Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power

Nam Sung Kim, *Member, IEEE*, Krisztian Flautner, *Member, IEEE*, David Blaauw, *Member, IEEE*, and Trevor Mudge, *Fellow, IEEE*

***Abstract*—On-chip caches represent a sizable fraction of the total power consumption of microprocessors. As feature sizes shrink, the dominant component of this power consumption will be leakage. However, during a fixed period of time, the activity in a data cache is only centered on a small subset of the lines. This behavior can be exploited to cut the leakage power of large data caches by putting the cold cache lines into a state preserving, low-power drowsey mode. In this paper, we investigate policies and circuit techniques for implementing drowsy data caches. We show that with simple microarchitectural techniques, about 80%–90% of the data cache lines can be maintained in a drowsy state without affecting performance by more than 0.6%, even though moving lines into and out of a drowsy state incurs a slight performance loss. According to our projections, in a 70-nm complementary metal–oxide–semiconductor process, drowsy data caches will be able to reduce the total leakage energy consumed in the caches by 60%–75%. In addition, we extend the drowsy cache concept to reduce leakage power of instruction caches without significant impact on execution time. Our results show that data and instruction caches require different control strategies for efficient execution. In order to enable drowsy instruction caches, we propose a technique called cache subbank prediction, which is used to selectively wake up only the necessary parts of the instruction cache, while allowing most of the cache to stay in a low-leakage drowsy mode. This prediction technique reduces the negative performance impact by 78% compared with the no-prediction policy. Our technique works well even with small predictor sizes and enables a 75% reduction of leakage energy in a 32-kB instruction cache.**

***Index Terms*—Dynamic voltage scaling, L1 caches, low power, subthreshold leakage power.**

I. Introduction

HISTORICALLY, one of the advantages of complementary metal–oxide–semiconductor (CMOS) over competing technologies, such as transistor-transistor logic (TTL) and emitter coupled logic (ECL), has been its lower power dissipation. When not switching, CMOS transistors have, in the past, dissipated negligible amounts of power. However, as the feature size of these devices has decreased, so has their subthreshold leakage (static) power dissipation. As processor technology moves below 0.1 μm, static power dissipation is set to dominate the total power used by digital circuits. Furthermore, subthreshold leakage presents an interesting tradeoff. On one hand, performance demands require the use of fast high-leakage transistors; on the other hand, new applications and cost issues favor designs that are energy efficient. Fig. 1 illustrates the magnitude of the problem with data from existing technologies and projections based on the *international technology roadmap for semiconductor* (ITRS) [1]. As it can be seen, even in current-generation technology, subthreshold leakage power dissipation is comparable to the dynamic power dissipation, and the fraction of the leakage power will increase significantly in the near future. In fact, the off-state subthreshold leakage component of the total power in a microprocessor may exceed active power as the technology decreases below the 65 nm technology node, according to a projection from Intel [2].

Subthreshold leakage is a problem for all transistors, but it is a particularly important problem in on-chip caches, because they are a growing fraction of the total number of microprocessor devices. For instance, 30% of Alpha 21 264 and 60% of StrongARM are devoted to cache and memory structure [3]. Furthermore, the leakage power is becoming the dominant fraction of total cache power consumption because of a very high number of storage cells (cross-coupled inverters) of on-chip caches where there is no stacking effect [4] reducing the leakage current. We project that in a 70-nm CMOS process, leakage will amount to more than 60% of power consumed in L1 caches if left unchecked. However, most data in caches is accessed relatively infrequently due to either temporal or spatial locality, thus, as the cost of storing data increases in the form of leakage power, the contribution of dynamic power dissipation diminishes. To alleviate this problem, transistors in caches could be statically designed such that they have less leakage, for example, by assigning them a higher threshold voltage [5]. However, computer architects would like to have the best of all worlds: large cache; fast access time; and low power consumption. We believe that it is possible to reconcile these aims by taking advantage of the runtime characteristics of workloads and by attacking the problem at both the circuit and microarchitecture levels. In particular, significant leakage reduction can also be achieved by putting infrequently accessed cache lines into a low-power *standby or drowsy* mode.

In this paper, we propose a simple but effective circuit technique for implementing caches that have a drowsy mode, where one can choose between two different supply voltages for each cache line. Such a dynamic voltage scaling (DVS) technique has been used in the past to trade off dynamic power consumption and performance [6], [7]. In this case, however, we exploit the voltage scaling technique to reduce leakage power dissipation. Due to short-channel effects in deep-submicron CMOS

Manuscript received February 28, 2003.

N. S. Kim, D. Blaauw, and T. Mudge are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: kimns@eecs.umich.edu).

K. Flautner is with ARM, Ltd., Cambridge CB1 9NJ, U.K.

Digital Object Identifier 10.1109/TVLSI.2003.821550

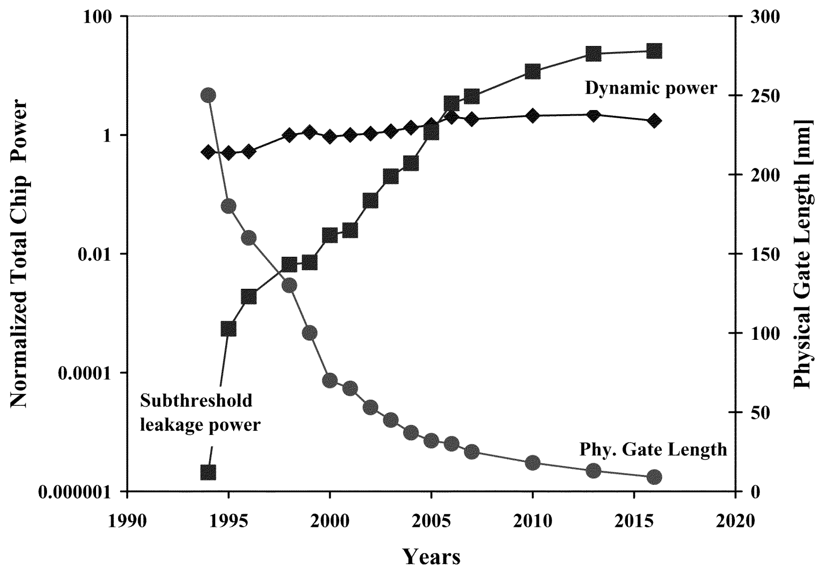


Fig. 1. Normalized dynamic and static power dissipation for (W/Lgate =3 device. Data is based on the ITRS [1] and normalized to the year 2001's figures.

processes, leakage current reduces substantially with voltage scaling [8]. The combined effect of reduced leakage current and voltage yields a drastic reduction in leakage power. In a regular cache, all lines leak at a high rate, but in the drowsy cache, the high leakage component is only incurred when the lines are *awake* or *active*. The key is to arrange things so that these active lines are the ones that are accessed. Although leakage is not zero in the drowsy mode, it provides more than a 10 $\times$ reduction (depending on design) over the regular high-leakage mode (see Section III-A for an analysis). While voltage scaling does not reduce leakage as much as the *gated-*V_{DD} technique appearing in [9] and [10], it has the crucial advantage of being able to preserve the state of the transistors.

The rest of the paper is organized as follows. In Section II, previously proposed circuit and microarchitectural techniques to reduce the leakage power of *static random-access memory* (SRAM) will be reviewed. In Section III, we will present a new 6T-SRAM circuit that uses DVS to reduce the subthreshold leakage power while preserving memory states. With the proposed circuit technique, we propose simple but effective microarchitectural control mechanisms that reduce the leakage power for L1 data and instruction caches. First, we focus on the policy implications of using drowsy data caches, evaluating the design tradeoffs between simple policies in Section IV. We argue that the simplest policy of periodically putting the entire data cache lines into drowsy mode does about as well as a policy that tracks access to cache lines. Second, in Section V, we concentrate on a *drowsy instruction cache* architecture and its performance implications of waking up and precharging a specific subbank on demand. We note that the on-demand wake up and precharge of the currently accessed subbank may degrade the performance significantly, due to the intermittent transitions among the subbanks. To reduce the performance impact of the subbank transitions, we propose two subbank prediction techniques and examine their performance implications. In Section VI, we present simulation models and experiment results for the proposed cache architectures. Section VII concludes and suggests directions for future research.

II. Background Work

A. Circuit and Device Techniques

In [11], a multithreshold CMOS (MTCMOS) circuit technique was proposed to satisfy both the requirement of lowering the threshold voltage of metal–oxide–semiconductor field-effect transistors (MOSFETs) and reducing standby or subthreshold leakage current. To increase operating speed, *low-*V_{TH} MOSFETs were used for logic gates, and during the long standby time (i.e., sleep time), the power supply was disconnected with *high-*V_{TH} MOSFETs. This concept was also applied in a prototype design of MTCMOS memories to reduce the power dissipation of peripheral circuits such as row decoders and input/output (I/O) circuitry [12], [13]. However, the MTCMOS technique cannot be applied to the memory cell array, because the sleep control transistors in the MTCMOS technique cut the power supply off, destroying the states of memory cells. Instead, to suppress the leakage power dissipation of the memory cell array, *high-*V_{TH} MOSFETs are used, which increases the overall memory access time. In [14], *adaptive reverse body biasing* (ABB) MTCMOS was proposed to control the leakage current of SRAM cells during standby mode. In general, the subthreshold leakage current decreases exponentially as V_{TH} increases, and the adaptive body biasing technique has the advantage that it reduces the leakage current exponentially by increasing V_{TH}. Although the SRAM designed with the ABB MTCMOS technique reduces a substantial amount of the leakage power, the transition between the active and sleep modes is very slow because the wake-up transistors have to drive large substrate capacitances.

Several dual-threshold voltage CMOS techniques for leakage power reduction have been investigated [15], [16]. The basic concept of dual-threshold voltage CMOS techniques is to use *low*-V_{TH} for circuits in the critical path, and to employ *high*-V_{TH} transistors for the rest of the circuits to suppress unnecessary leakage current. In SRAM design, *low*-V_{TH} transistors have been used in the peripheral circuits of caches with *high*-V_{TH} transistors for the memory cells. In [5], different cache designs with *dual*-V_{TH} were investigated in a 130-nm technology. Among other design choices, they showed that a SRAM cell using all *high*-V_{TH} transistors increased the access time of memory by up to 26% compared with that of a memory design with *low*-V_{TH} devices [5].

The *gated*-V_{DD} structure was introduced in [9] and [10]. This technique reduces the leakage power by using a *high*-V_{TH} transistor between VV_{SS} (a virtual ground) and V_{SS} (the ground) to cut off the power supply of the memory cell when the cell is set to low-power mode. This *high*-V_{TH} gating transistor dramatically reduces the leakage of the circuit, due to the stacking effect [4] and the exponential dependence of leakage on the V_{TH} of the gating transistor. While this method is very effective at suppressing leakage, its main disadvantage lies in that it loses any information stored in the cell when switched into low-leakage mode. This means that a significant performance penalty may be incurred when data in the cell is accessed and more complex and conservative cache policies must be employed. Also, the stacked transistor that reduces the leakage current is in the critical path, which results in increased access time of the memory. A *single*-V_{TH} *data-retention gated-ground* (DRG) SRAM was proposed in [17]. This technique relies solely on the forced-stacking effect to reduce the leakage current. To retain the cell state, this technique requires sophisticated transistor sizing and is sensitive to noise during sleep time. For a processor with $V_{\mathrm{TH}} = 250$ mV, this technique reduces leakage power by 40% while increasing read time by 4.4%, compared with the conventional SRAM. In comparison, the *gated*-V_{DD} technique without data-retention capability reduces the leakage power by 97%, while increasing read time by 8% and area by 5% using 0.2 V and 0.4 V for *low*- and *high*-V_{TH}'s, respectively.

