Routine based OS-aware Microprocessor Resource Adaptation for Run-time Operating System Power Saving

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 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. Operating System (OS) is a major power consumer in many modern applications execution. This paper advocates a routine based OS-aware microprocessor resource adaptation mechanism targeting run-time OS power savings. Simulation results show that compared with the existing sampling-based adaptation schemes, this novel methodology yields more attractive power and performance trade-off on the OS execution. To our knowledge, this paper is the first step to address the power saving issue of the OS itself, an increasingly important area that has been largely overlooked in the previous studies.

Categories and Subject Descriptors
C.1.3 [**Processor Architectures**]: Other Architecture Styles-*adaptable architectures, pipeline processors*
D.4.m [**Operating Systems**]: Miscellaneous

General Terms
Performance, Design.

Keywords
Low power, adaptive processor, operating system.

1. INTRODUCTION
Today's high-performance microprocessor constitutes of millions of transistors clocked at Giga Hz frequency, which translates to the significant power dissipation. Its performance-driven market and increasingly constrained power budget necessitate the power saving consideration of all components in a system, spanning from circuits to the software running on it.

Operating System (OS) constitutes a major software component of today's complex systems featured 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]. Figure 1 shows that on the average, the OS draws 32% of the total energy (CPU, cache and main memory) during the execution of the 12 studied workloads (see section 2 for detail), making it a major power consumer. The proportion of the OS power consumption is projected to increase due to the increasing demands for system management activities, such as thermal sensor reading, energy accounting and power control for memory and I/O devices [2]. Clearly, in a power constrained environment, OS power saving needs to be addressed. However, previous studies [3, 4, 5] entirely focus on lowing power for user-only applications. To our knowledge, power saving and optimization for the OS itself have received little attention.

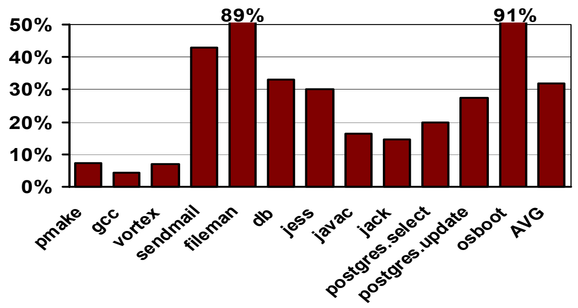

Figure 1. % of Energy Dissipated by the OS

In this study, we explore the adaptation of processor resources to reduce OS power on today's high-performance superscalar processors, which exploit aggressive hardware design to maximize performance across a wide range of targeted applications. It has been observed that program's computational requirement, generally measured by the instruction per cycle (IPC), varies during its execution. By tuning processor resources to be appropriate to the actual needs of the program, significant power savings can be achieved with minimal impact on performance. Figure 2 illustrates the IPC variation over time for *jess*, a SPECjvm98 Java benchmark [19] running on an 8-issue superscalar processor. The benchmark's IPC varies from as low as nearly zero to as high as five, indicating the significant discrepancy in computational requirement during its execution.

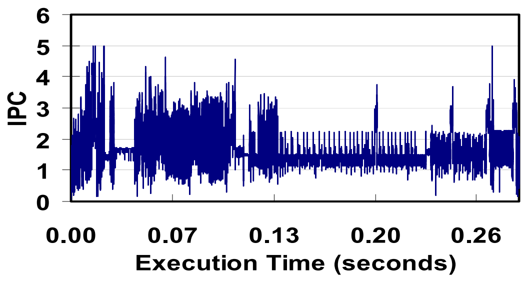

Figure 2. IPC Variation in Benchmark *jess*

One factor that contributes to the widely varying IPC is the frequent OS activity: the inherent instruction level parallelism (ILP) in the OS has been found to be much lower than user applications [6]. The abundant use of serializing instructions and highly-optimized instruction sequence in the OS design limit the available instruction level parallelism (ILP). Figure 3 compares the IPC of user and OS running the 12 studied benchmarks on an 8-issue machine. The OS IPC is 1.2x to 2.4x lower than the user IPC, implying that: (1) the OS does not exploit the superscalar capabilities provided by the wide-issue, aggressive processor as efficiently as user code does; (2) power savings can be achieved by allocating processor resources (with lower computational capabilities) matching the OS requirement.

