OS-aware Tuning: Improving Instruction Cache Energy Efficiency on System Workloads

Tao Li
Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida, 32611
taoli@ece.ufl.edu

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

Abstract

Low power has been considered as an important issue in instruction cache (I-cache) designs. Several studies have shown that the I-cache can be tuned to reduce power. These techniques, however, exclusively focus on user-level applications, even though there is evidence that many commercial and emerging workloads often involve heavy use of the operating system (OS). This study goes beyond previous work to explore the opportunities to design energy-efficient I-cache for system workloads. Employing a full-system experimental framework and a wide range of workloads, we characterize user and OS I-cache accesses and motivate OS-aware I-cache tuning to save power. We then present two techniques (OS-aware cache way lookup and OS-aware cache set drowsy mode) to reduce the dynamic and the static power consumption of I-cache. The proposed OS-aware cache way lookup reduces the number of parallel tag comparisons and data array read-outs for cache accesses to save dynamic I-cache power in a given operation mode. The proposed OS-aware cache set drowsy mode puts I-cache regions that are only heavily used by another operation mode to reduce leakage power. The proposed mechanisms require minimal hardware modification and addition. Simulation based experiments show that with no or negligible impact on performance, applying OS-aware tuning techniques yields significant dynamic and static power savings across the experimented applications. To our knowledge, this is the first work to explore cache power optimization by considering the interactions of application-OS-hardware. It is our belief that the proposed techniques can be applied to improve the I-cache energy efficiency on server processors mostly targeting on modern and commercial applications that heavily invoke OS activities.

1. Introduction

Power dissipation is considered as a major impediment in today's high performance microprocessor designs. Caches account for a sizeable fraction of the total power consumption of microprocessors. High performance cache accesses dissipate significant dynamic power due to charging and discharging highly capacitive bit lines and sense amps [2]. Moreover, on-chip caches constitute a significant portion of the transistor budget of current microprocessors. With the continued scaling down of threshold voltages, static power due to leakage current in caches grows rapidly. Clearly, with the increasingly constrained power budget of today's high performance microprocessors, low power has been considered as an important issue in cache designs. In this paper, we focus on techniques to reduce both dynamic and static power of instruction cache (I-cache).

In general, processor I-cache is designed to accommodate a wide range of applications. Nevertheless, it has been observed that the performance of a given I-cache architecture is largely determined by the behavior of the application using that cache [4, 5]. To reduce power, previous studies [6, 7, 8, 9, 10, 1, 11, 12, 13, 14, 4, 15, 16] proposed adapting I-cache to the need of application's demand. These techniques, however, exclusively focus on user-level applications, even though there is evidence that many system workloads (e.g., database, web and file/e-mail servers) often involve heavy use of the operating system (OS) [3, 18, 33].

During system workload execution, both user applications and OS contribute to power dissipation. Previous studies [17, 18, 33] shown that without considering the impact of OS, performance evaluations on system workloads can only capture incomplete scenarios. To understand the impact of OS on the I-cache power dissipation, we run fifteen benchmarks with various OS activity (see Section 2 for benchmarks description) on a full-system power simulation framework [18] and breakdown the I-cache (32KB, 4-way set associative and 32-byte cache line) power into user applications and OS components.

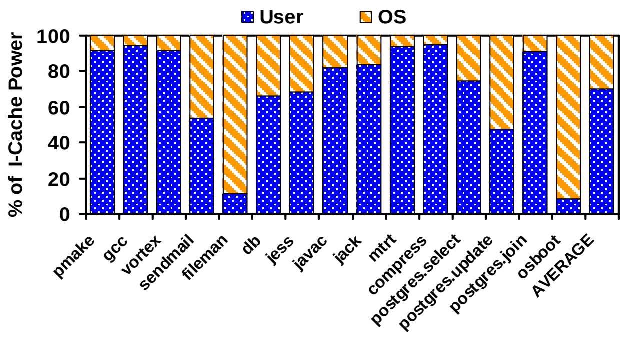


Figure 1. I-Cache Power Breakdown: User vs. OS

Figure 1 shows that OS can highly impact processor I-cache power on modern and emerging applications, such as e-mail (*sendmail*), file (*fileman*), Java (*db*, *jess*) and database (*postgres.select*, *postgres.update*) programs. On the average, OS accounts for 30% of total I-cache power on the 15 experimented workloads. The proportion of OS energy overhead is likely to continuously grow in the future due to other emerging system administrative activities, such as thermal sensor reading, energy accounting and power mode control for memory and I/O devices [19, 20]. Therefore, it is necessary to consider the OS for I-cache power modeling and optimization.

Adhering to this philosophy, this paper explores the opportunities to design low power I-cache by considering the interactions of application-OS-hardware. We start from characterizing user and OS I-cache access behavior to identify power saving opportunities. We observe that in a system that frequently invokes OS activity, instruction blocks from user applications and OS often interleave and co-exist within I-cache that is shared by all processes.

To ensure proper operation and protect the OS from errant users, modern processors and operating systems provide two separate modes of operation: user mode and privileged mode. Processor executes user processes in user mode. Whenever the OS is invoked (by a trap or an interrupt/exception), the hardware switches to privileged mode. The OS always switches back to user mode before passing control to a user program.

The semantics of dual mode operation provides opportunities to save the dynamic power of I-cache access: without affecting the performance and the correctness of program execution, I-cache lookups for user applications can bypass caches lines that store OS code and vice-versa. Therefore, the number of parallel tag comparisons and data array read-outs needed to fulfill a set-associative I-cache access can be reduced, implying less dynamic power dissipation per access. Moreover, we find that a significant fraction of I-cache regions are only heavily accessed in one operation mode. This characteristic can be exploited to reduce I-cache leakage power: when processor executes in one mode, cache regions that are only frequently accessed in another mode can be put into lower power state.

To explore these power saving opportunities, we propose two OS-aware tuning techniques - OS-aware cache way lookup and OS-aware cache set drowsy mode - to improve the I-cache energy efficiency for system workloads. We show in this paper that with very simple hardware modification and addition, OS-aware I-cache tuning exhibits promising dynamic and static power reduction. More attractively, the OS-aware tuning yields no or negligible impacts on performance. Since system performance is sensitive to that of the OS, the proposed techniques preserve merits especially valuable for the energy-efficient, high performance server processor I-cache designs.