Recently, several other SRAM cell-design techniques to reduce the leakage power were proposed [18]–[20]. All these circuit techniques rely on the microarchitectural dynamic characteristics of the workloads such as the bits being highly biased to state "0" [18], [19] or the bits in the branch predictor or *branch target buffer* (BTB) being transient and predictive [20]. In [18], *high*-V_{TH} transistors were asymmetrically used in an SRAM cell for the selected biased state. To compensate for the slow access time in the biased state, a special sense-amplifier circuit was also proposed, which requires two more transistors per sense-amplifier. To suppress the leakage current from bit lines through the access transistors of SRAM cells, a leakage-biased bit-line architecture was proposed with dual-V_{TH} storage cells in [19]. In this technique, the bit lines in inactive subbanks are floated, which makes the leakage currents from the bit cells automatically bias the bit line to a mid-rail voltage that minimizes the bit line leakage current through the access transistors. The leakage reduction of this technique depends on the percentage of zero or one resident bits in caches. Finally, in [20], a quasi-static 4T SRAM cell was proposed for the transient and predictive bits in the memory structure for the branch prediction. The 4T cells are about as fast as 6T cells, but they do not store charge indefinitely due to leakage. This technique can be applied only to structures used for speculative operations which do not affect the correctness of the execution of programs.

B. Microarchitectural and Compiler Techniques

Most microarchitectural and compiler leakage suppression techniques are combined with the circuit techniques mentioned in Section II-A. In [9] and [10], dynamically resizable cache architectures were proposed. The key observation behind these techniques is that there is a large variability in instruction cache utilization, both within and across programs leading to large energy inefficiency in conventional caches. To take advantage of this observation, these approaches resize caches to increase or decrease the number of unused sets in the caches by turning on or off *high*-V_{TH} nMOS gating transistor between VV_{SS} and V_{SS} using the circuit technique proposed in [9]. The authors assumed that the states of the cache cells are lost when the gating transistor is turned off. The technique requires extra hardware for estimating cache miss rates and resizing factors. The dynamically resizing cache also requires its tag bits to be compatible with both small and large caches. Furthermore, resizing caches may require data remapping on the fly, and thus, incur compulsory misses whenever resizing occurs. It was reported that resizing the cache on the fly can lead to large performance penalties in some applications [17]. The approaches proposed in [21] exploit generational behavior of caches to reduce the leakage power. It turns off a "dead" cache line if a preset number of cycles have elapsed since its last access. To turn off cache lines selectively, it also uses the same circuit technique proposed in [9]. This technique incurs a cache-miss penalty like the techniques proposed in [9] and [10] when the turned off cache lines are required to be accessed. Adaptive techniques to determine the dead cache lines were also proposed to reduce the cache-miss penalty.

In [22], a compiler-based leakage optimization strategy for instruction caches was introduced. This approach is based on determining the last use of an instruction at the granularity of a loop. Once the last use of the instructions is detected at the loop exit, the corresponding cache lines in the loop are either turned off or switched to the state-preserving sleep state, assuming that the loop will not be revisited in the near future. To turn off or switch specific cache lines to sleep mode using the circuit proposed in [23] and [24], a cache-line mode-control instruction was also proposed.

III. DVS for SRAM Leakage Power Reduction

The method proposed in this paper utilizes DVS to reduce the leakage power of SRAM cells. In active mode, the standard supply voltage is provided to the SRAM cells. However, when cells are not intended to be accessed for a time period, they are placed in a sleep or "drowsy" mode by supplying a standby voltage in the range of 200–300 mV to the SRAM cells. In drowsy mode, the leakage power is significantly reduced due to the decreases in both leakage current and supply voltage. Supply voltage reduction is especially effective for leakage power reduction due to short-channel effects, such as drain-induced barrier lowering (DIBL), which results in a superlinear dependence of leakage current on the supply voltage [8]. To understand the

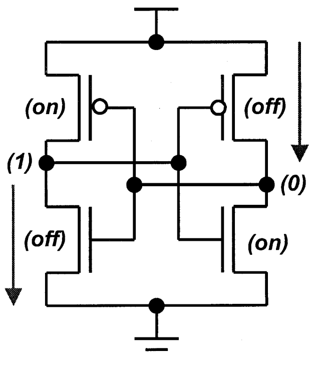


Fig. 2. Leakage inside a SRAM cell.

leakage power reduction achievable in the drowsy mode, we present in this section a subthreshold leakage current analysis of SRAM cells, and a leakage power reduction technique using DVS. Also, we discuss other issues such as wake-up latency and crosstalk noise stability of the proposed technique.

A. SRAM Leakage Power Reduction Using DVS

Basically, there are two off-state leakage current paths through the two cross-coupled inverters in the standard SRAM cell, as shown in Fig. 2. The off-state leakage current is dominated by the weak inversion current in the next-generation 70-nm technology, which can be modeled as [25], [26]

$$I_D = I_{s0} e^{V_{GS} - V_{\mathrm{TH}}/(nkT/q)} \left(1 - e^{-V_{DS}/(nkT/q)}\right)(1 + \lambda V_{DS}) \quad (1)$$

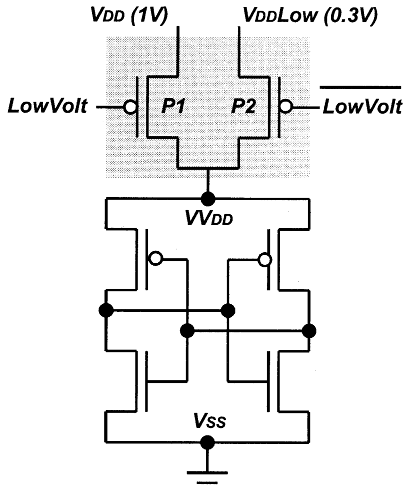
where λ is a parameter modeling the pseudosaturation region in weak inversion. Since the on transistors are in a strong inversion region and only present a small serial resistance that can be ignored, the overall leakage current is the sum of leakage currents from the off transistors in the cross-coupled inverters. The overall leakage of the SRAM cell is, therefore, modeled as follows:

$$I_L = ((I_{SN} + I_{SP}) + (I_{SN}\lambda_N + I_{SP}\lambda_P)V_{DD}) \times \left(1 - e^{-V_{DD}/(nkT/q)}\right) \quad (2)$$

where I_{SN} and I_{SP} are nMOS and pMOS off-transistor current factors that are independent of V_{DS} in (1).

From (2), it is clear that the leakage current reduces superlinearly with V_{DD}, and hence, significant reduction in leakage power can be obtained in drowsy mode. In drowsy mode, a minimum voltage must be applied to maintain state. In our implementation, assuming that the threshold voltage of 70-nm technology is ~200 mV, we set the state-preserving or data-retention voltage 50% higher than the threshold voltage. It was found that, despite process variations, this state-preserving voltage is quite conservative, and that the state-preserving supply voltage can be even reduced further if necessary [27].

In Fig. 3(a) we illustrate a proposed drowsy SRAM cell with the supply voltage control mechanism. The two pMOS transistors, P1 and P2, control the supply voltage of the memory cell based on the operating mode: active or standby mode. When the cell is in the active mode, P1 supplies a standard supply voltage (1 V), and P2 supplies a standby voltage (~300 mV) when it is



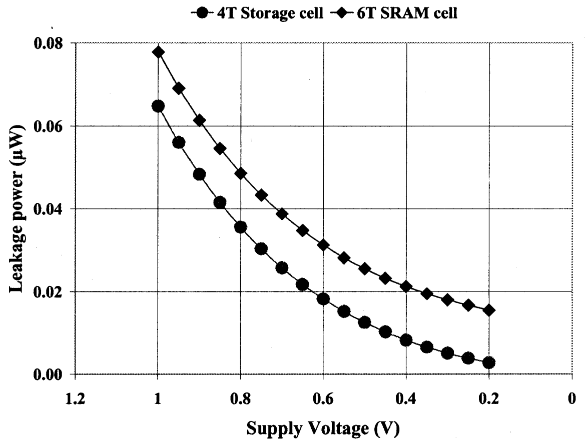


Fig. 3. A drowsy SRAM cell with (a) the supply voltage control mechanism and (b) the subthreshold leakage power reduction of the SRAM cell with DVS.

in the drowsy mode. P1 and P2 are controlled by complementary supply voltage control signals. In drowsy mode, however, SRAM cell accesses are not allowed, because the bit line voltage is higher than the storage cell core voltage, which may result in destroying the cell state. Moreover, the sense-amplifier may not operate properly at the low storage cell core voltage, because the memory cell does not have enough driving capability to sink the charge stored in the bit lines. When we build a cache system for a microprocessor with a drowsy SRAM circuit technique, we can apply it to either each cache-line or on a subbank basis, depending on the microarchitectural control mechanism; each cache line or subbank shares the same virtual supply voltage node (VV_{DD}).

In Fig. 3(b), we show the subthreshold leakage power reduction trend of 4T storage and 6T SRAM cells with the 70-nm technology as we scale the supply voltage down. According to our HSPICE measurements, 4T and 6T SRAM cells dissipate about 65 nW and 78 nW, respectively, at 1 V standard supply voltage. We can reduce the leakage power of the 4T and 6T SRAM cells by 92% and 77% at 300 mV standby supply voltage. We initially ignore the bit line leakage through the access transistors, although in a 6T SRAM cell, the leakage power dissipation through the access transistors is responsible for approximately 20% of the total leakage power of the active mode. However, in Sections IV-A and V-A, we will address how this leakage can be controlled by either assigning a high threshold

TABLE I
SUPPLY VOLTAGE RAIL INTERCONNECT PARAMETERS

Wire dimension					Dielectric
Width (μm)	Space (μm)	Length (μm)	Thickness (μm)	Height (μm)	K
0.14	1.42	183.18	0.35	0.20	3.9
Capacitance			Resistance		
Cground (fP)	Ccouple (fP)	Ctotal (fP)	R (Ω /mm)		Rtotal (Ω)
21.30	2.30	25.89	408.16		74.77

The estimated height and width of the SRAM cell is 1.42 μm by 0.72 μm, respectively, in 70-nm technology.