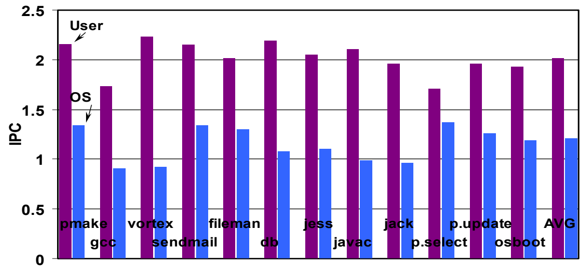

Figure 3. IPC of User and OS on an 8-issue Machine

Current adaptation techniques [3, 4, 5, 11] rely on periodic sampling schemes to match program computational requirement with processor resources. However, we show in this paper that resource adaptation based on sampling window becomes less efficient when applied to the exception-driven and short-lived OS execution. Moreover, for large and sophisticated programs like 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 rationale is that although modern operating systems are large sophisticated software, their complexities are hidden behind a relatively simple interface - a set of OS kernel service routines, which provides a common interface to exercise the OS. The power and performance knowledge of different OS routines can be characterized then exposed to the hardware to finely tune the power/performance knob of the OS at run-time.

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 individual routine, yielding 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 paper is organized as follows: Section 2 describes the experimental methodology and executed benchmarks. Section 3 presents a based line sampling-adaptation scheme and demonstrates the challenges in sampling OS activity. Section 4 proposes the routine based OS-aware microarchitecture adaptation scheme and discusses its benefits. Section 5 presents simulation results. Section 6 discusses related work. In Section 7, we conclude with some final remarks.

2. EXPERIMENTAL METHODOLOGY

As described in Table 1, the baseline machine we consider for this study is an aggressive, 8-issue superscalar processor. To reduce its power consumption, the processor can be reconfigured to the 6-issue, 4-issue, 2-issue and 1-issue modes by reducing its computational capacity. Previous studies [3, 4, 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 6-issue mode, we limit the instruction fetch, decode, issue and retire width to be 6 and disable 1/4 of the IW, ROB and LSQ entries. In the 4-issue, 2-issue and 1-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 1. Baseline Machine

Processor Core	
Technology/V_{dd}/Frequency	0.18 um/2.0V/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, 4-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, 4KB page size
L1 I-Cache	32KB, 2-way, 64B blocks, 1 cycle
L1 D-Cache	32KB, 2-way, 32B blocks, 1 cycle
L2 Cache	512KB, 2-way, 128B blocks, 9 cycle
Memory	256MB, 180 cycle access

We use the complete system power simulator SoftWatt [7]. The SoftWatt tool, built on top of the SimOS infrastructure [8], integrates validated energy model similar to other low level power simulator like Wattch [9]. By leveraging the SimOS cycle-accurate full-system simulation capability, SoftWatt captures both the power and performance characteristics of the unmodified OS running on the machine model described above. In our study, the simulated OS is a full-blown, commercial version of the SGI IRIX 5.3.

We use 12 applications that have different characteristics. The *pmake* is a parallel program development workload. The *vortex* and *gcc* are two benchmarks from the SPECint95. The *sendmail* benchmark forwards emails using the Simple Mail Transport Protocol (SMTP). The *db*, *jess*, *javac* and *jack* are Java programs from the SPECjvm98 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 benchmark. 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. The *fileman* performs popular file management activities, such as copy, remove, tar -cvf and tar -xvf operations. The *osboot* executes a complete OS booting sequence from a root disk image and then generates a shell for the user.

Table 2 shows the OS IPC and power consumption (average over all benchmarks) on the different modes. It can be seen that by reducing processor resources, the 4-issue mode 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 ideal candidate for resource adaptation. Given the assumption that the OS execution can be timely and accurately detected, significant power savings can be achieved (with tolerable performance penalty) by catering appropriate processor computational resource for it.