The rest of this paper is organized as follows: Section 2 describes the experimental framework, methodology and benchmarks. Section 3 characterizes user applications and OS I-cache access behavior to identify power saving opportunities. Section 4 proposes two OS-aware tuning techniques to improve I-cache energy efficiency. Section 5 evaluates the impact of proposed techniques on power and performance. Section 6 discusses related work. Finally, Section 7 concludes with some final remarks.

2. Experimental Methodology

In this study, we use energy-aware, full-system simulation driven by a wide range of applications with various OS activity. This section describes the simulation framework, the machine architecture modeled and the workloads executed.

2.1 Framework and System Configuration

We use a modified version of the complete system power simulator SoftWatt [18] that models the power dissipation of the CPU, memory hierarchy and I/O subsystems. The SoftWatt framework, built on top of the SimOS [21] infrastructure, integrating energy models similar to Wattch [34] into the full-system simulator. By leveraging the SimOS cycle-accurate, full-system simulation capability, the framework captures complete performance and power characteristics on system workload execution. The simulated OS is a commercial version of the SGI IRIX 5.3 (a modern UNIX variant).

The simulated architectural model is an 8-issue superscalar processor with instruction latencies as in the MIPS R10000. The memory subsystem consists of a split L1 instruction and data caches, a unified L2 cache and main memory. The simulated machine also includes a scaled SCSI HP97560 disk model. The described architecture is simulated cycle by cycle. The execution and power dissipation of both user applications and operating system are modeled. The detailed configuration of the simulated machine architecture can be found in [18].

2.2 Benchmarks

We use fifteen applications that have different characteristics. This section provides a brief overview of the selected applications. *Pmake* is a parallel program development workload that is a variant of the compiled phase of the modified Andrew benchmark employed in [22]. *Vortex* is a database manipulation code and *gcc* is a compiler code from the *SPECint95* benchmark suite. The *sendmail* benchmark forwards emails messages to local accounts on the system using the Simple Mail Transport Protocol (SMTP). The sizes of the messages vary from 1KB to 1.5MB. The *fileman* program performs file management activities, such as copy, remove, chmod, tar -cvf and tar -xvf. A set of six Java applications (*db*, *jess*, *javac, jack, mtrt* and *compress*) from the *SPECjvm98* [23] suite is executed using a commercial JDK from Sun Microsystems. We also use three benchmarks that run on a relational database management system (DBMS) engine- PostgreSQL [24]. The database is populated with relational tables for the TPC-C benchmark [25]. We evaluate the execution of three specific queries on this data set. 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 form the root disk image and then generates a shell for the root user. The *postgres.select*, *postgres.update, postgres.join* benchmarks are simulated for billion-instruction execution. All other applications are all executed to completion. The OS activity in the selected benchmarks ranges from 6% in *compress* to as high as 92% in *fileman*.

3. User/OS I-Cache Accesses Characterization

During system workload execution, instructions from user applications and OS are fetched into I-cache and exercise on the processor alternately, as shown in Figure 2(a). OS is activated either 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. Among multiple processes that must all share the same I-cache, instruction blocks from the OS co-exist with those from user processes. Previous studies analyzed the impact of inter-mingling of user and OS instructions in the I-cache and found that interferences between the two degrade performance. The interest of our characterization in this study, however, is to identify the power saving opportunities.

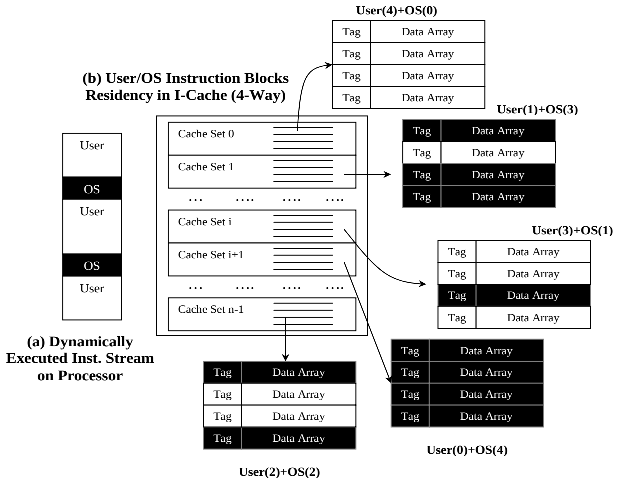


Figure 2. I-Cache Set Categories based on User/OS Instruction Blocks Residency (Assuming a 4-way I-Cache)

To achieve low miss rates for typical applications, modern microprocessors employ set-associative I-Caches. In a system that frequently invokes OS, there is a high possibility that user and OS code simultaneously reside within the same cache set. As illustrated in Figure 2(b), in a 4-way set-associative I-cache, based on user/OS residency, cache sets can be classified as: (1) user code occupies all of the four cache lines (*User(4)+OS(0)*); (2) user occupies three cache lines and OS resides in one cache line (*User(3)+OS(1)*); (3) user and OS each occupy two cache lines (*User(2)+OS(2)*); (4) user resides in one cache line and OS occupies three cache lines (*User(1)+OS(3)*); and (5) OS dominates all of the four cache lines (*User(0)+OS(4)*).

To protect OS from malfunctioning programs, modern processor architectures support user and privileged mode operations. Processor executes user applications in user mode and OS instructions can only be exercised in privileged mode. At any time, processor runs in one of the two modes. Therefore, OS instructions in I-cache will not be selected when processor runs in user mode and vice versa. The semantic of dual-mode operation implies opportunities to save the dynamic power of set-associative I-cache accesses: when processor runs in one mode, the number of parallel cache way lookups can be reduced by filtering out accesses to cache lines holding instruction blocks that are only executed in another mode. For example, to access cache sets in the *User(2)+OS(2)* category, processor really needs to only perform two parallel cache way lookups. Similarly, in the OS mode, if the processor is aware of user/OS instruction block residency, 75% of parallel cache way lookups can be reduced when the processor accesses cache sets in the *User(3)+OS(1)* category.