TABLE II
WAKE-UP LATENCIES AND ENERGY OF EACH SIZE OF THE VOLTAGE CONTROLLERS

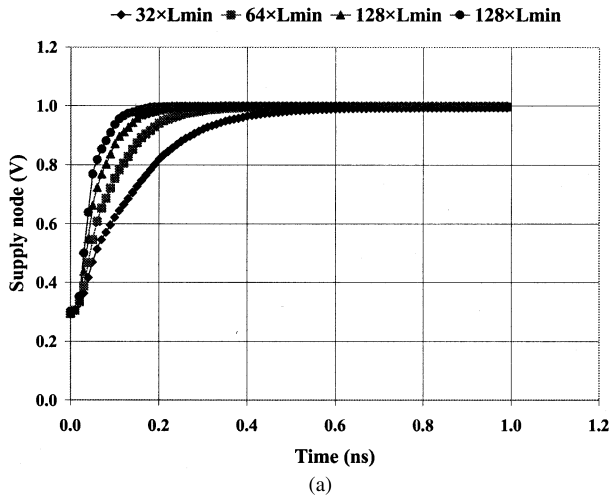
	16xLmin	32xLmin	64xLmin	128xLmin	256xLmin
Latency (ps)	919 (2.33)	473 (1.20)	274 (0.69)	179 (0.45)	132 (0.33)
Energy (pJ)	108.48	109.33	115.72	130.34	163.90

$L_{\min}$ is equivalent to 2 λ and the number in the parenthesis in normalized to 12 $\times$ FO4 delay.

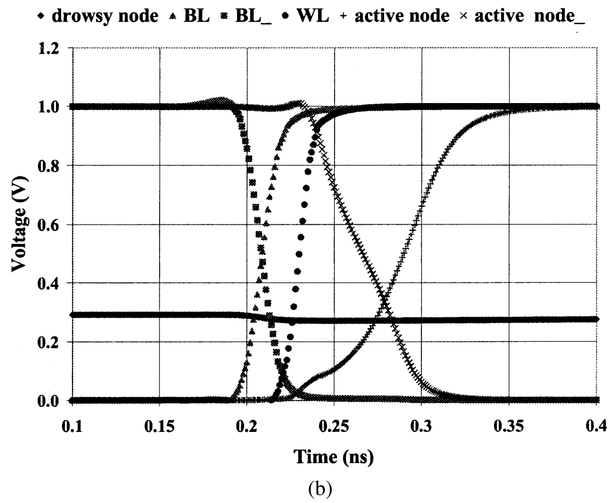
voltage to the access transistors or gating the precharge signals of memory subbanks.

B. Wake-Up Latency and Crosstalk Noise Stability

When the SRAM cells are in standby or drowsy mode, it takes a finite amount of time to restore the voltage level of the VV_{DD} node from standby to standard supply voltage level, which we refer to as "wake-up" latency. To estimate the wake-up latency, we connected 128 SRAM cells to a voltage controller. We also model the interconnect capacitance and resistance according to the estimated supply voltage line length and width based on 70-nm technology [28]. To estimate the 70-nm technology SRAM cell dimension, we applied a linear scaling to the Artisan 180-nm technology memory cell. Table I shows the detailed power supply wire parameters for the HSPICE experiments, and the wire capacitance and resistance are obtained from [28] with the given wire dimensions. To estimate the number of cycles for restoring the supply voltage level of VV_{DD} node, we need to estimate the clock frequency of a typical microprocessor. According to [29], the clock frequency has been around 16 $\times$ FO4 (fan-out of four gate delay). This corresponds to 527ps in the 70-nm technology, and the frequency will approach 12 $\times$ FO4 in future technology. Table II shows the wake-up latency and energy estimated for each size of the voltage controller transistors (see Fig. 4(a) for the HSPICE wake-up latency simulation waveforms) as the width of the P1 transistor is increased. We defined the rise time of VVDD from 0.25 to 0.99 V as a wake-up latency. The latency number in the parentheses is normalized to 12 $\times$ FO4. The energy number in Table II includes the dissipation by the circuitry driving the voltage controller transistor. The normalized numbers of cycles in Table II are a conservative estimation. When we use 32 $\times$ and 64$\times L_{\min}$-size voltage controllers, we can restore the full supply voltage level of VV_{DD} node in two or one cycles, respectively. However, the voltage controller for supplying standby voltage does not need to use such a wide pMOS transistor (e.g., 64 $\times L_{\min}$ size), because the latency from active to drowsy mode is not critical for the processor performance. Thus, we can use a minimum-size pMOS transistor [P2 in Fig. 3(a)] that provides enough current for sustaining the standby voltage of cache lines.



(a)



(b)

Fig. 4. Wake-up delay for (a) the VV_{DD} node and (b) the crosstalk noise stability of a drowsy cell.

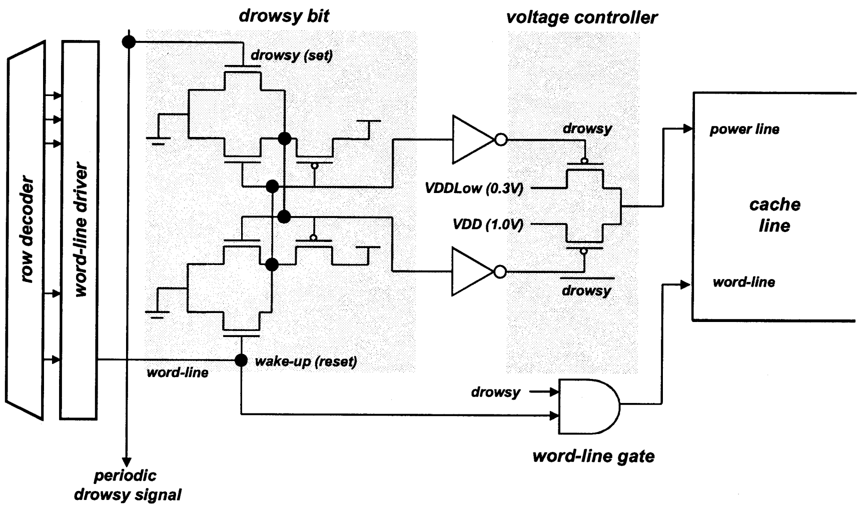


Fig. 5. An implementation of a cache line for drowsy data caches. Note that, for simplicity, the word line, bit lines, and two pass transistors in the drowsy bit are not shown in this figure.

To estimate the area overhead of the voltage controllers, we drew the actual layout using TSMC 0.18 μm technology, which was the smallest feature size available to the academic community at the time of writing. The dimensions of the memory cell from the memory compiler were 3.66 μm $(40\,\lambda)$ $\times$ 1.84 $(20\,\lambda)$ μm, and those for the voltage controllers were 3.66 μm $(40\,\lambda) \times$ 1.98 μm $(22\,\lambda)$ and 3.66 μm $(40\,\lambda) \times$ 3.42 μm $(38\,\lambda)$ for 32 $\times$ and 64 $\times L_{\min}$ size voltage controllers, respectively. We estimate the extra area by the voltage controller per 128-bit cache line is equivalent to 1.1 (0.87%) and 1.9 (1.48%) extra memory cells for the 32 $\times$ and the 64 $\times L_{\min}$ voltage controller, respectively. This relatively low area overhead can be achieved because of the negligible amount of the interconnect in the voltage controller compared with that of SRAM cell; a significant amount of a 6T SRAM cell area is consumed by local interconnects although the effective size of each transistor is small.

As we decrease the supply voltage further, we can suppress more leakage power. However, the stability of drowsy cells in the presence of crosstalk noise can be a problem, because scaling supply voltage down reduces the charge stored in the drowsy nodes. Assuming that row n is in drowsy mode and an adjacent row $n+1$ is in awake mode, we examined the crosstalk noise stability of the drowsy cells in row n by applying a write operation to the awake row $n+1$. This makes all the bit lines connected to both drowsy and active cells swing rail-to-rail. According to the HSPICE simulation shown in Fig. 4(b), there is only a slight voltage fluctuation at the drowsy node during the write operation to the awake row, while the voltage level of the drowsy cells recovers its standby voltage level quickly, because the cross-coupled inverters in the drowsy cell continue driving its internal nodes.

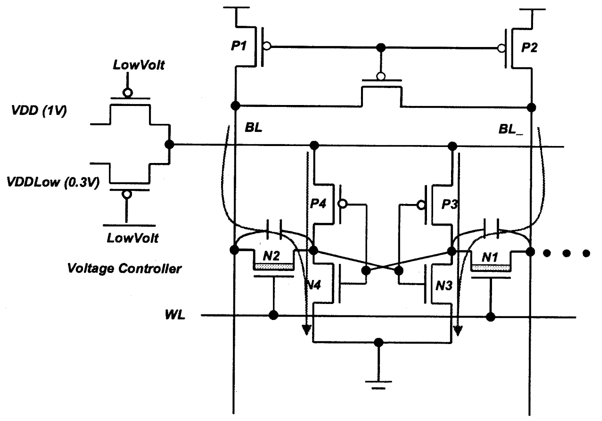
IV. Drowsy Data Caches

A. Cache-Line Architecture Using DVS

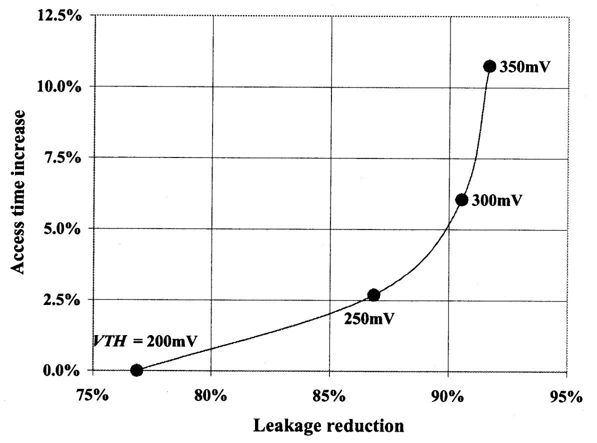
Fig. 5 shows the changes necessary for implementing a cache line that supports a drowsy mode. There are very few additions required to a standard cache line. The main ones are a drowsy bit, a mechanism for controlling the voltage to the memory cells, and a word-line gating circuit. In order to support the drowsy mode, the cache-line circuit includes the voltage controller illustrated in Section III, which switches the cache-line voltage between the standard (active) and standby (drowsy) supply voltages depending on the state of the drowsy bit. If a drowsy cache line is accessed, the drowsy bit is cleared, and consequently the supply voltage is switched to *high* V_{DD}. The word-line gating circuit is used to prevent access of the cache line when it is in drowsy mode, because the supply voltage of the drowsy cache line is lower than the bit line precharge voltage; unchecked access to a drowsy line might destroy the memory states.

Whenever a cache line is accessed, the cache controller monitors the condition of the voltage of the cache line by reading the drowsy bit. If the accessed line is in standard supply-voltage mode, we can read out the contents of the cache line without loss of performance. No performance penalty is incurred, because the power mode of the line can be checked by reading the drowsy bit, concurrently with the read out and comparison of the tag. However, if the memory array is in drowsy mode, we need to prevent the discharge of the bit lines of the memory array because it may read out incorrect data. The line is woken up automatically during the next cycle, and the data can be read out during consecutive cycles.

