% of the OS energy. As described earlier, there is no guarantee that a sampling window, which is larger than 2048 cycles, will resolve these OS activities or will adapt processor resources in a timely fashion in order to reduce that portion of OS energy (shown in the left side of the dotted line in Fig. 8).

To summarize, a long window interval does not provide the opportunity to switch modes when the program phases change due to the exception-driven, non-deterministic and short-lived nature of user/OS interactions. On the other hand, the fine-grained switching required by the brief OS invocations makes it difficult to amortize the performance degradation due to frequent adaptations. To reconfigure the processor resources for the short-lived OS activity without raising costly adaptation overhead, we propose a routine based OS-aware processor adaptation mechanism targeting run-time OS power savings, as described in Section 5.

5. Proposed solution: OS-aware routine based adaptation

Routine based OS-aware adaptation is dedicated to reconfiguring the processor upon OS execution. The current machine execution mode is stored in the Processor Status Register (PSR); therefore, separating out OS execution can easily be done at run-time by looking at the PSR. Processor adaptations occur only at the boundaries of the user/OS context switches, as shown in Fig. 9. Today, almost all high-performance, out-of-order machines support precise exception to ensure the correctness of program execution. The OS invocations, either explicitly (e.g. system calls and I/O interrupts) or implicitly (e.g. fault handling) are treated as exceptions on these processors. Upon receiving an exception, the processor completes all previous instructions (specified in program order) and then flushes the pipeline [18]. At this point, a reconfiguration can be made with zero latency because there is no instruction left in the pipeline and the partitioned hardware structures. Similarly, when the processor returns from an OS service, another adaptation happens immediately by restoring the processor to the mode prior to the user/OS context switch. The processor then fetches the instructions

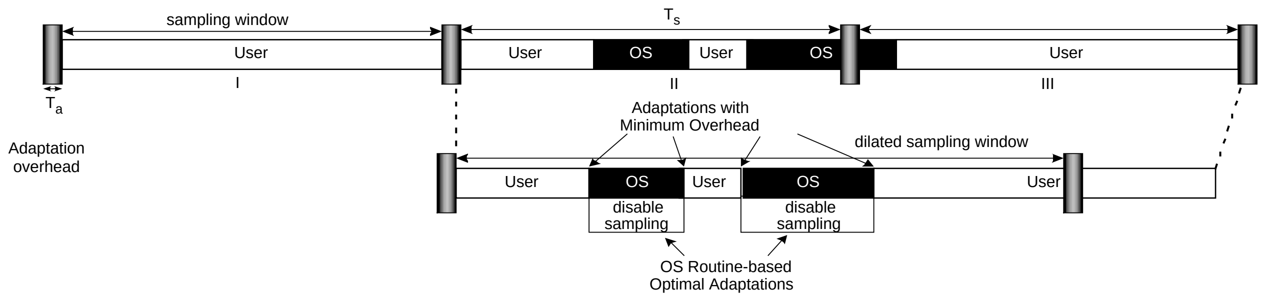

Fig. 9. Routine based OS-aware adaptation.

from the user applications and continuously executes using that mode.

Therefore, routine based OS-aware adaptation is capable of capturing all OS activity timely and accurately, while retaining a zero adaptation overhead in the OS. Separating OS activity out of the regular sampling interval creates a 'dilated' sampling window (as shown in Fig. 9), diminishing the number of reconfigurations and the total execution cycles of the user program. Moreover, this technique prevents pathological IPC degradations arising from erroneously matching processor configurations tailored for the OS to the user program (as shown in Fig. 9, windows II and III). This is critical since the user program, with the context switched from the OS, generally requires the full issuing capabilities of the machine to operate on new data and on the working set.