Table 2. OS IPC and Power on Different Modes

	1-issue	2-issue	4-issue	6-issue	8-issue
IPC	0.88	1.09	1.15	1.19	1.21
Power(W)	6.4	12.2	21.7	31.1	42.8

3. 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 the adaptations. Current sampling-adaptation approaches [3, 11] use a finite state machine (FSM) to specify the transitions between different configurations. For example, Figure 4 shows a FSM for transitioning between the normal mode (8-issue) and the low power modes (6, 4, 2 and 1-issue) described in Section 2. The enabling (E_{xI}) and disabling conditions (D_{xI}) 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 4-issue mode are E_{4I} or $!D_{4I}\&!E_{2I}$ or $E_{4I}\&!E_{2I}\&!E_{1I}$ respectively. In this paper, we consider this adaptation technique as the baseline scheme.

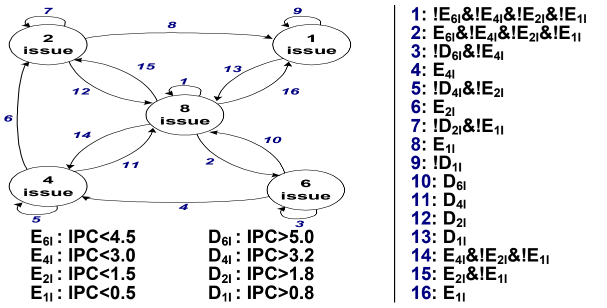

Figure 4. FMS used in Sampling based Adaptation (Trigger Conditions and Thresholds are set and extended according to [3])

At run-time, the estimated program IPC within the previous sampling window (A) serves as the input of FMS to choose the configurations for the current interval (B), as shown in Figure 5.1. 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 of program behavior.

In practice, accomplishing an adaptation can cause performance penalty (latency marked as T_a in Figure 5.1). In the superscalar processor design, IW, LSQ and ROB are implemented with 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 non-trivial amount of time, depending on the number of instructions already in pipeline and the cycles 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 Figure 5.2, reducing sampling window period ($T_h << T_s$) offers capability to capture fine-grained phase changes in execution. However, the aggregated adaptation overhead can be prohibitive. This fact prevents the use of small sampling window without significantly slowing down 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 Figure 5. OS is activated with 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) mechanisms. The IPC discrepancy between user and OS indicates the different computational requirement when the user/OS context switches. When program phase shifts (e.g., due to user/OS interactions), the prior interval becomes a poor estimate for the next.

In the traditional and performance-centric OS design, highly optimized lightweight routines (e.g., faults and interrupt handlers) are usually implemented in order to keep the cycles down. Therefore, many OS service routines show 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. $T_{osd} \geq 2T_s$.

Our characterization shows that there are only 16 OS routines satisfy the above restriction on the duration (≥ 4096 cycles) required by the 2048 cycles sampling interval, a commonly used window granularity to avoid the costly reconfiguration overhead. Figure 6 further illustrates how OS service routines with different duration contribute to the total OS energy dissipation (Note that the x-axis uses logarithmic scale). It is observed that even though some OS services are very efficiently implemented from the execution cycle viewpoint, those 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, a sampling window which is larger than 2048 cycles can not guarantee to resolve these OS activity and adapt processor resource timely to reduce that portion of OS energy (shown on the left side of the dotted line in Figure 6).