Fig. 6(a) illustrates a SRAM cell architecture for drowsy data caches. There is an additional leakage current path through one of the access transistors depending on the cell state in the 6T SRAM cell. When we use *low*-V_{TH} (200 mV) access transistors, the total leakage power reduction is limited to 77% at 300 mV drowsy supply voltage level, due to the access transistor leakage current. To further reduce the leakage power of the 6T SRAM cell, *high*-V_{TH} transistors can be optionally used for the access transistors that connect the 4T cross-coupled inverters of storage cells to bit lines ($N1$ and $N2$) in the 6T SRAM cell. However, using *high*-V_{TH} access transistors will increase the access time of the memory or require wider $N3$ and $N4$ transistors to compensate for the increased access time. According to the HSPICE simulation shown in Fig. 6(b), we can reduce the leakage power of the 6T SRAM cell by $\sim 91\%$ by adjusting the



(a)



(b)

Fig. 6. A SRAM architecture for (a) a drowsy data cache line and (b) the tradeoff between the leakage power reduction and access time increase when *high-*$V_{\rm TH}$ access transistors are used. The arrows in (a) are possible leakage current paths in the SRAM cell, and a *high-*$V_{\rm TH}$ can be used for $N1$ and $N2$ transistors to reduce the leakage power dissipation through the access transistors (optional).

threshold voltage of the access transistors to 300 mV, which increases the access time of the memory by $\sim$ 6%. As explained further in Section V-A, it should be noted that if the drowsy cache approach is used on a subbank basis, the supply voltage of all bit lines can be floated in drowsy mode, similar to that of the bit cell supply voltage, thereby making it unnecessary to use *high-*$V_{\rm TH}$ access transistors.

B. Cache Management Policies

The key difference between drowsy caches and caches that use *gated-*V_{DD} is that in drowsy caches, the cost of being wrong—putting a line into drowsy mode that is accessed soon thereafter—is relatively small. The only penalty one must contend with is an additional delay and energy cost for having to wake up a drowsy line. One of the simplest policies that one might consider is that all lines in the cache, regardless of access patterns, are put into drowsy mode periodically and a line is woken up only when it is accessed again. This policy requires only a single global counter and no per-line statistics. Table III shows the working set characteristics of some of our workloads using a 2000-cycle update window, meaning that all cache lines are put into drowsy mode every 2000 cycles. Observations of cache activity are made over this same period. Based on this information, we can estimate how effective this simple policy could be.

The results show that on most of the benchmarks the working set (the fraction of unique cache lines accessed during an update window) is relatively small. On most benchmarks, more than 90% of the lines can be in drowsy mode at any one time. This has the potential to significantly reduce the static power consumption of the cache. The downside of the approach is that the wake-up cost has to be amortized over a relatively small number of accesses: between 7 (*crafty*) and 21 (*equake*), depending on the benchmark. See (3) at the bottom of the page.

Equation (3) shows the formula for computing the expected worst-case execution time increase for the baseline algorithm. All variables except *memory impact* and *wake-up latency* are directly from Table III. The term *memory impact* can be used to describe how much impact a single memory access has on overall performance. The simplifying assumption is that any increase in cache access latency translates directly into increased execution time, in which case, *memory impact* is set to 1. Using this formula and assuming a one-cycle wake-up latency, we get a maximum of 9% performance degradation for *crafty* and under 4% for *equake*. One can further refine the model by coming up with a more accurate value for memory impact. Its value is a function of both the microarchitecture and the workload.

- The workload determines the ratio of the number of memory accesses to instructions.
- The microarchitecture determines what fraction of wake-up transitions can be hidden, i.e., not translated into global performance degradation.
- The microarchitecture also has a significant bearing on IPC, which, in turn, determines the number of memory accesses per cycle.

Assuming that half of the wake-up latencies can be hidden by the microarchitecture, and based on a ratio of 0.63 of memory accesses per cycle, the prediction for worst-case performance impact for the *crafty* benchmark reduces to 2.8%. Similarly, using the figure of 0.76 memory accesses per cycle and the same fraction of hidden wake-up transitions, we get a performance impact of about 1.4%. The actual impact of the baseline technique is likely to be significantly lower than the results from the analytical model, but nonetheless, these results show that there is no need to look for prediction techniques to control the

$$\text{Execution factor} = \frac{\text{accesses}\left(\frac{\text{wake-up latency}\times\text{memory impact}}{\text{access per line}}\right) + (\text{window size} - \text{accesses})}{\text{window size}} \tag{3}$$

TABLE III
WORKING SET AND REUSE CHARACTERISTICS

	L1 data cache 32KB 4-way set associative cache with a 2048-cycle window						
	Working set	**Number of accesses**	**Accesses per line**	**Accesses per cycle**	**Fraction of accesses that are the same as in the *n*-th previous window**		
					***n*=1**	***n*=8**	***n*=32**
bzip	5.9%	1055.8	17.4	0.53	32.5%	19.7%	17.2%
crafty	17.6%	1250.6	7.0	0.63	65.2%	54.9%	49.3%
equake	7.0%	1513.3	21.1	0.76	92.8%	91.4%	90.7%
facerec	10.4%	970.0	9.2	0.49	37.4%	27.5%	33.6%
gcc	8.1%	809.7	9.8	0.40	36.9%	24.9%	21.1%
mcf	8.9%	1831.7	20.1	0.92	61.0%	60.8%	60.4%
mesa	8.0%	1537.1	18.7	0.77	83.8%	76.8%	74.5%
parser	8.7%	971.7	10.9	0.49	46.9%	34.6%	28.4%
vortex	10.8%	1209.1	10.9	0.60	54.3%	29.0%	31.0%
vpr	9.2%	1438.7	15.3	0.72	62.2%	46.9%	45.6%

TABLE IV
LATENCIES OF ACCESSING LINES IN THE DROWSY CACHE

	Active	**Drowsy**
Hit	• 1 cycle	• 1 cycle — to wake-up line • 1 cycle — to read / write line
Miss	• 1 cycle — to find line to replace • *n* cycle — to access upper memory hierarchy	• 1 cycle — to find line to replace • 1 cycle — to wake-up line (overlapped with the upper memory access latency) • *n* cycle — to access upper memory hierarchy

Note that we use one cycle for the cache access and wake-up latencies.

drowsy cache. As long as the drowsy cache can transition between drowsy and awake modes relatively quickly, simple algorithms should suffice.

The right side of Table III contains information about how quickly the working set of the workloads are changing. The results in the table specify what fraction of references in a window are to lines that had been accessed 1, 8, or 32 windows before. This information can be used to gauge the applicability of control policies that predict the working set of applications based on past accesses. As it can be seen, on many benchmarks (e.g., *bzip* and *gcc*), a significant fraction of lines are not accessed again in a successive drowsy window, which implies that past accesses are not always a good indication of future use. Aside from the *equake* and *mesa* benchmarks, where past accesses do correlate well with future accesses, most benchmarks only reaccess 40%–60% of the lines between windows. The implications of this observation are twofold. If an algorithm keeps track of which cache lines are accessed in a window, and only puts the ones into drowsy mode that have not been accessed in a certain number of past windows, then the number of awake to drowsy transitions per window can be reduced by about 50%. This, in turn, decreases the number of later wake ups, which reduces the impact on execution time. However, the impact on energy savings is negative because a larger fraction of lines are kept in full power mode, and in fact, many of those lines will not be accessed for the next several windows, if at all.

Table IV shows the latencies associated with the different modes of operation. No extra latencies are involved when an active line is accessed. Hits and misses are determined the same way as in normal caches for the active cache lines. However, a hit for the drowsy cache line costs one extra cycle to wake up the line, although this wake-up penalty may be overlapped with the upper memory such as L2 or main memory-access latency.

C. Policy Evaluation

In this section, we evaluate the different policy configurations with respect to their impact on runtime and the fraction of cache lines that are in drowsy mode during the execution of SPEC2000 benchmarks. All our algorithms work by periodically evaluating the contents of the cache and selectively putting cache lines into drowsy mode. The following parameters can be varied.

- *Update window size*: specifies in cycles how frequently decisions are made about which cache lines are put into drowsy mode.
- *Wake-up latency*: the number of cycles for waking up drowsy cache lines. We consider one, two, or four-cycle transition times, since our circuit simulations indicate that

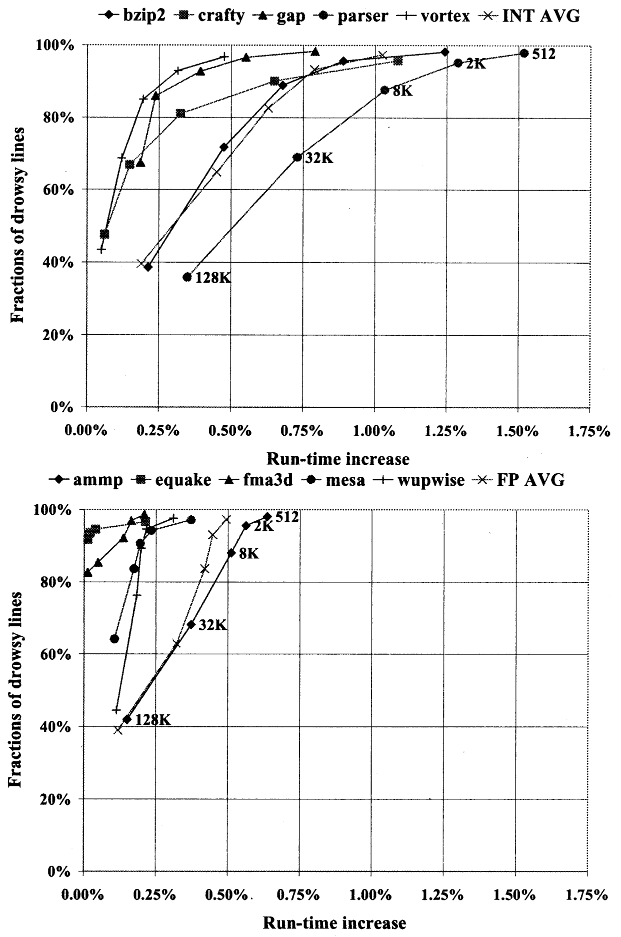


Fig. 7. Impact of window size on the runtime and fraction of drowsy lines. We use a *simple* policy with 128 K, 32 K, 8 K, 2 K, and 512 update window sizes and one-cycle drowsy line wake-up latency.