As described earlier, processor resource adaptation saves power but is detrimental to performance. The goal of such adaptation is to reduce power with minimum performance lost. The Energy-Delay product (EDP) is a reasonable metric to evaluate energy efficiency, namely, the goal of achieving high performance while minimizing energy consumption. However, due to the different characteristics of programs, a solution that is good for one program may not be an optimal solution for another program. For example, as illustrated in Fig. 10, given the power budget ($Power_{th}$), Energy×Delay trade-off-1 (T1) works better than Energy×Delay trade-off-2 (T2) does on program 1 ($Perf_{11} > Perf_{12}$). However, the observation does not hold on program 2 ($Perf_{21} < Perf_{22}$).

An individual OS routine performs a specific functionality and can exhibit vast variation in computational requirements. A configuration that is good for one routine may not be optimal for another. For example, Fig. 11 shows the Energy×Delay (normalized with eight-issue mode) of different OS service routines (*clock*, *COW_fault* and *read*) running on different modes. The routine *Clock* processes timer interrupts. *COW_ fault* performs page level copy-on-write operations and *read* transfers data from the OS file cache to the user address space. Fig. 11 leads to a number of interesting observations. In general, the eight-issue mode is not energy efficient as indicated by the elevated Energy×Delay on all of the three OS routines. The application of the one-issue, two-issue, four-issue and six-issue modes yields better trade-off between power and performance. More interestingly, the optimal configuration changes depending on the OS routines. The optimal configuration has the lowest Energy×Delay value. For example, on the one-issue mode, *clock* shows its best Energy×Delay scenario (0.3), while *COW_fault* yields an Energy×Delay value of 0.8.

The heterogeneous Energy×Delay behavior of various OS routines makes a unified adaptation for the whole OS less attractive. However, it provides an avenue to finely tune the OS power/performance knob: the per-OS routine based optimal configuration can be exploited by the hardware to achieve a better OS Energy×Delay trade-off. In practice, a simple profile-driven methodology [17] can be used for finding the optimal configuration for an individual routine in a

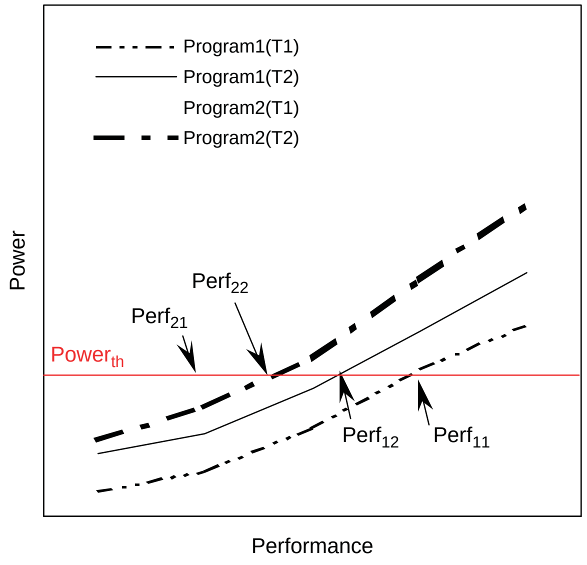

Fig. 10. Effectiveness of energy×delay trade-offs is program dependent.

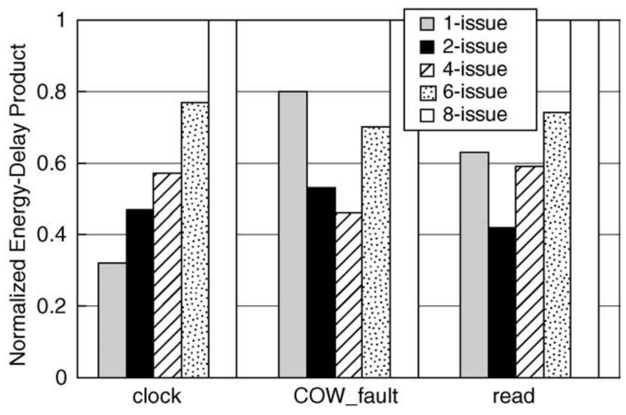

Fig. 11. Energy×Delay of different OS services.

pre-characterization stage. At run-time, the hardware selectively applies the pre-characterized, optimal configuration to an individual OS routine instantaneously, eliminating a search of the configuration space. The optimal adaptation solution can be encoded into each routine with ISA extension. A performance degradation tolerance setting that specifies how aggressively to trade-off additional delay for lower energy can be used to guide configuration selection.