To evaluate the opportunities to reduce cache way lookups by exploiting the information of user/OS cache blocks residency within cache sets, we count the frequencies of I-cache accesses to each cache set category during program execution. The results are summarized in Table 1. Not surprisingly, during system workload execution, a significant fraction of I-cache accesses encounters cache sets in which both user and OS instruction blocks reside (marked with categories II, III, IV and shown by the shaded columns in Table 1). On benchmarks *gcc* and *vortex*, user mode dominates execution cycles. Still, more than 25% of I-cache references access cache sets in categories II, III, and IV. Interestingly, on benchmark *compress*, 97% of I-cache accesses encounter OS cache lines, even though OS accounts for only 6% of program execution time. This is because *compress* has small I-cache footprint and a few most frequently accessed cache sets (hot-spot) are mapped by codes from both user and kernel spaces. On benchmarks *fileman* and *osboot* where OS mode dominates, there are still 35% and 16% of I-cache references that touch user blocks. Table 1 shows that on the average, 56% of program I-cache references access cache sets in categories II, III and IV, indicating there are abundant opportunities to reduce the number of parallel cache way lookups (and associated dynamic power) by incorporating user/OS operation mode in I-cache designs.

Table 1. I-Cache (32KB, 4-Way and 32Byte Block) Accesses Categorized by User/OS Residency

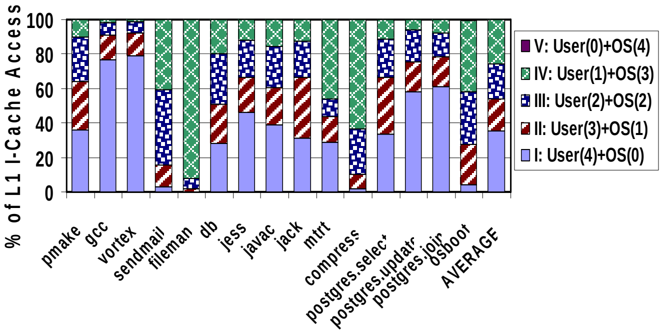
Benchmarks	**% in Program I-Cache Accesses**				
	I	***II***	***III***	***IV***	***V***
	User(4) +OS(0)	***User(3) +OS(1)***	***User(2) +OS(2)***	***User(1) +OS(3)***	***User(0) +OS(4)***
pmake	33	26	25	11	5
gcc	73	17	7	2	1
vortex	72	20	6	1	0
sendmail	1	8	28	33	30
fileman	0	0	2	33	65
db	19	17	28	27	10
jess	32	21	23	20	5
javac	32	22	24	18	4
jack	26	34	26	14	1
mtrt	27	17	11	44	1
compress	2	8	25	64	1
postgres.select	25	27	21	22	4
postgres.update	28	19	17	20	17
postgres.join	55	18	13	12	1
osboot	0	2	4	9	84
AVERAGE	28	17	17	22	16

Previous research [15, 16] found that during program execution, not all cache regions are accessed frequently. To save energy, the less frequently accessed cache regions can be put into lower power state with tolerable performance loss. The dual-mode operation provides yet another opportunity: if cache regions are heavily accessed by processor in only one operation mode, then those cache regions can be put into lower power state when the processor runs in another mode.

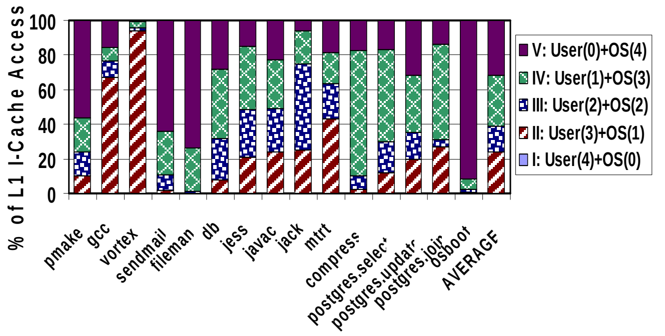
To identify cache regions heavily accessed only in one of the two operation modes, we further breakdown the characterization shown in Table 1 into user and OS parts. The results are shown by Figure 3 (a) and (b).

Figure 3 (a) and (b) show both user and OS access cache sets in the categories II, III and IV frequently. Interestingly, we find that cache sets in the category *User(4)+OS(0)* are heavily accessed only in user mode. In contrast, cache sets in the category *User(0)+OS(4)* are heavily accessed in OS mode but they are rarely accessed in user mode. On the average, only 0.08% of user I-cache accesses touch cache sets in the category *User(0)+OS(4)*. The percentile of OS I-cache accesses that encounter cache sets in the category *User(4)+OS(0)* is merely 0.11%. The above characterization implies that during user execution, cache sets in the category *User(0)+OS(4)* can be put into lower power state. On the other hand, when processor runs in OS, cache sets in the category *User(4)+OS(0)* can remain in lower power state.

To summarize, in this section, we characterize system workload user/OS I-cache accesses categorized by the user/OS residency. We find that dual-mode operation opens additional opportunities to save processor I-cache power. We discuss how I-cache design can explore these opportunities to achieve low power in the following sections.



(a) User I-Cache Accesses



(b) OS I-Cache Accesses

Figure 3. User and OS I-Cache Accesses Categorized by User/OS Residency

4. OS-aware I-Cache Tuning

This section proposes two simple mechanisms to improve I-cache energy efficiency for system workloads.

4.1 OS-aware Cache Way Lookup

In a set associative cache, the number of cache way lookups largely determines the dynamic power of a cache access. A conventional 4-way set associative cache requires four tag comparisons and four data array read-outs for a cache access. Nevertheless, during user execution, performing tag comparisons and data array read-outs for OS cache lines are unnecessary and waste extra dynamic power. Therefore, processor operation mode can be integrated with I-cache design to reduce the number of cache way lookups (and hence dynamic power) on cache accesses.

Figure 4 illustrates architectural modifications to support OS-aware cache way lookup. A bit called cache way mode bit is added to each cache line. With the cache way mode bit (e.g., 0 for OS and 1 for user), we are able to differentiate between cache way stores instructions on behalf of the operating system, and of one that stores instructions on behalf of the user applications. When a cache line is uploaded to I-cache the first time, its cache way mode bit is generated, depending on the processor operation mode. The cache way mode bit will keep unchanged unless the associated cache line is replaced. The current machine execution mode in processor status register (PSR) is used to compare with cache way mode bit to decide whether a cache way needs to be accessed in a given operation mode. The results of comparisons are used to generate enable signals (assuming active low) to circuitry such as tag and data array access logic, tag comparators, data array sense amps and output drivers. As can be seen from Figure 4, the hardware modification and addition needed to support OS-aware cache way lookup is simple.