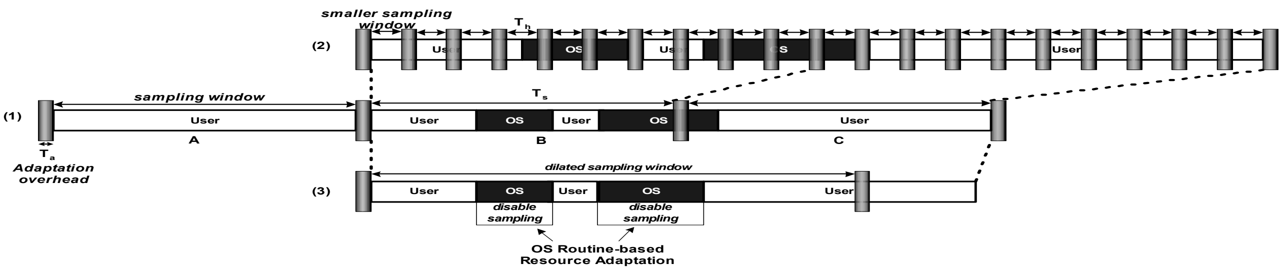

Figure 5. Sampling Policies used in Resource Adaptation

To summarize, a long window interval does not provide the opportunity to switch mode when the program phases change due to the exception-driven, non-deterministic and short-live 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 the frequent adaptations. To reconfigure processor resource for the short-lived OS activity without rising costly adaptation overhead, we propose a routine based OS-aware processor adaptation mechanism targeting on the run-time OS power savings.

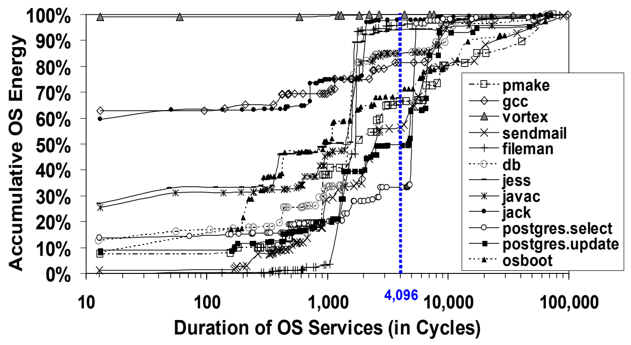

Figure 6. Accumulative OS Energy vs. OS Service Duration

4. PROPOSED SOLUTION: OS-AWARE ROUTINE BASED ADAPTATION

Routine based OS-aware adaptation dedicates to reconfigure processor upon the OS execution. Separating out OS execution can easily be done at run-time by using the Processor Status Register. Processor adaptations occur only at the boundaries of the user/OS context switches, as shown by Figure 5.3. 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 from the user applications and continuously executes using that mode. Therefore, routine based OS-aware adaptation is capable to capture all OS activity timely and accurately, while retaining a zero adaptation overhead in the OS. The separating OS activity out of the regular sampling interval creates the "dilated" sampling window (as shown in Figure 5.3), 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 catered for OS to user program's requirement (as shown in Figure 5.1, window B and C). This is critical since user program after the context switched from OS generally requires the full issuing capabilities of the machine to operate on new data and working set. Though interesting, a detailed analysis of such impact is beyond the scope of this paper.

As described earlier, processor resource adaptation saves power and is detrimental to performance. The goal of such adaptation is to reduce power with the minimum performance lost. Modern operating systems are large, sophisticated software. Individual OS routine performs specific functionality and can exhibit vast variation in computational requirement. A configuration that is good for one piece of code may not turn out to be optimal for another.

The Energy-Delay product (EDP) is a reasonable metric to evaluate energy efficiency, namely, the goal of achieving high performance while minimizing energy consumption. Figure 7 shows the Energy×Delay (normalized with 8-issue mode) of different OS service routines (*clock*, *COW_fault* and *read*) running on different modes. *Clock* processes timer interrupt. *COW_fault* performs page level copy-on-write operations and *read* transfers data from OS file cache to the user address space.

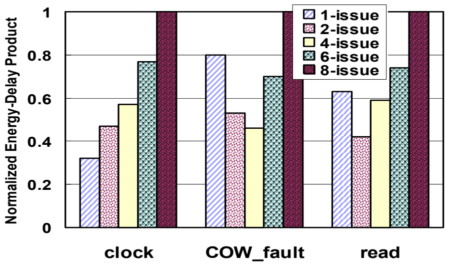

Figure 7. Energy×Delay of Different OS Services

Figure 7 leads to a number of interesting observations. In general, the 8-issue mode is not energy efficient by showing the highest Energy×Delay on all of the three OS routines. The application of the 1-issue, 2-issue, 4-issue and 6-issue modes yields better trade-off between power and performance. More interesting, the optimal configuration (with the lowest Energy×Delay value) changes, depending on the OS routines. For example, on the 1-issue mode, the *clock* shows its best Energy×Delay scenario (0.3), while the *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 exposed to and 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 individual routine in a pre-characterization stage. At run-time, the hardware selectively applies the pre-characterized, optimal configuration to individual OS routine instantaneously, eliminating a search of the configuration space.