Knowing the nature and functionality of an OS invocation, one can apply Energy×Delay optimizations even more aggressively. In this paper, we consider the following two optimizations (dubbed as OS-aware SDPT w/AO in Section 6).

5.1. Resizing register file

Modern superscalar machines exploit register renaming and use a large register file to eliminate false dependencies between instructions. In many hand-tuned and highly optimized OS routines, however, the true dependencies dominate. In these scenarios, the size of the physical register file can be reduced to save more power. Specifically, we observe that disabling half of the physical registers for the OS routines *utlb*, *timein*, *clock*, *close*, *brk*, *alarm*, *dup*, *pipe*, *ioctl*, *utsys*, *prctl*, and *msync* saves 5–7% of the processor power with no performance loss [23]. Generally, the additional complexity in resizing a register file greatly diminishes the value of any advantage that may be achieved [5]. The proposed routine based OS-aware adaptation scheme can safely and efficiently resize the register file because it guarantees that no physical register is mapped whenever a resizing occurs at the user/OS context switch boundaries.

5.2. OS-aware control flow speculation

Control flow speculation has been widely adopted in today's microprocessor design to exploit the ILP in programs. Nevertheless, the fetches and subsequent processing of misspeculated instructions will waste more energy and cycles [12]. It has been observed that the conventional branch predictors can frequently mispredict the control flow transfers in the exception-driven and short-lived OS execution [37]. In [13], Li et al. propose an OS-aware control flow speculation scheme which allocates a dedicated branch prediction resource to the OS to improve its branch prediction accuracy. In this study, we integrate an OS-aware hybrid predictor [13] with the proposed processor adaptation scheme to further optimize its energy efficiency in light of the exception-driven and non-deterministic OS execution.

6. Power savings and performance evaluation

This section presents power savings as well as performance evaluations of the proposed technique and the baseline adaptation mechanism (described in Section 4) on OS execution. The schemes we compare are: (1) a baseline adaptation scheme with a 2048-cycle sampling window (ADPT with sw = 2048); (2) a baseline adaptation scheme with a fine-grained 128-cycle sampling window (ADPT with

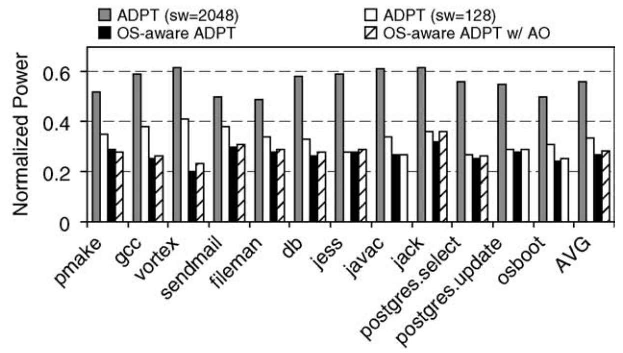

Fig. 12. Normalized power (ADPT with sw = 2048 is sampling-based adaptation with 2048-cycle window, ADPT with sw = 128 is sampling based adaptation with 128-cycle window, OS-aware ADPT is OS routine based adaptation, and OS-aware ADPT w/AO is OS routine based adaptation with aggressive optimizations).

sw = 128); (3) the routine based OS-aware adaptation (OS-aware ADPT); (4) the routine based OS-aware adaptation with aggressive optimizations (OS-aware ADPT w/AO, see Section 3). Fig. 12 shows the average power of the simulated workloads on different schemes. Figs. 13 and 14 show the performance (IPC) and Energy×Delay metric on the same scenario. All values are normalized with respect to the baseline eight-issue machine without implementing any adaptation.