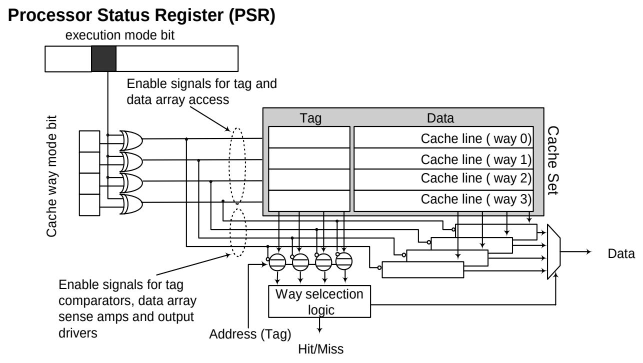


Figure 4. Hardware Modification/Addition Required to Implement OS-aware Cache Way Lookup

The generation of above enable signals is not in the critical path of I-cache access because once generated, they remain unchanged (due to the one-to-one hard-wired mapping between each cache way mode bit and each cache block) unless a cache line replacement (due to a cache miss) occurs or the processor switches mode. When a cache miss occurs, the requested cache line is retrieved from the next level of memory hierarchy and is immediately forwarded to processor for execution. The corresponding cache mode bit needs to be accessed and then updated. The latency to access and update cache way mode bit array and regenerate cache way access enable signals can be overlapped with processor execution. Similarly, the latency of regenerating cache way access enable signals due to processor mode changes can be easily hidden as well due to the inherent cost and the low frequency of user/OS context switches.

Note that the correctness of OS-aware cache way lookup is ensured by the dual-mode operation semantic and the precise exception handling mechanism. Processor switches mode to OS upon handing an exception or interrupt or upon handling a TRAP instruction (usually used to implement all system calls by OS), which all raises an exception. To handle precise exceptions, the processor pipeline must drain before OS code execution can begin. To return the processor to user/unprivileged mode, most architectures use a privileged instruction (return-from-exception) that performs this step in an atomic manner. Therefore, even on processors with out-of-order and speculative execution, instructions from user and OS will not be fetched from I-cache and executed in pipeline simultaneously.

For some systems, there could be certain circumstances where user-defined signal handlers were performed within the OS. Also, it is possible that certain runtime actions/exceptions of user code, may be trapped by the hardware, given to the OS, and the OS executes the user code in OS mode. For user code that dedicatedly runs in OS mode, OS-aware cache way lookups treat it as if it was OS code. However, for user code that can run in both user and OS mode, additional attention is required to ensure correctness. For example, a special purpose register (1 bit) can be added to enable/disable OS-aware cache way lookup by gating the cache way lookup enable signals. An instruction writes to that special purpose register to set (or reset) OS-aware cache way lookup. Two such instructions are placed at the boundaries of the above code region so that OS-aware cache way lookups are disabled before the code region execution starts and are resumed after the code region execution completes. During the above code region execution, full cache way lookups are required and no power saving is achieved. Because this situation happens infrequently, its impact on performance as well as energy saving is negligible. We never encountered this situation during all applications simulation on the OS we studied in this paper.

We measure the reduction of cache way accesses on a 4-way set-associative I-cache by employing OS-aware cache way lookup, as shown in Figure 5. The results are shown for user, OS and the aggregated cache accesses on each benchmark. On benchmarks *gcc* and *vortex* where the OS frequently accesses cache sets in the category *User(3)+OS(1)*, OS-aware cache way lookup reduces the number of cache way accesses in OS significantly. On the other hand, the number of cache way lookups during the user execution on benchmark *sendmail* is largely reduced due to its high access frequencies to cache sets in the categories *User(1)+OS(3)* and *User(2)+OS(2)*. On the average, the proposed technique reduces cache way lookups in user, OS and aggregated I-cache accesses by 34%, 35% and 35% respectively, implying significant I-cache dynamic power saving.

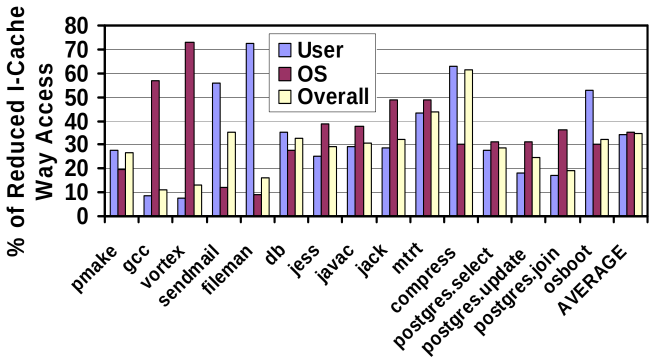


Figure 5. % of Reduced I-Cache Way Accesses by OS-aware Cache Way Lookup

4.2 OS-aware Cache Set Drowsy Mode

Caches comprise a large portion of the on-chip transistor budget. Due to CMOS technology scaling, static power due to leakage current is gaining in importance in I-cache power dissipation. For example, Agarwal et al. [26] report that leakage energy accounts for 30% of L1 cache energy for a 0.13-micro process technology. In a 0.07 micron process, ITRS predicts leakage may constitute as much as 50% of total power budget [31]. These make efforts at leakage control essential to maintain control of I-cache power on current and next generations of processors.

To reduce cache leakage power, researchers [16, 27] have proposed turning off the unlikely used cache lines using gated-V_{dd} technique [11]. While the gated-V_{dd} technique is efficient in saving leakage, the disconnected cache line loses its state and needs to be fetched from L2 cache, causing performance penalty and dynamic power consumption due to an extra L2 access. Alternatively, cache lines can be put into a low-leakage drowsy mode to save power by exploiting the short-channel effects on dynamic voltage scaling [15]. Unlike the gated-V_{dd}, in drowsy mode, the information in the cache line is preserved. However, the cache line in drowsy mode must be reinstated to a high-power mode before its contents can be accessed. The performance penalty of accessing a drowsy cache line is an extra cycle to restore the full voltage for that cache line.

Recent studies show that state-preserving drowsy cache techniques are preferable for leakage control in L1 caches where high performance is a must. Since system performance is sensitive to that of the OS, our objective here is to reduce power yet preserve high performance. Therefore, in this paper, we explore the opportunity of integrating OS-aware cache tuning with a state-preserving, leakage control mechanism. The rationale is to put cache regions that heavily accessed in only one operation mode into drowsy state when processor runs in another mode. A key issue is to classify or identify which cache regions are “hot” in one operation mode but stay “cool” in another operation mode.