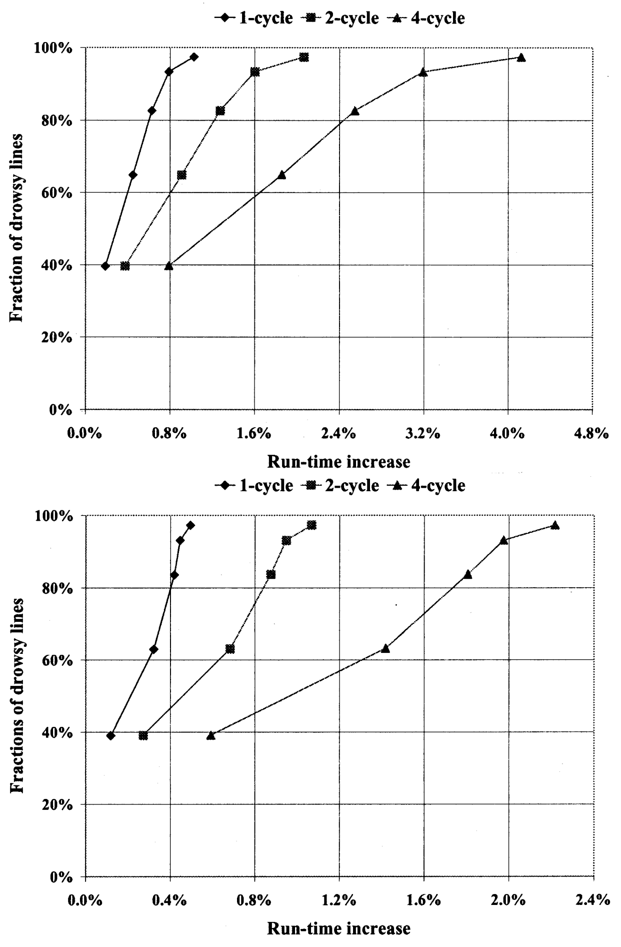


Fig. 8. Impact of increased drowsy access latencies. We use a *simple* with one-, two-, and four-cycle drowsy line wakeup latencies and 128 K, 32 K, 8 K, 2 K, and 512 update window sizes.

these are reasonable assumptions, with four cycles being a conservation extreme.

- *Simple or Noaccess policy*: The policy that uses no per-line access history is referred to as the *simple* policy. In this case, all lines in the cache are put into drowsy mode periodically (the period is the window size). The noaccess policy means that only lines that have not been accessed in a window are put into drowsy mode.

The detailed simulation methodology and processor parameters are discussed in Section VI-A.

Fig. 7 shows how the update window size impacts the runtime and the fraction of drowsy lines using a *simple* policy with a one-cycle wake-up latency. For clarity, we are showing only a subset of the benchmark. As we decrease the update window size, we have larger fractions of drowsy lines, which implies that we are able to reduce more leakage power dissipation. However, this comes along with the runtime increases of the workloads. Also, at the same update window size, the floating-point benchmarks show less runtime increase compared with the integer ones, because the floating-point applications have more temporal locality, accessing a specific region of data cache more frequently than other regions of the cache according to our experimental results. On the give processor configuration; the sweetspot, where the energy-delay product is maximized, is around 2 K cycles in the *simple* policy with the one-cycle wake-up latency.

From the entire SPEC2000 benchmarks, the average fractions of drowsy lines are 97%, 93%, 83%, 64%, and 39%, while the average runtime increases are 0.76%, 0.62%, 0.52%, 0.39%, and 0.15% for 512, 2 K, 8 K, 32 K, and 128 K update window sizes. The reason for the relatively small impact of the drowsy wake-up penalty on the processor's performance is due to our use of a nonblocking memory system, which can handle a number of outstanding loads and stores while continuing execution of independent instructions. Moreover, the drowsy wake-up penalty is usually only incurred with load instructions, because stores are put into a write buffer, which, if not full, allows execution to continue without having to wait for the completion of the store instruction. In terms of the average leakage power reduction, roughly 85% of leakage power can be reduced with a 0.62% runtime increase when the average fraction of drowsy lines are 93% and a drowsy cache line consumes 10% leakage power of an awake line.

The impact of increased wake-up latencies is shown in Fig. 8; the graphs in the figures show the runtime increases of the processor using a *simple* policy with one-, two-, and four-cycle wake-up latencies and 512, 2 K, 8 K, 32 K, and 128 K update window sizes. According to the experimental results, the fraction of drowsy lines remains relatively constant, but the runtime is increased significantly as the wake-up latency is increased, because the extra number of wake-up cycles increases idle cycles in the processor pipeline. For the 512, 2 K, 8 K, 32 K, and 128 K update window sizes, the average runtime increases of the simple policy with the two-cycle (four-cycle) wake-up latency are 1.6% (3.2%), 1.3% (2.6%), 1.1% (2.2%), 0.8% (1.6%), and 0.3% (0.7%), respectively; the impact on the runtime is doubled as the wake-up latency is doubled, and the runtime increase trend is consistent with the results of one-, two-, and four-cycle wake-up latencies. To minimize the runtime impact by the wake-up penalty, it is necessary to use a voltage controller that wakes up the drowsy cache line as fast as possible, but this increases the area overhead as well as dynamic power dissipation of the voltage controller (see Section III-B for the wake-up latency versus voltage controller size). To strike a balance, we will use two-cycle wake-up latency in the rest of the paper.

Fig. 9 contrasts the *noaccess* and the *simple* policies. The main question that we are trying to answer is whether there is a point to keeping any per-line statistics to guide drowsy decisions or if the indiscriminate approach is good enough. We show three different configurations for each benchmark on the graph: the *noaccess* policy with a 2 K-cycle window and two configurations of the *simple* policy (4 K- and 2 K-cycle windows). In all cases, the policy configurations follow each other from bottom to top in the aforementioned order. This means that in all cases, the *noaccess* policy has the smallest fraction of drowsy lines, which is to be expected, since it is conservative about which lines are put into drowsy mode.

The benchmarks on the graph can be partitioned into two groups: ones on lines whose slopes are close to the vertical, and ones on lines that are more horizontal, and thus, have a smaller positive slope. All the benchmarks that are close to the vertical are floating-point benchmarks, and their orientation implies that there is very little or no performance benefit to using the *noaccess* policy or larger window sizes. In fact, the *mgrid* benchmark in the graph has a slight negative slope, implying that not only would the simpler policy win on power savings, it would also win on performance. However, in all cases, the performance difference is negligible and the potential leakage power improvement is under 5% in most floating-point applications. The reason for this behavior is the very bad reuse characteristics of data access in these benchmarks. Thus, keeping lines awake (i.e., *noaccess* policy, or larger window sizes) is unnecessary and even counterproductive.

This anomalous behavior is not replicated on the integer benchmarks, where, in all cases, the *noaccess* policy wins on performance, but saves the least amount of power. Does this statement imply that if performance degradation is an issue, then one should go with the more sophisticated *noaccess* policy? It does not. The slope between the upper two points on each line is almost always the same as the slope between the bottom two points, which implies that the rates of change between the datapoints of a benchmark are the same; the data point for the *noaccess* policy should be able to be matched by a different configuration of the *simple* policy. We ran experiments to verify this hypothesis and found that a window size of 8 K of the *simple* policy comes very close to the coordinates for the *noaccess* policy with a window size of 2 K. We find that the *simple* policy with a window size of 4 K cycles reaches a reasonable compromise between simplicity of implementation, power savings, and performance. The impact of this policy on leakage energy is evaluated in Section VI-B.

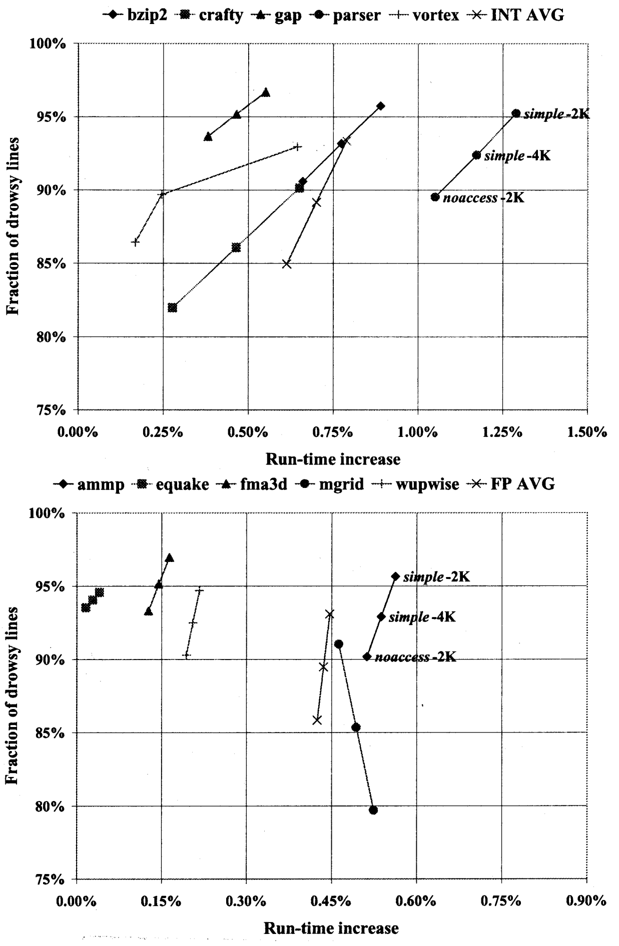


Fig. 9. Comparison of *noaccess* and *simple* policies. The bottom markers on each line correspond to the *noaccess* policy with 2 K-cycle window, the markers above it represent the *simple* policy with 4 K- and 2 K-cycle windows, respectively.

V. Drowsy Instruction Caches

In this section, we propose a new microarchitectural control technique for making drowsy instruction caches (as opposed to data caches that were proposed in Section IV). We found that while our previous algorithm was very effective for data caches, it does not work well for instruction caches due to the different locality characteristics between the caches. When we evaluate the *simple* policy, putting all the cache lines into drowsy mode every n cycles, for both data and instruction caches, the experimental results show that this algorithm, which was designed for data caches, is not as effective for instruction caches. Fig. 10 shows the runtime increase and the fraction of drowsy lines, which is proportional to leakage power reduction, of workloads using the *simple* policy with a one-cycle wake-up latency and a 4 K-cycle update window size, meaning that all cache lines are put into drowsy mode every 4 K cycles. According to the

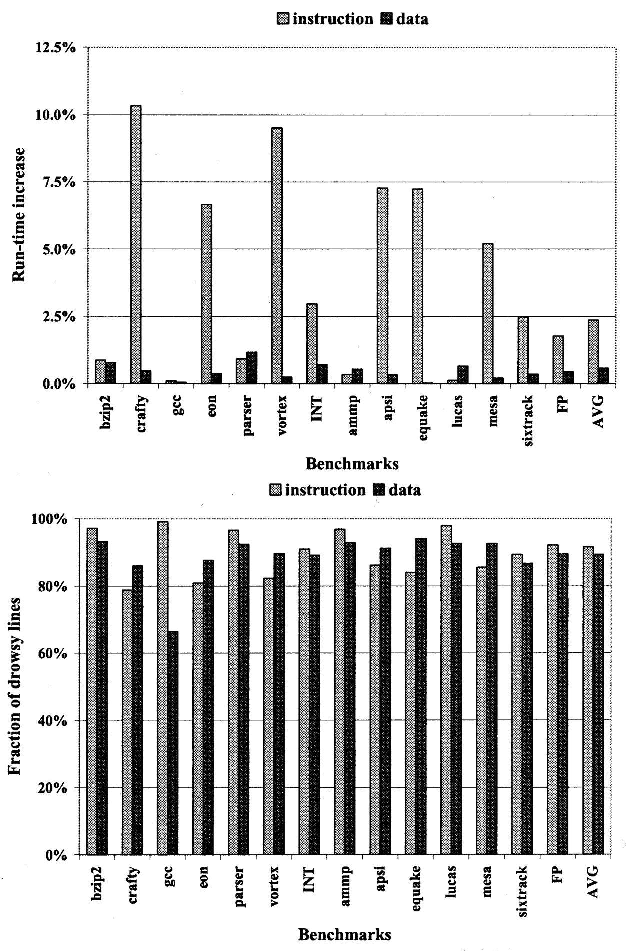


Fig. 10. Processor, runtime increase, and the fractions of drowsy lines of *simple* policy. We use a *simple* policy with a one-cycle drowsy-line wake-up latency and a 4 K-cycle update window size for both L1 instruction data caches (see Section VI-A for the detailed processor configuration).

experimental results shown in Fig. 10, the worst case runtime increase is as much as 10.3% (*crafty*), and the average is 2.4% runtime increase, although we employ an aggressive one-cycle wake-up latency for the 32 kB two-way set associative instruction cache. This is in sharp contrast with the *simple* policy for the data cache, where the worst runtime increase is no more than 1.2%, and the average is 0.6% runtime increase, although the fractions of drowsy lines for both caches are similar. These experimental results imply that the application of the technique developed in Section IV to instruction caches can result in poor processor performance compared with the data cache. The main reason for this behavior is that data caches tend to have better temporal locality, while instruction caches tend to have better spatial locality.