Fig. 12 shows that compared to the coarse-grained sampling technique (ADPT with sw = 2048), the OS-aware ADPT can reduce power more aggressively by being able to accurately capture the exception-driven, short-lived OS activity and matching it with appropriate resources in a timely fashion. For the same reason, the scheme using fine-grained sampling window (ADPT with sw = 128) is also observed to achieve good power savings. The OS-aware ADPT w/AO has a dual impact on power savings: reducing the size of register file drops power while the improved control flow speculation tends to increase power because the pipeline flushing stalls happen less frequently. Intuitively, optimizations such as OS-aware control-flow speculation could increase per-cycle processor power. Nevertheless, it reduces program execution cycles and the total clock power, on which both the processor and software energy largely depends. Therefore, overall it will benefit the targeted program Energy×Delay metric that we try to optimize. Moreover, as can be seen in Fig. 12, one factor

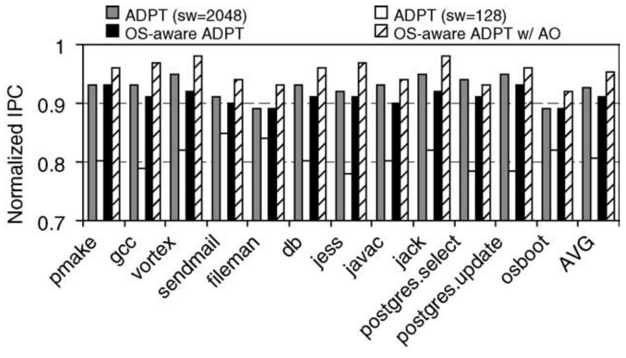

Fig. 13. Normalized IPC.

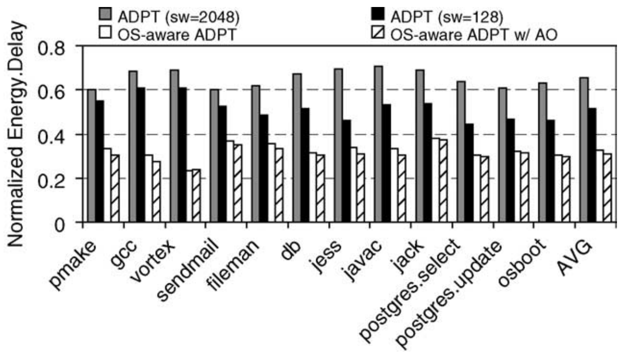

Fig. 14. Normalized Energy×Delay.

does not dominate another by showing drastic changes in power compared with the OS-aware ADPT scheme.

Looking at Fig. 13, one can see that the performance of the OS-aware ADPT is competitive with that of the ADPT (sw=2048), despite that the ADPT (sw=2048) favors the OS performance by overestimating its computational requirement due to the interference of the higher user IPC. Fig. 13 also shows that using the fine-grained window sampling scheme (ADPT with sw=128) measurably degrades performance due to the aggregated adaptation overhead. As described earlier, the OS-aware ADPT does not incur adaptation overheads in the OS. The use of the optimal solution for an individual routine further eliminates the unnecessary adaptations within a routine, leading to a better performance than the existing fine-grained adaptation scheme. Another observation from Fig. 13 is that the OS-aware ADPT w/AO further increases performance by reducing the time spent on processing wrong-path instructions. Note that the *y*-axis begins at 70% normalized IPC in Fig. 13.

The results shown in Fig. 14 indicate the OS-aware ADPT retains performance while reducing power by showing the desirable characteristics when both performance and energy are under consideration. The OS-aware ADPT w/AO further improves the OS Energy×Delay behavior, suggesting that although the aggressive optimizations such as resizing register file may yield an unbalanced machine for many user applications, they produce more energy savings when judiciously applied to certain OS routines.