Having known 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:

(1) *Resizing Register File*

Modern superscalar machines exploit register renaming and use 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 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. Generally, the additional complexity for resizing a register file greatly diminishes the likelihood to do so [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.

(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. In [13], Li et al. propose an OS-aware control flow speculation scheme which allocates 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 the light of the exception-driven and non-deterministic OS execution.

5. 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 3) on the 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 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). Figure 8 shows the average power of the experimented workloads on different schemes. Figure 9 and Figure 10 show the performance (IPC) and Energy×Delay metric on the same scenario. All values are normalized with respect to the baseline 8-issue machine without implementing any adaptation.

Figure 8 shows that compared with 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 match them with appropriate resources in a timely fashion. For the same reason, 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 double-fold 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 the Figure 8, one factor does not dominate another one by showing drastic changes in power compared with the OS-aware ADPT scheme.

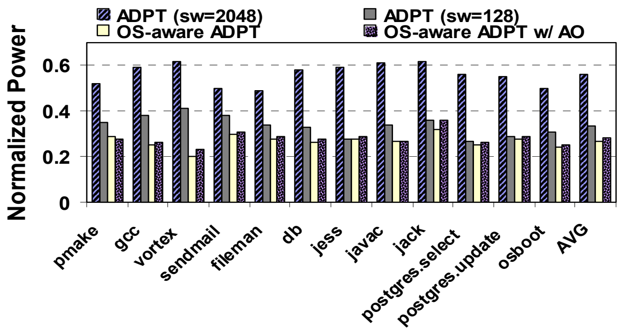

Figure 8. Normalized Power **(ADPT with sw=2048 is sampling-based adaptation with 2048-cycle window, ADPT with sw=128 is sampling based adaptation with 128-cycle window, OS-aware ADPT is OS routine based adaptation, and OS-aware ADPT w/ AO is OS routine based adaptation with aggressive optimizations)**

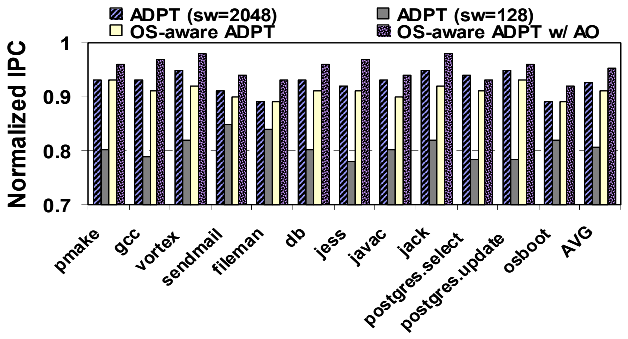

Figure 9. Normalized IPC

Looking at Figure 9, one can see that the performance of 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. Figure 9 also shows that using 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 OS. The use of the optimal solution for 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 Figure 9 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 Figure 9.

The results shown in Figure 10 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, implying that although the aggressive optimizations such as resizing register file may yield unbalanced machine for many user applications, they produce more energy savings when judiciously applied to certain OS routines.