Several researchers have proposed the use of subbanks as a means of reducing power consumption in caches. In [31], the cache is partitioned into several subbanks, and on each access, only a limited set of subbanks are checked for their contents. This approach reduces the dynamic power consumption at the cost of slightly increasing the cache access time due to additional decoder logic for indexing the subbanks. In [19], a leakage power reduction circuit technique is applied to the subbank that has been most recently accessed (see Section II-A for the detailed technique). However, it requires a finite wake up or precharge time to access a subbank which has not been accessed or has been in sleep mode. This incurs a processor performance penalty to wake up the next target subbank on the critical path, where the penalty for the wake up can be several cycles. According to our experiments, the use of architectural techniques employed in [19] (we call it "*simple* wake up") can result in a worst case runtime increase of 15.7% (*crafty*), and the average runtime impact is 3.6% for the 32 kB two-way set associative instruction cache, even when assuming an aggressive one-cycle wake-up latency.

To minimize the performance impact as well as leakage power consumption, we propose a low-leakage instruction cache architecture using our drowsy circuit technique combined with cache subbanking and two subbank prediction techniques. Our prediction techniques rely on the insight that transitions between subbanks are often correlated with specific types of instructions. For example, the program counter, which is the instruction cache access index, remains in small cache regions for relatively long periods of time due to the program loops. On the other hand, there are often abrupt changes in the accessed cache region when subroutines are called, or when the subroutines return, and long-distance unconditional branches are executed. Most conditional branches stay within the current cache region, and it is rare that the these branches jump across page boundaries.

A. Cache Architecture Using DVS and Memory Subbanking Techniques

Fig. 11 illustrates a 16 kB direct-mapped cache architecture using a DVS technique and four 4 kB subbanks. The predecoder identifies which subbank is accessed with a cache access address, and the decoder in each subbank selects an appropriate cache line in the subbank with the predecoded address. In this technique, the predecoder includes the wake-up logic and drives a wake-up signal to the target subbank. One subbank is active, while the other subbanks are in the drowsy mode. Whenever the processor accesses cache lines in the other subbank and the cache hits, the predecoder activates that subbank, and puts the current subbank into the drowsy mode. During the wake-up of the next subbank, the processor halts, because we must wait to reinstate the power supply levels to the normal voltage level in the manner of the DVS technique explained earlier. This is the wake-up latency. If the cache misses, the wake-up latency can be hidden during the miss-handling cycles. Therefore, it is critical to wake up the next subbank in case of a cache hit as soon as possible to reduce performance degradation.

Each cache line consists of a drowsy cache line, illustrated in Fig. 12. In this architecture, compared with the drowsy cache line represented in Fig. 5, we do not need a drowsy bit for each line. Instead, the wake-up logic in the predecoder sends active low wake-up signals to the target subbank. Furthermore, we modify the precharge circuit to reduce the leakage current through the access pass transistors in the conventional 6T memory cell by gating the precharge signal with the wake-up signal. With this precharge gating technique, we do not need to use *high*-V_{TH} transistors to reduce the leakage power through

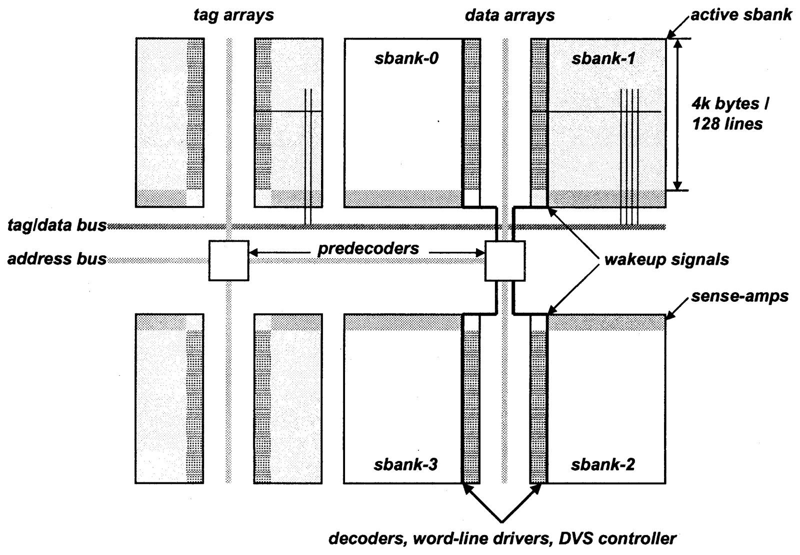


Fig. 11. Instruction cache architecture using memory subbanking and DVS techniques.

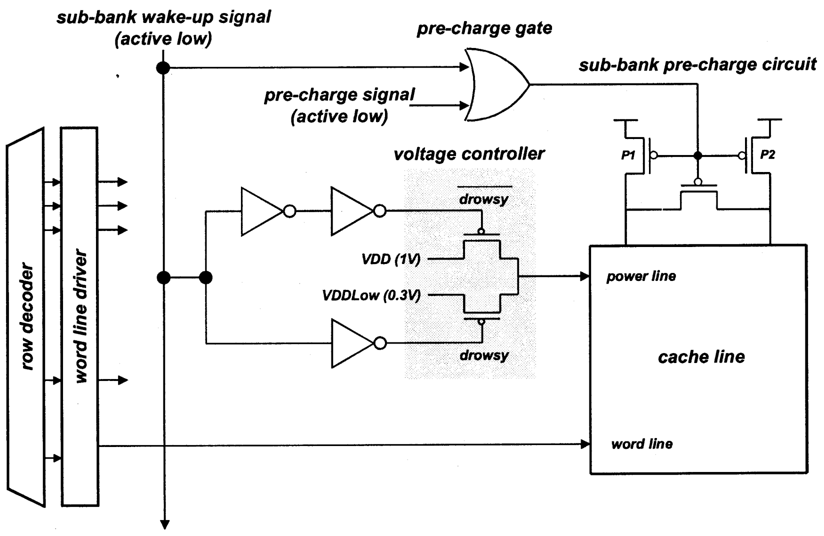


Fig. 12. Implementation of a drowsy cache line and a precharge circuitry for a subbank.

the access transistors, which either improves access time or further reduces the leakage power of the caches, compared with the technique in Section IV-A. For a set-associative cache, we can organize the subbanks in the following way. Assuming a 32 kB four-way associative cache, we can organize each way with two 4-kB subbanks, and we activate only one subbank among eight 4-kB subbanks.

B. Next Subbank Prediction Techniques

Without any prediction for the next target subbank, we lose significant performance as a result of the wake-up penalties, as mentioned in Section V-A. However, according to our analysis, most transitions between the subbanks are caused by subroutine calls, returns, and long-distance jumps, and these transition points or addresses from one to another subbank in the instruction caches are quite predictable. If we maintain and look up the transition address history of the instruction cache in a buffer structure, we will be able to wake up the next subbank before accessing it and not pay any additional wake-up penalty. To reduce the runtime impact, we introduce two next subbank prediction techniques in this section.

1) Next Subbank Prediction Buffer (NSPB): Fig. 13 illustrates a NSPB technique for an instruction cache. In the NSPB, each entry contains a *predicted subbank index*, a *valid bit*, a *subbank transition sign instruction* address fetched one cycle earlier than a *subbank transition triggering instruction*, assuming the subbank wake-up latency is one cycle. In Fig. 13, `str [r1],r3` at `0002fff8H` is the subbank transition sign instruction and `jmp 00182ffcH` at `0002fff8H` is the subbank transition, triggering instruction causing a transition from subbank three to two. The NSPB is built with a *content addressable memory* (CAM), and the address field of the NSPB entry is used for an associative search index. To detect the subbank transition and wake up the target subbank one cycle ahead, the NSPB entry should contain an instruction address fetched one cycle earlier than the triggering instruction, assuming that the processor fetches one instruction per cycle in this example.

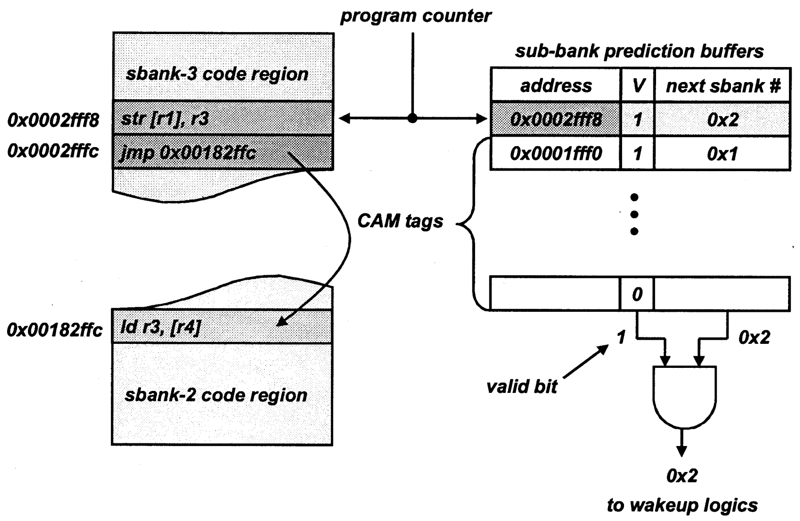


Fig. 13. Next subbank prediction buffer.

In each instruction fetch cycle, the processor looks up the NSPB entries based on the program counter value (e.g., 0002fff8H). When there is any matched address in the NSPB, the processor wakes up a subbank based on the predicted next target subbank index (e.g., 2). Although a subbank transition triggering instruction is fetched in the consecutive cycle (e.g., jmp 00182ffcH at 0002fff8H), the processor does not need to wait until the next subbank is woken up, because it has been woken up in the previous cycle. For the processor fetching and executing multiple instructions per cycle, the last instruction address accessed in the previous cycle or cache line is used for the NSPB index (the processor cannot access two different cache lines in a cycle). Whenever the processor encounters a subbank transition, it checks whether the NSPB sent any predicted subbank index in the previous cycle, an NSPB hit, and it compares the predicted subbank index with the current one and updates the prediction information accordingly. In case of a NSPB miss, it inserts the current subbank index with the previous cycle instruction address to an empty NSPB entry. If the NSPB entries are full, the prediction information in a *least recently used* (LRU) entry is replaced, which may cause NSPB misses in the future cycles. Therefore, the prediction accuracy will be dependent on the number of entries in the prediction buffers.