7. Related work

Previous research [14] employs the OS to reduce power at the system level. Recently, the energy behavior of embedded, real-time operating systems has been studied in [15,16,33,34]. In [7,38], a full-system energy simulator is developed and the necessity of simulating OS energy is quantified. There has been much research [3–5,11,12,20,27,29] focusing on reducing the run-time software power consumption (mostly, user applications). So far, techniques for run-time software power savings exclusively focus on user-only applications. Among those, microarchitecture level power management [3–5,11] has been demonstrated to be an attractive solution for fine-grained program Energy×Delay optimization. It has been observed that by appropriately allocating microarchitectural resources required by the actual program, significant power savings can be achieved with a tolerable performance lost. In [3], Bahar et al. exploit IPC variations in programs to reduce power. Our proposed scheme further explores the IPC variations between the user and the OS and the fine-grained phase changes due to the user/OS context switches. By varying processor fetch and execution rates, Marculescu et al. [17] studied power-performance trade-off based on a profile-driven methodology, which is employed in this study to characterize the per-OS routine based Energy×Delay behavior. In [4,5], the authors propose mechanisms for independently monitoring and adapting multiple microarchitectural structures in one system. By leveraging the pre-characterized Energy×Delay knowledge, our approach avoids the complexity of the simultaneous control and independent operation of multiple adaptive structures.

8. Conclusion

Modern applications spend a significant proportion of their execution time within the operating system, making the OS a major power consumer. To save power, hardware can provide resources that closely match the needs of the software. However, with exception-driven and intermittent execution, it becomes difficult to accurately predict and adapt processor resources in a timely fashion. The novel approach we propose in this paper permits precise hardware reconfigurations for the OS with low overhead and allows fine-grained performance/power tuning at the microarchitectural level. This scheme is orthogonal to and can be integrated with existing techniques proposed for user-only applications to further enhance their efficiency in light of the prevalent, OS-intensive and emerging workloads. With the increasing impact of the leakage power, routine customized aggressive adaptation tends to save more power by safely turning off more transistors. The proposed scheme can be exploited in mobile computing systems for energy saving, as well as in conventional systems for dynamic thermal management.

References

[1] J.A. Redstone, S.J. Eggers, H.M. Levy, An analysis of operating system behavior on a simultaneous multithreaded architecture, in: Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems, 2000.

[2] H. Zeng, X.B. Fan, C. Ellis, A. Lebeck, A. Vahdat, ECOSystem: managing energy as a first class operating system resource, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[3] R.I. Bahar, S. Manne, Power and energy reduction via pipeline balancing, in: Proceedings of the International Symposium on Computer Architecture, 2001.

[4] D. Ponomarev, G. Kucuk, K. Ghose, Reducing power requirements of instruction scheduling through dynamic allocation of multiple data-path resources, in: Proceedings of the International Symposium on Microarchitecture, 2002.

[5] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, M.L. Scott, Integrating adaptive

on-chip storage structures for reduced dynamic power, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2002.

[6] K. Keeton, D.A. Patterson, Y.Q. He, R.C. Raphael, W.E. Baker, Performance characterization of a quad pentium pro SMP using OLTP workloads, in: Proceedings of the International Symposium on Computer Architecture, 1998.

[7] S. Gurumurthi, A. Sivasubramaniam, M. Jane Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, 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.

[8] M. Rosenblum, S.A. Herrod, E. Witchel, A. Gupta, Complete computer system simulation: the simOS approach, IEEE Parallel and Distributed Technology: Systems and Applications 3 (4) (1995).

[9] D. Brooks, V. Tiwari, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: Proceedings of the International Symposium on Computer Architecture, 2000.

[10] 'PostgreSQL', http://www.us.postgresql.org/

[11] A. Iyer, D. Marculescu, Microarchitecture level power management, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10 (3) (2002).

[12] S. Manne, A. Klauser, D. Grunwald, Pipeline gating: speculation control for energy reduction, in: Proceedings of the International Symposium on Computer Architecture, 1998.