The user/OS I-cache accesses on system workloads show that the intra-cache set user/OS residency can be used as proximity for the above classification. During OS execution, cache sets in the category *User(4)+OS(0)* are infrequency accessed and can be put into drowsy state. Similarly, during user mode execution, cache sets in the category *User(0)+OS(4)* can remain in drowsy state.

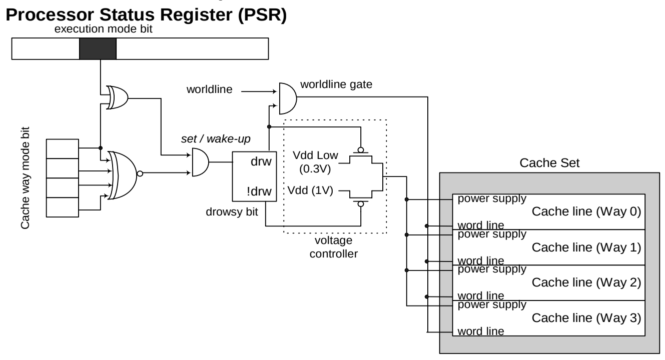


Figure 6. Control Circuitry to Implement OS-aware Cache Set Drowsy Mode

Figure 6 illustrates the control circuitry to implement OS-aware cache set drowsy mode. To control memory cells leakage power, we use circuit technique proposed in [15]. A drowsy bit is used to control the supply voltages to the memory cells within a cache set. For a 0.07 micron process with normal supply voltage (V_{dd}) of 1.0V, the threshold voltage (V_{dd} Low) needed to preserve the state of memory cells is about 0.3V [15]. Depending on the state of the drowsy bit, all cache lines within a cache set can be put into either the high power active state or the low leakage drowsy state.

In Figure 6, if all cache way mode bits within a cache set are identical (e.g., 0000 or 1111) and they are different with the current processor mode, the whole cache set is put into drowsy mode. This control logic puts cache sets in the category *User(4)+OS(0)* to drowsy mode during OS execution. When context switches back to user, cache sets in the category *User(4)+OS(0)* are waken up and cache sets in the category *User(0)+OS(4)* are then put into drowsy state. Moreover, if an OS (or a user) cache miss occurs on a cache set in the category *User(4)+OS(0)* (or *User(0)+OS(4)*), the cache set is waken up due to the change of intra-cache set user/OS residency.

Whenever a cache set is accessed, the drowsy bit associated with it is checked. If the cache set stays in active mode, the ongoing cache access acts normally. Otherwise, if a drowsy cache set is encountered, the drowsy bit is cleared; causing the supply voltage resorted back to the normal V_{dd} during the next cycle. The data can be accessed during consecutive cycles. The wordline gating circuit is used to prevent unchecked accesses to a drowsy set which could destroy the memory's contents.

In Figure 6, OS-aware cache set drowsy mode uses a shared source (cache way mode bit) to control leakage, reducing the cost of drowsy I-cache implementation. We count the percentile of cache sets can be put into drowsy state on user, OS and aggregated execution by employing the leakage control method described. The results are shown in Figure 7. On the average, 17% of I-cache sets can be put into drowsy mode during user execution while the percentage of I-cache sets remain in drowsy state during OS execution is 35%. Overall, 22% of I-cache sets can be put into drowsy mode during program execution. On most benchmarks, we observed that larger fraction of cache regions can remain in drowsy mode during OS execution. This is because that although OS is large and sophisticated software, OS execution is usually dominated by a small fraction of highly invoked service routines [18]. Therefore, a sizeable fraction of the I-cache is not accessed by the OS during its execution.

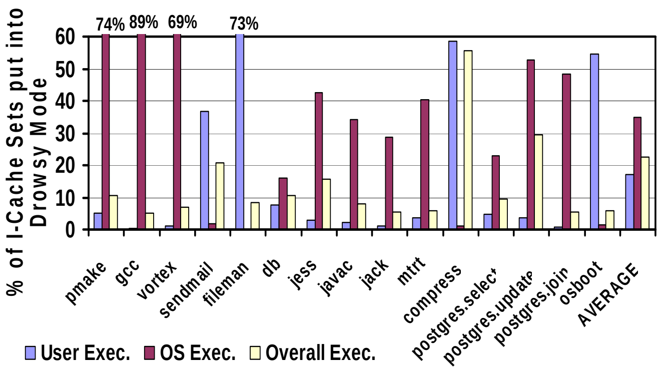


Figure 7. % of I-cache Sets can be put into Drowsy State by Using Leakage Control Illustrated in Fig. 6

Table 2. % of I-Cache Accesses to Drowsy Sets and Average Number of Reinstated Drowsy Sets

Benchmarks	% of User Accesses to Drowsy Sets (in the category User(0)+OS(4))	Avg. Num. of Drowsy Sets Reinstated in User	% of OS Accesses to Drowsy Sets (in the category User(4)+OS(0))	Avg. Num. of Drowsy Sets Reinstated in OS
pmake	0.01	0.18	0.10	0.16
gcc	0.00	0.01	0.21	0.04
vortex	0.00	0.00	0.05	0.01
sendmail	0.15	1.40	0.01	0.10
fileman	0.22	0.92	0.00	0.01
db	0.05	0.10	0.09	0.09
jess	0.04	0.04	0.09	0.04
javac	0.02	0.04	0.14	0.07
jack	0.01	0.01	0.28	0.06
mtrt	0.00	0.02	0.11	0.03
compress	0.00	0.01	0.03	0.01
postgres.select	0.04	0.15	0.23	0.26
postgres.update	0.20	0.30	0.20	0.33
postgres.join	0.01	0.03	0.08	0.04
osboot	0.47	2.17	0.00	0.11

As described earlier, an extra cycle is needed to access cache sets in drowsy mode, implying a performance penalty. To effectively save power while maintaining high performance, both the number of accesses to the drowsy sets and the number of drowsy cache sets reinstated to the high power mode should be small. Table 2 summarizes the percentage of I-cache accesses to the drowsy sets and the average number of drowsy sets that are waken-up. The data are shown for both user and OS execution. As can be seen from Table 2, the possibilities to access a drowsy cache set in both operation modes are extremely low (< 0.1% in most cases), indicating negligible performance lost due to drowsy cache sets wake ups. Additionally, most of the drowsy sets remain in the low power state during a given mode execution by showing very small fraction of reinstated drowsy sets.