Fig. 14 shows the NSPB prediction accuracies and the processor runtime increases with the NSPB. The average prediction accuracies for 32-, 64-, 128-, and 256-entry NSPBs are 51.4%, 59.2%, 71.6%, and 78.4%, respectively; increasing the number of entries in NSPB results in improved prediction accuracy. While the average processor runtime increase with “simple wake up” or no prediction is 3.62%, those of the processor with 32-, 64-, 128-, and 256-entry NSPBs are 1.9%, 1.6%, 1.0%, 0.7%, and 0.8%, respectively. It is clear that the average processor runtime increase is inversely proportional to the prediction accuracy; improved prediction accuracies reduce the processor runtime increase impacts by 47.5%, 56.3%, 71.3%, and 79.5%. In particular, the prediction accuracy for *bzip2, ammp* (shown in Fig. 15), *applu, art, lucas, mgrid, mcf, swim,* and *vpr* (not shown in Fig. 14) is approaching 100%, eliminating the

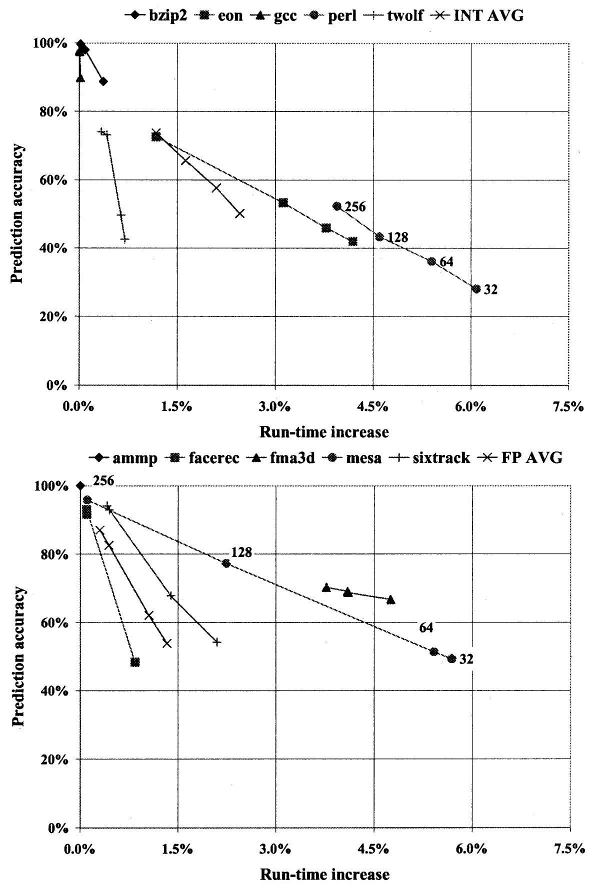


Fig. 14. NSPB prediction accuracy and the processor runtime increase with the NSPB.

runtime increase completely. In addition, the prediction accuracies for the floating-point applications are better than those for the integer applications because of their greater spatial locality.

2) Next Subbank Predictor in Cache Tag (NSPCT): However, the area and power consumption of the NSPB can be significant. Here, we propose another prediction technique in which we extend cache tags instead of using a

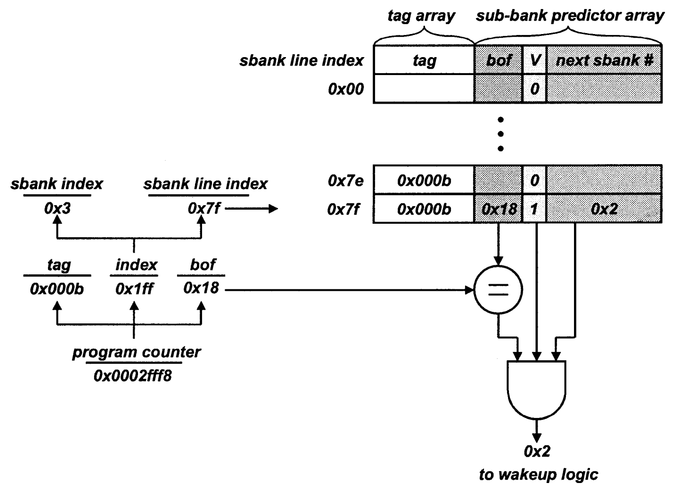


Fig. 15. Next subbank predictor in cache tags.

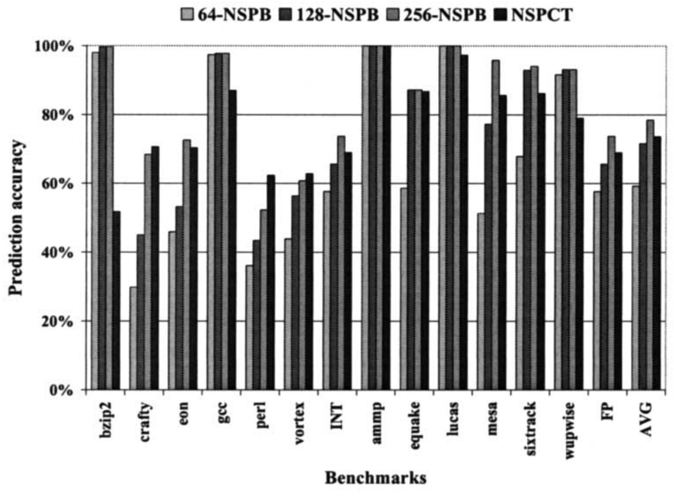


Fig. 16. NSPCT predictor accuracy comparison with the NSPB's. We use 64-, 128-, and 256-entry NSPBs with a one-cycle drowsy subbank wakeup latency and a 32 kB two-way set associative instrucion cache (see Section VI-A for the detailed processor configuration).

separate fully associative memory structure like the NSPB. Fig. 15 illustrates the extended cache tags to support the next subbank prediction, NSPCT. Each extended tag entry contains the next subbank index, a valid bit, and a cache block address of a subbank transition sign instruction (*bof* field in Fig. 15) as well; we can reduce a substantial amount of power and hardware by sharing the decoder of the tags with the predictor field in the extended tags. The update of the predictor is performed in the same way as the NSPB technique. However, a separate write port is not necessary to update the predictor, because the predictor update only occurs on a mispredicted subbank transition; the processor updates the predictor in the previous subbank while it accesses the next subbank. Whenever the processor accesses the cache, it compares the block address of the current instruction, and checks the validity of the prediction information. If the address matches and the prediction information is valid, the processor sends the predicted next subbank index to the wake-up logic.

The disadvantage of this technique is that the prediction information is lost when the cache tag line containing the prediction information is replaced. Regardless of the disadvantage, however, the number of predictor entries in this technique is proportional to the cache (set) size and only a small number of additional bits are added to each cache tag entry. Therefore, the NSPCT technique can be less expensive but more accurate than the smaller NSPB requiring CAM tags. In Fig. 16, we compare the NSPCT predictor accuracy with 64-, 128-, and 256-entry NSPBs; the average prediction accuracy of the NSPCT is 73.6%. It shows better accuracy than that of a 128-entry NSPB. Furthermore, in some applications such as *crafty* and *perl*, the NSPCT outperforms the 256-entry NSPB.

VI. EXPERIMENTS

A. Processor Simulation Methodology

The architectural simulator used in this study is derived from the SimpleScalar/Alpha 3.0 tool set [32], a suite of functional and timing simulation tools for the Alpha AXP ISA. Simulation is execution driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Specifically, we extended `sim-outorder` to reflect the performance impact of waking up the drowsy cache lines or subbanks in the L1 data and instruction caches. The processor simulation parameters are listed in Table V. The processor microarchitectural parameters model a high-end microprocessor similar to an Alpha 21 264. We augment it with a generous supply of functional units, aggressive main memory, L1 caches, and a register file with a latency to reduce the execution variability due to resource constraints and memory latencies. To perform our evaluation, we collected results from all 25 of the SPEC2000 benchmarks [33]. All SPEC programs were compiled for a Compaq Alpha AXP-21 264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system, using full compiler optimizations ($-O4$). The simulations were run for 100 million instructions using the SPEC reference inputs. We used the utility Early SimPoints [34] to pinpoint program locations of peak performance so that we can find simulation regions that most stress, in particular, instruction caches.

B. Runtime Impact and Leakage Reduction of Drowsy Caches

To examine the effectiveness of the proposed technique, we compare our drowsy cache techniques with the state-of-the-art *cache decay* technique [21]. In the cache decay technique, it is critical to consider the extra energy dissipation of the L2 cache access, because decayed or dead L1 cache lines result in additional accesses to the L2 cache. We assume that the 512 kB four-way set associative L2 cache is designed with the subbanking technique of [31] to improve both the access time and energy dissipation efficiency of the cache. We estimated the energy dissipation of the L2 cache (380pJ per access) using CACTI 3.2 [30] with the number of subbanks that result in the least energy dissipation per access. To benchmark the two techniques, we compare the normalized leakage energy: the ratio of total leakage energy dissipation by L1 cache (plus dynamic energy dissipation by the extra L2 cache accesses for the cache decay technique) divided by the total leakage energy consumed in the same-size regular cache. To calculate the leakage energy from the leakage power, we need to estimate the cycle time of

TABLE V
PROCESSOR SIMULATION PARAMETERS

Parameters	Value
fetch / issue / decode/ commit width	4 instructions each
fetch queue / speed	16 instructions / 1×
branch prediction	bimod / 2K
ROB size	64 entry
LSQ size	32 entry
integer ALUs/multi-divs / memory ports	4 / 1 / 2
floating point ALUs / multi-divs	4 / 1
functional unit latencies	INT: mul 3, div 20, all others 1 FP: adder 2, mul 4, div 12, sqrt 24
memory bus width / latency	4 bytes / 80 and 8 cycles for the first and inter chunks
instruction / data TLB's	128 entry / 32 entry in each way, 8KB page size, fully-associative, LRU, 28-cycle latency
L1 caches	32KB, 2-way, 32B blocks, LRU, 1 cycle latency for both inst and data caches, write-back
L2 unified cache	512KB, 4-way, 64B line block, LRU, 12 cycle latency

TABLE VI
COMPARISON OF THE PROCESSOR RUNTIME IMPACT AND LEAKAGE ENERGY REDUCTION BETWEEN DROWSY DATA CACHE AND CACHE DECAY TECHNIQUES

FP	Run-time impact (%)		Norm. leakage (%)		INT	Run-time impact (%)		Norm. leakage (%)	
	drowsy	decay	drowsy	decay		drowsy	decay	drowsy	decay
ammp	0.54	0.12	34 (22)	81	bzip2	0.77	0.38	34 (22)	35
applu	0.50	0.00	37 (26)	41	crafty	0.46	1.08	39 (28)	114
apsi	0.32	0.02	35 (23)	31	eon	0.36	0.13	38 (26)	38
art	0.61	0.01	38 (27)	44	gap	0.46	0.90	32 (20)	31
equake	0.03	0.04	33 (21)	10	gcc	0.06	0.00	53 (45)	96
facerec	0.15	0.03	32 (20)	23	gzip	2.07	1.23	34 (22)	85
fma3d	0.14	0.87	32 (20)	141	mcf	0.66	0.04	34 (22)	35
galgel	0.31	0.00	51 (42)	68	parser	1.17	1.75	35 (22)	63
lucas	0.66	0.00	34 (22)	25	perl	0.87	6.97	36 (24)	224
mesa	0.21	0.09	34 (22)	22	twolf	0.87	0.29	35 (22)	36
mgrid	0.49	0.00	40 (28)	42	vortex	0.25	0.78	36 (25)	126
sixtrack	0.34	0.18	39 (27)	56	vpr	0.89	0.84	36 (24)	76
swim	0.61	0.00	37 (25)	34	avg	0.57	0.64	37 (25)	59
wupwise	0.20	0.01	34 (22)	50					

The normalized leakage figures in parenthesis are calculated using the leakage power of the 6 T SRAM cell implemented with 300 mV *high-*V_{TH}-access transistors.

the processor, which is roughly 16 × FO4 delay, according to [29]. The HSPICE simulation with the projected 70–nm technology shows that 16 × FO4 is around 193 ps.