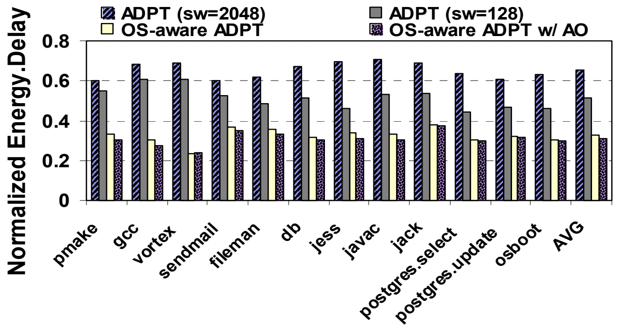

Figure 10. Normalized Energy×Delay

6. RELATED WORK

Previous research [14] employs the OS to reduce power at system level. Recently, the energy behavior of embedded, real-time operating systems has been studied in [15, 16]. In [7], a full-system energy simulator is developed and the necessity of simulating OS energy is quantified. So far, techniques for run-time software power savings exclusively focus on the user-only applications. Among those, microarchitecture level power management [3, 4, 5, 11] has been demonstrated to be an attractive solution for the fine-grained program Energy×Delay optimization. In [3], Bahar et al. exploit IPC variations in program to reduce power. Our proposed scheme further explores the IPC variations between user and 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] study 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 in the simultaneous control and independent operation of multiple adaptive structures.

7. CONCLUSION

Modern applications spend a significant proportion of their execution time within the operating system, making OS a major power consumer. To save power, we argue that hardware should provide resources that closely match the needs of the OS. However, with exception-driven and intermittent execution in nature, it becomes difficult to accurately predict and adapt processor resources in a timely fashion. The novel approach we proposed in this paper permits precise hardware reconfigurations for the OS with low overhead and allows fine-grained performance/power tuning at 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 the 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.

8. ACKNOWLEDEGMENT

This research is partially supported by the National Science Foundation under grant number 0113105; the Defense Advanced Research Projects Agency under contract F33615-01-C-1892; and by the AMD, Intel, IBM, Tivoli and Microsoft Corporations.

9. REFERENCES

[1] J. A. Redstone et al., An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture, In Proc. of ASPLOS, 2000.
[2] H. Zeng et al., ECOSystem: Managing Energy as a First Class Operating System Resource, In Proc. of ASPLOS, 2002.
[3] R. I. Bahar et al., Power and Energy Reduction via Pipeline Balancing, In Proc. of ISCA, 2001.
[4] D. Ponomarev et al., Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Data-path Resources, In Proc. of MICRO, 2002.
[5] S. Dropsho et al., Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power, In Proc. of PACT, 2002.
[6] K. Keeton et al., Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads, In Proc. of ISCA, 1998.
[7] S. Gurumurthi et al., Using Complete Machine Simulation for Software Power Estimation: the SoftWatt Approach, In Proc. of HPCA, 2002.
[8] M. Rosenblum et al., Complete Computer System Simulation: the SimOS Approach, IEEE Parallel and Distributed Technology: Systems and Applications, vol.3, no.4, 1995.
[9] D. Brooks et al., Wattch: A Framework for Architectural-level Power Analysis and Optimizations, In Proc. of ISCA, 2000.
[10] "PostgreSQL", http://www.us.postgresql.org/
[11] A. Iyer et al., Microarchitecture Level Power Management, IEEE Transactions on VLSI, vol. 10, no. 3, 2002.
[12] S. Manne et al., Pipeline Gating: Speculation Control for Energy Reduction, In Proc. of ISCA, 2001.
[13] T. Li et al., Understanding and Improving Operating System Effects in Control Flow Prediction, In Proc. of ASPLOS, 2002.
[14] L. Benini et al., Monitoring System Activity for OS-directed Dynamic Power Management, In Proc. of ISLPED 1998.
[15] K. Baynes et al., The Performance and Energy Consumption of three Embedded Real-Time Operating Systems, In Proc. of CASES, 2001.
[16] T. K. Tan et al., Embedded Operating System Energy Analysis and Macro-modeling, In Proc. of ICCD 2002.
[17] D. Marculescu, Profile-Driven Code Execution for Low Power Dissipation, In Proc. of ISLPED, 2000.
[18] J. L. Hennessy et al., Computer Architecture: A Quantitative Approach, Morgan Kaufman Publishers, 1996.
[19] SPEC JVM98 Benchmarks, http://www.spec.org/jvm98/