It should be noticed that although the intra-cache set user/OS residency provides a good approximation on user/OS access frequencies to that cache set, this heuristic may be too conservative from the perspective of power saving. We further explore the directly using of cache set access frequencies from different operation mode as the metric to control cache set drowsy mode.

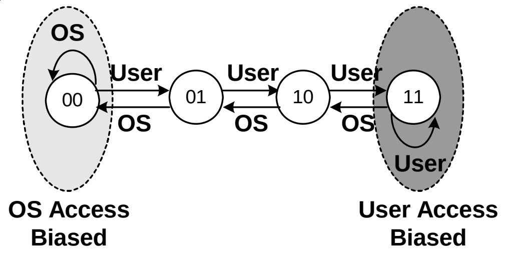


Figure 8. The 2-bit Counter and Finite State Machine to Implement User/OS Access-biased Classification

This user/OS access-biased classification is similar to the one that has been used in classifying the biases of branches. To be more specific, a finite state machine formed by a 2-bit saturating up/down counter is used by each cache set to keep tracking the accesses from user and OS execution, as shown in Figure 8. Whenever an access to that cache set comes from user mode, the associated counter is increased by 1. On the other hand, when an access to that cache set from the OS mode occurs, the counter is decreased by 1. As a result, cache sets with counter's value equals to 3 indicate they are user access-biased and cache sets with counter's value equals to 0 are classified as OS access-biased. During user execution, the OS access-biased cache sets are put into drowsy mode. On the other hand, when processor runs in OS, the user access-biased cache sets are put into drowsy mode.

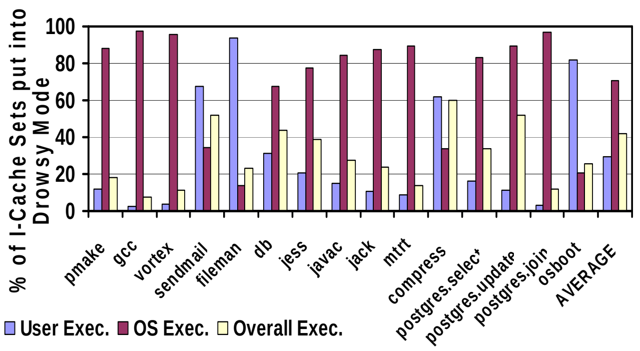


Figure 9. % of I-cache Sets put into Drowsy State by using User/OS Access-biased Classification

Figure 9 shows the percentile of cache sets can be put into drowsy state on user, OS and aggregated execution by employing the less restricted user/OS access-biased leakage control mechanism. One can see that the access-based classification has the capability of putting more cache sets into drowsy state. This is because access-based scheme can identify all cache sets that can be classified by the residency-based scheme. Additionally, access-based scheme captures more scenarios. For example, it could be possible that a cache set has both user and OS blocks reside in it but are accessed frequently only in one operation mode. On the average, 29% of I-cache sets can be put into drowsy mode during user execution while the percentage of I-cache sets can be put into drowsy state during OS execution is 71%. Overall, 42% of I-cache sets can remain in the drowsy state during program execution.

Table 3 further summarizes the percentage of I-cache accesses to the drowsy sets and the average number of drowsy sets that are waken-up by using the access-based classification. As can be seen, both numbers are higher than the residency-based classification but are still low enough to incur observable performance degradation.

Table 3. % of I-Cache Accesses to Drowsy Sets and Average Number of Reinstated Drowsy Sets using Access-Based Classification

Benchmarks	% of User Accesses to Drowsy Sets (User(0)+OS(4))	Avg. Num. of Drowsy Sets Reinstated in User	% of OS Accesses to Drowsy Sets (User(4)+OS(0))	Avg. Num. of Drowsy Sets Reinstated in OS
pmake	0.06	1.06	0.69	1.07
gcc	0.05	0.17	0.81	0.16
vortex	0.03	0.04	0.32	0.04
sendmail	0.44	4.13	0.50	4.14
fileman	3.05	12.79	0.34	11.15
db	0.69	1.33	1.31	1.30
jess	0.68	0.70	1.44	0.67
javac	0.26	0.58	1.18	0.57
jack	0.26	0.28	1.26	0.26
mtrt	0.07	0.25	0.98	0.25
compress	0.15	0.54	2.59	0.53
postgres.select	0.24	0.82	0.70	0.82
postgres.update	0.45	0.67	0.41	0.68
postgres.join	0.04	0.20	0.42	0.21
osboot	1.18	5.45	0.11	5.42

5. Power and Performance Evaluation

This section provides results showing the I-cache power savings as well as the performance impact due to the proposed OS-aware I-cache tuning. By default, the power and performance numbers are normalized to the base line I-cache and machine configuration described in Section 2. In our simulation, we account for the energy overhead due to hardware modification and addition to implement the proposed OS-aware tuning.

Figure 10 shows the normalized I-cache dynamic power after employing the OS-aware cache way lookup scheme. On the average, the OS-aware cache way lookup can save 29% and 30% of I-cache dynamic power on user and OS execution respectively. The aggregated dynamic power saving of this technique is 30%. Looking at Figure 5 and Figure 10, one can see that dynamic power saving is largely correlated with the reduced cache way accesses. It should be noticed that this 30% of dynamic power saving is achieved without any impact on performance. This feature is especially valuable for the OS since system performance is sensitive to that of the OS and the processor energy overhead caused by performance degradation can easily offset the benefit of power saving in I-cache.

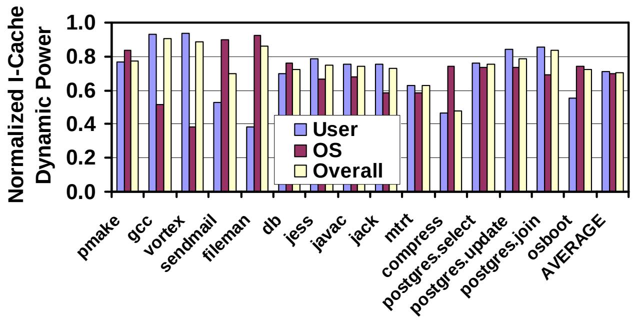


Figure 10. % of I-Cache Dynamic Power Savings by Incorporating OS-aware Cache Way Lookup

Table 4 summarizes the I-cache leakage power savings as well as the run-time increases due to the OS-aware cache leakage control. One can see that both policies (i.e., residency-based and access-based) lead to a significant leakage power reduction. The residency-based drowsy mode scheme is more conservative, resulting in 5% - 50% of leakage power saving on the experimented applications. Access-based drowsy mode scheme, on the other hand, yields greater leakage power reduction by putting larger fraction of cache regions in to drowsy state, resulting in an average of 37% of overall leakage power reduction.