[13] T. Li, L.K. John, A. Sivasubramaniam, N. Vijaykrishnan, 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 (ASPLOS), 2002.

[14] L. Benini, A. Bogliolo, S. Cavallucci, B. Ricco, Monitoring system activity for OS-directed dynamic power management, in: Proceedings of the International Symposium on Low Power Electronics and Design, 1998.

[15] K Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Lohout, C. Smit, T.B. Zhang, 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.

[16] T.K. Tan, A. Raghunathan, N.K. Jha, Embedded operating system energy analysis and macro-modeling, in: Proceedings of the International Conference on Computer Design, 2002.

[17] D. Marculescu, Profile-driven code execution for low power dissipation, in: Proceedings of the International Symposium of Low Power Electronics and Design, 2000.

[18] J.L. Hennessy, David A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufman, Los Altos, CA, 1996.

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

[20] D. Brooks, M. Martonosi, Dynamic thermal management for high-performance microprocessors, in: Proceedings of the International Symposium on High Performance Computer Architecture, 2001.

[21] T. Li, L. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan, A. Murthy, Using complete system simulation to characterize SPECjvm98 benchmarks, in: Proceedings of the International Conference on Supercomputing (ICS), 2000.

[22] T. Li, L.K. John, Run-time modeling and estimation of operating system power consumption, in: Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2003.

[23] T. Li, L.K. John, Routine based OS-aware microprocessor resource adaptation for run-time operating system power saving, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2003.

[24] M.K. Gowan, L.L. Biro, D.B. Jackson, Power considerations in the design of the alpha 21264 microprocessor, in: Proceedings of the Design Automation Conference, 1998.

[25] D.H. Albonesi, Dynamic IPC/Clock rate optimization, in: Proceedings of the International Symposium on Computer Architecture, 1998.

[26] R. Joseph, M. Martonosi, Run-time power estimation in high performance microprocessors, in: Proceedings of the International Symposium on Low Power Electronic Device, 2001.

[27] V. Tiwari, S. Malik, A. Wolfe, M.T.C. Lee, Instruction level power analysis and optimization of software, Journal of VLSI Signal Processing 1–18 (1996).

[28] W. Ye, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, The design and use of simplePower: a cycle-accurate energy estimation tool, in: Proceedings of the Design Automation Conference, 2000.

[29] A. Sinha, A. Wang, A.P. Chandrakasan, Algorithmic transforms for efficient energy scalable computation, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2000.

[30] S. Palacharla, N.P. Jouppi, J.E. Smith, Quantifying the complexity of superscalar processors, CS-TR-1996-1328, University of Wisconsin, Madison, WI, 1996.

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

[32] M. Valluri, L.K. John, Is compiling for performance = compiling for power? in: Proceedings of the Fifth Annual Workshop on Interaction between Compilers and Computer Architectures, 2001.

[33] T.K. Tan, A. Raghunathan, N.K. Jha, EMSIM: an energy simulation framework for an embedded operating system, in: Proceedings of the International Conference on Circuits and Systems, 2002.

[34] R.P. Dick, G. Lakshminarayana, A. Raghunathan, N.K. Jha, Power analysis of embedded operating systems, in: Proceedings of the Design Automation Conference, June 2000.

[35] F. Bellosa, The benefits of event-driven energy accounting in power-sensitive systems, in: Proceedings of the Ninth ACM SIGOPS European Workshop, 2000.

[36] A.R. Lebeck, X.B. Fan, H. Zeng, C. Ellis, Power aware page allocation, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[37] T. Li, L.K. John, Understanding control flow transfer and its predictability in java processing, in: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2001.

[38] J.W. Chen, M. Dubois, P. Stenström, Integrating complete-system and user-level performance/power simulators: the simWattch approach, in: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2003.

[39] J. Ousterhout, Why aren't operating system getting faster as fast as hardware? in: Proceedings of the Summer 1990 USENIX Conference, 1990.