Table 4 also shows that both OS-aware cache set drowsy policies incur negligible (<1% in most case) run-time increase. This is because: (1) the cost of wrongly-putting a cache set into drowsy mode that is accessed thereafter is relatively small, and (2) using the proposed cache set drowsy policies makes the possibilities of accessing drowsy cache sets become extremely low. Therefore, the proposed leakage control techniques again preserve merits especially valuable for designing the power efficient, high performance server processor I-cache targeting on modern and commercial applications that heavily invoke OS activities.

Table 4. Normalized Leakage Power and Run-time Increase by Using the OS-aware Cache Set Drowsy Mode

	Residency-based						Access-based					
	Normalized Leakage Power			Increased Execution Cycle			Normalized Leakage Power			Increased Execution Cycle		
	User	**OS**	**Over-all**	**User**	**OS**	**Over-all**	**User**	**OS**	**Over-all**	**User**	**OS**	**Over-all**
pmake	0.96	0.34	0.90	0.03%	0.19%	0.04%	0.89	0.21	0.84	0.15%	1.15%	0.23%
gcc	1.00	0.20	0.95	0.02%	0.32%	0.04%	0.98	0.12	0.93	0.09%	1.22%	0.15%
vortex	0.99	0.38	0.94	0.03%	0.11%	0.04%	0.97	0.14	0.90	0.08%	0.84%	0.14%
sendmail	0.67	0.98	0.82	0.21%	0.05%	0.14%	0.41	0.70	0.55	0.71%	1.23%	0.95%
fileman	0.35	1.00	0.93	0.45%	0.04%	0.09%	0.24	0.89	0.81	4.95%	0.47%	0.98%
db	0.93	0.85	0.91	0.12%	0.23%	0.16%	0.72	0.40	0.61	1.06%	2.48%	1.54%
jess	0.97	0.62	0.86	0.09%	0.12%	0.10%	0.81	0.30	0.65	0.97%	2.05%	1.31%
javac	0.98	0.69	0.93	0.07%	0.19%	0.09%	0.87	0.25	0.76	0.61%	1.97%	0.85%
jack	0.99	0.74	0.95	0.03%	0.36%	0.08%	0.90	0.21	0.79	0.45%	2.08%	0.72%
mtrt	0.97	0.64	0.95	0.02%	0.24%	0.03%	0.92	0.20	0.88	0.11%	1.42%	0.19%
compress	0.47	0.99	0.50	0.05%	0.09%	0.05%	0.45	0.70	0.46	0.42%	4.09%	0.61%
postgres.select	0.96	0.79	0.92	0.07%	0.35%	0.14%	0.85	0.26	0.70	0.24%	0.70%	0.36%
postgres.update	0.97	0.53	0.74	0.49%	0.33%	0.41%	0.90	0.20	0.53	0.99%	0.65%	0.81%
postgres.join	0.99	0.56	0.95	0.05%	0.13%	0.06%	0.97	0.13	0.89	0.12%	0.76%	0.18%
osboot	0.52	0.99	0.95	0.98%	0.03%	0.11%	0.30	0.82	0.78	2.46%	0.34%	0.52%
AVERAGE	0.85	0.69	0.80	0.18%	0.19%	0.18%	0.75	0.37	0.63	0.89%	1.43%	1.05%

6. Related Work

Selective cache ways [6] reduce cache access energy by turning off unneeded ways in a set-associative cache. Recently, Zhang [4] proposed a reconfigurable cache architecture using way concatenation to adapt cache associativity for embedded applications. To use these techniques, the designers have to determine the appropriate configurations for a given program by exhaustively searching all possible configurations. The caches are reconfigured for the entire program execution. In contrast, OS-aware cache way lookups decide at run-time whether cache line lookups are necessary for a given operation mode.

Phased-lookup cache [28] uses a two-phase lookup, where all tag arrays are accessed in the first phase, but then only the one hit data way is accessed in the second phase. The employing of phased-lookup cache results in less data-way access energy at the expense of longer access time. Way prediction [10, 12] speculatively selects a way to access initially, and only access the other arrays if that initial array did not result in a match. To support way prediction, processor branch prediction mechanism has to be extended. Adding way-prediction to the branch prediction mechanism may affect the processor cycle time because the branch prediction access is often on one of the critical path. Way prediction scheme incurs a performance penalty by spending an extra cycle to access the other ways when a prediction fails. Moreover, way predicting of all I-cache accesses is non-trivial. In [12], Powell reported that even an elegant way predictor could make no prediction for a sizable fraction of I-cache accesses. We expect that the fraction of no prediction and misprediction on way prediction could even increase on system workloads due to their larger I-cache footprints and the effects of exception-driven, non-deterministic OS execution [3]. Compared with way prediction, the proposed OS-aware cache way lookups do not cause performance degradation and is easier to implement because no predictor is involved. Moreover, way prediction still needs full tag comparisons to verify the correctness of a prediction while OS-aware cache way lookups only probe selected cache tags. In [14], Lee et al. proposed region-based caching by re-organizing the first level cache to more efficiently exploit memory region (stack, global, heap) reference characteristics produced by programming language semantics. In [30], Kim et al. investigated ways of splitting the cache into several smaller units, each of which is a cache by itself (called a sub-cache). However, implementing region-based caching or sub-caching scheme requires substantial amount of modifications to be made in cache and other structures (e.g. TLB).

Approaches for reducing static power consumption of caches by turning off cache lines using the gated-Vdd technique have been described in [16, 27]. The drawback of this approach is that the state of the cache line is lost when it is turned off and reloading it from the L2 cache has a significant impact on performance. Because system performance largely depends on the performance of OS, the inherently high penalty gated-Vdd technique become less to control I-cache leakage for the OS. In [29], the using of compiler to insert power mode instructions to control cache leakage power was proposed. However, this approach requires the re-compilation of program source code, which is not generally applicable to the OS as well as many commercial applications. To reduce leakage energy dissipation, Yang [35] proposed a dynamically resizing I-cache. Compared with resizable cache, the proposed OS-aware cache tuning reduces power while still utilizing the full cache capacity. The drowsy instruction cache [32] uses dynamic voltage scaling and cache sub-bank prediction to achieve leakage power reduction. Like way prediction, a misprediction on cache sub-bank incurs a performance penalty. When applied to large, set-associative cache, an aggressive cache sub-bank predictor yields mediocre prediction accuracies [32]. The area as well as power overhead of the memory sub-bank prediction buffers, which yield better prediction accuracies, can be significant.

7. Conclusions

Many modern applications result in a significant operating system (OS) component. This trend is likely to continue in the near future and it is very important to consider the OS not only for performance evaluations, but also when attempting to optimize the performance and power of hardware. Adhering to this philosophy, this paper explores the opportunities of employing the three subsystems – application, OS and hardware – to improve I-cache energy efficiency. We start from characterizing user/OS I-cache accesses on system workloads to identify power saving opportunities due to dual-mode operation. We then propose two simple OS-aware techniques incorporating processor operation mode to improve I-cache energy efficiency on system workloads. The proposed OS-aware cache way lookup reduces the number of parallel tag comparisons and data array read-outs for cache accesses and saves dynamic power. Integrating with a state-preserving, leakage control mechanism, OS-aware tuning effectively reduces static power, which is gaining in importance due to CMOS technology scaling. Unlike other proposed schemes, OS-aware tuning achieves both dynamic and static power savings but requires minimal hardware modification and addition. To our knowledge, this work is the first step to explore cache power optimization on system workloads including the OS. The proposed techniques can be implanted into server processor I-caches mostly targeting on OS-intensive commercial applications.

References

[1] J. Kin, M. Gupta and W. H. Mangione-Smith, The Filter Cache: An Energy Efficient Memory Structure, In Proceedings of the International Symposium on Microarchitecture, 1997.

[2] M. B. Kamble and K. Ghose, Energy-Efficiency of VLSI Caches: A Comparative Study, In Proceedings of the IEEE 10th International Conference on VLSI Design, 1997.

[3] Y. Luo, P. Seshadri, J. Rubio, L. K. John and A. Mericas, A Case Study of 3 Internet Server Benchmarks on 3 Superscalar Machines, IEEE Computer, Feb. 2003.

[4] C. Zhang, F. Vahid and W. Najjar, A Highly Configurable Cache Architecture for Embedded Systems, In Proceedings of the International Symposium on Computer Architecture, 2003.

[5] P. Ranganathan, S. Adve and N.P. Jouppi, Reconfigurable Caches and their Application to Media Processing, In Proceedings of the International Symposium on Computer Architecture, 2000.

[6] D. H. Albonesi, Selective Cache Ways: On-Demand Cache Resource Allocation, Journal of Instruction Level Parallelism, May 2000.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures, In Proceedings of the International Symposium on Microarchitecture, 2000.

[8] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and 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.

[9] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas, L1 Data Cache Decomposition for Energy Efficiency, In Proceedings of the International Symposium on Low Power Electronics and Design, 2001.

[10] K. Inoue, T. Ishihara, and K. Murakami, Way-Predictive Set-Associative Cache for High Performance and Low Energy Consumption, In Proceedings of the International Symposium on Low Power Electronics and Design, 1999.

[11] M. Powell, S. H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories, In Proceedings of the International Symposium on Low Power Electronics and Design, 2000.

[12] M. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, In Proceedings of the International Symposium on Microarchitecture, 2001.

[13] J. Yang and R. Gupta, Energy Efficient Frequent Value Data Cache Design, In Proceedings of the International Symposium on Microarchitecture, 2002.

[14] H.-H. S. Lee, G. S. Tyson, Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors, In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2000.

[15] K. Flautner, N. S. Kim, S. Martin, D. Blaauw and T. Mudge, Drowsy Caches: Simple Techniques for Reducing Leakage Power, In Proceedings of the International Symposium on Computer Architecture, 2002.

[16] S. Kaxiras, Z. G. Hu and M. Martonosi, Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power, In Proceedings of the International Symposium on Computer Architecture, 2001.

[17] N. Gloy, C. Young, J. B. Chen and M. D. Smith, An Analysis of Dynamic Branch Prediction Schemes on System Workloads, In Proceedings of the International Symposium on Computer Architecture, 1996.

[18] T. Li and L. K. John, Run-time Modeling and Estimation of Operating System Power Consumption, International Conference on Measurement and Modeling of Computer Systems, 2003.

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

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

[21] 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.

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

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

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

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

[26] A. Agarwal, H. Li, and K. Roy, DRG-Cache: A Data Retention Gated-Ground Cache for Low Power, In Proceedings of the International Design Automation Conference, 2002.

[27] H. Y. Zhou, M. C. Toburen, E. Rotenberg and T. M. Conte, Adaptive Mode Control: a Static Power-Efficient Cache Design, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2001.

[28] A. Hasegawa, I.Kawasaki, K.Yamada, S.Yoshioka, S. Kawasaki, and P. Biswas, SH3: High Code Density, Low Power, IEEE Micro, Dec. 1995.

[29] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, Compiler-Directed Instruction Cache Leakage Optimization, In Proceedings of the International symposium on Microarchitecture, 2002.

[30] S. Kim, N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam and M. J. Irwin, Partitioned Instruction Cache Architecture for Energy Efficiency, ACM Transactions on Embedded Computing Systems, Vol. 2, Issue 2, May 2003.

[31] SIA. International Technology Roadmap for Semiconductors, 2001.

[32] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, Drowsy Instruction Caches, In Proceedings of the International Symposium on Microarchitecture, 2002.

[33] J. A. Redstone, S. J. Eggers and H. M. Levy, An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture, In Proceedings of the International Conference on Architectural Support for Program Languages and Operating Systems, 2000.

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

[35] S. H. Yang, M. Powell, B. Falsafi and T. N. Vijay, Exploiting Choice in Resizable Cache Design to Optimize Deep-submicron Processor Energy-delay, In Proceedings of the International Symposium on High-Performance Computer Architecture, 2002.