1) Drowsy Data Cache: Table VI shows the comparison of the processor runtime impact and the leakage power reduction between the *drowsy data cache* and *cache decay* techniques. We use the *simple* policy with a 4 K-cycle drowsy window size and one-cycle wake-up latency for the drowsy data cache, and the 8 K-cycle decay window size as appeared in [21] for the cache decay technique. The normalized leakage figures in the parenthesis are recalculated using the leakage power of the 6T SRAM cell implemented with 300 mV *high-*V_{TH} access transistors. For all the results in the table, we conservatively assume that there are only 20 tag bits for the 32 kB two-way set associative cache (corresponding to 32-bit addressing) per line, which translates into 6.9% of the bits in a cache line. The experimental results show that our implementation of a drowsy data cache can reduce the total leakage energy consumed in the data cache by more than 60% without significant runtime impact (0.57% in average). If the 6T SRAM cell is implemented with 300 mV *high-*V_{TH}-access transistors, the leakage energy could potentially be reduced by 75% with a 6% increase in cache access time.

Compared with the cache decay technique, the drowsy data cache shows less runtime impact, as well as more leakage energy reduction, on average. Also, the drowsy data cache has

TABLE VII
COMPARISON OF THE PROCESSOR RUNTIME IMPACT AND LEAKAGE ENERGY REDUCTION BETWEEN DROWSY INSTRUCTION CACHE AND CACHE DECAY TECHNIQUES

FP	**Run-time impact (%)**		**Norm. leakage (%)**		**INT**	**Run-time impact (%)**		**Norm. leakage (%)**	
	drowsy	**decay**	**drowsy**	**decay**		**drowsy**	**decay**	**drowsy**	**decay**
ammp	0.00 (99)	1.83	25	28	bzip2	0.83 (52)	0.69	24	8
applu	0.00 (99)	0.22	24	10	crafty	3.75 (71)	6.26	26	118
apsi	2.01 (67)	0.14	25	25	eon	2.54 (70)	0.31	25	39
art	0.00 (99)	0.21	24	1	gap	1.93 (70)	0.55	25	40
equake	0.71 (87)	0.07	25	19	gcc	0.03 (78)	2.37	24	15
facerec	0.39 (76)	0.26	24	5	gzip	1.00 (87)	0.24	25	6
fma3d	4.52 (74)	0.08	26	54	mcf	0.19 (81)	0.23	24	3
galgel	0.00 (27)	0.04	24	1	parser	0.51 (72)	0.44	24	8
lucas	0.01 (97)	0.00	24	2	perl	2.34 (62)	1.84	25	68
mesa	0.67 (86)	0.37	25	23	twolf	0.30 (74)	2.98	24	56
mgrid	0.01 (99)	0.10	24	19	vortex	4.34 (63)	8.43	26	147
sixtrack	1.23 (86)	0.45	25	15	vpr	0.00 (99)	0.16	24	5
swim	0.00 (99)	0.28	24	9	avg	0.79 (74)	0.99	25	23
wupwise	0.79 (79)	0.32	24	20					

The figure in parenthesis represents the prediction accuracy of the NSPCT in percentage.

relatively uniform leakage reduction over the entire SPEC2000 benchmark programs, while the decay data cache does not. Furthermore, in some applications such as *crafty, fma3d, perl,* and *vortex* (see the shaded region in Table IV), the system using the cache decay technique dissipates more energy than the regular cache and it shows more runtime impact, due to the relatively high number of extra L2 cache accesses incurred by the aggressive turn-off of the live cache lines. Therefore, to prevent such a side effect, a sophisticated tuning technique for the individual application is required for the cache decay technique. In terms of extra hardware, both techniques require additional transistors to control the power mode of the cache lines. Comparing the access time of caches, both techniques also have additional gates in their critical paths; the power and word-line gating transistors for the cache decay and drowsy cache, respectively, although they impact the access time very slightly.

2) Drowsy Instruction Cache: Table VII shows the comparison of the processor runtime impact and the leakage power reduction between the *drowsy instruction cache* and *cache decay* techniques. We use the 4-kB subbank size, NTPCT, and one-cycle wake-up latency for the drowsy instruction cache. We choose the NTPCT technique because it shows modest prediction accuracy (74%) with minimal hardware resource, compared with the NSPB technique; the parenthesized figures in Table VII represent the prediction accuracy of the NSPCT in percentage. We conservatively assume that there are an additional seven bits for the NSPCT prediction field (three bits for the *bof* field, three bits for the predicted subbank field, and one bit for the valid bit), which translates into 2.7% of the bits on a cache line. For the cache decay technique, we use the 16K-cycle decay window size showing the most energy saving among the window sizes from 4 K to 128 K cycles. The experimental results show that our implementation of a drowsy instruction cache can reduce the total leakage energy consumed in the data cache by more than 77% with a modest runtime impact (0.79% in average).

Compared with the cache decay technique, the drowsy instruction cache shows less runtime impact and comparable leakage energy savings on average. As noted in Section V-B, the prediction technique reduces the runtime impact from 3.64% (no prediction) to 0.76% (NSPCT) on average, which can be translated into a runtime impact reduction of 78%. Also, in some applications, such as *crafty* and *vortex* (see the shaded region in Table VII), the system using the cache decay technique dissipates more energy than the regular cache, and it shows more runtime impact due to the relatively high number of extra L2 cache accesses incurred by turning off live cache lines.

VII. CONCLUSION

The relative merits of both approaches depend strongly on the latency and size of L2 caches. If the L2 cache is larger than 512 K bytes, as is common in today's high performance microprocessors, the performance of the drowsy cache approach improves relative to that of the decay cache. Furthermore, in small low-power systems with no L2 cache, and in which L1 misses require an off-chip access, the drowsy cache approach shows strong advantage. During our investigations of drowsy data caches, we found that our simplest policy, where cache lines are periodically put into a low-power mode without regard to their access histories, can reduce the cache's leakage power consumption by 60%–75%. Our comparisons with the cache decay algorithm, which uses a state-destructive *gated*-V_{DD} cache to reduce leakage power, indicate that the drowsy technique not only offers better energy savings but also lacks the pathological behavior (which can actually increase power consumption) that is inherent in the *gated*-V_{DD}-based technique.

The simple policy is not a solution to all caches in the processor. In particular, the L1 instruction cache does not do as well with the simple algorithm and only slightly better with the *noaccess* policy. Instead of applying the *simple* policy to the instruction cache, we proposed subbanked drowsy instruction caches. Our approach uses a subbank predictor that keeps only the predicted bank awake and puts the rest of the subbanks into a low-leakage drowsy mode, achieving leakage power reduction of more than 84% for a 32-kB cache. Our results show that the prediction technique using the extended cache tag can reduce the runtime overhead by 71% for the 32-kB two-way set associative cache, compared with the default policy where no prediction was employed.

We believe that our combination of a simple circuit technique with a simple microarchitectural mechanism provides sufficient leakage power savings at a modest performance impact to make more complex solutions unattractive. Since the cost of misprediction in a drowsy cache is low both in terms of power and performance overhead, it is especially useful for embedded processors that lack on-chip L2 caches. On a miss, a drowsy cache need only wake up the drowsy line that is already in the cache, as opposed to gated-V_{DD}-based designs, which would have to perform a costly off-chip access to main memory to load the line.

An open question remains as to the role of adaptability in determining the window size. We found that for a given machine configuration, a single static window size of 4 K cycles performs adequately on all of the SPEC2000 benchmarks. However, the optimum varies slightly for each workload, thus, making the window size adaptive would allow a finer power-performance tradeoff.

References

[1] "International Technology Roadmap for Semiconductors 2002 Edition," Semiconductor Industry Association, http://public.itrs.net.

[2] B. Doyle, R. Arghavani, D. Barlage, S. Datta, M. Doczy, J. Kavalieros, A. Murthy, and R. Chau, "Transistor elements for 30 nm physical gate lengths and beyond," *Intel Technol. J.*, vol. 6, pp. 42–54, May 2002.

[3] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *Proc. IEEE/ACM Int. Symp. Computer Architecture (ISCA25)*, July 1998, pp. 132–141.

[4] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan, "Scaling of stack effect and its application for leakage reduction," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, Aug. 2001, pp. 195–200.

[5] F. Hamzaoglu, Y. Ye, A. Keshavarzi, K. Zhang, S. Narendra, S. Borkar, M. Stan, and V. De, "Analysis of dual-VT SRAM cells with full-swing single-ended bit line sensing for on-chip cache," *IEEE Trans. VLSI Syst.*, vol. 10, pp. 91–95, Apr. 2002.

[6] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, June 1998, pp. 76–81.

[7] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic performance-setting for dynamic voltage scaling," in *Proc. ACM Int. Conf. Mobile Computing and Networking (MOBICOM-7)*, July 2001, pp. 260–271.

[8] S. Wolf, *Silicon Processing for the VLSI Era Volume 3—The Submicron MOSFET*. Sunset Beach, CA: Lattice Press, 1995, pp. 213–222.

[9] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. Vijaykumar, "*Gated-V_{DD}*: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, 2000, pp. 90–95.

[10] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches," in *Proc. IEEE/ACM Int. Symp. High-Performance Computer Architecture*, 2001, pp. 147–157.

[11] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *IEEE J. Solid-State Circuits*, vol. 30, pp. 847–854, Aug. 1995.

[12] N. Shibata, H. Morimura, and M. Harada, "1-V 100-MHz embedded SRAM techniques for battery-operated MTCMOS/SIMOX ASICs," *IEEE J. Solid-State Circuits*, vol. 3, pp. 1396–1407, Oct. 2000.

[13] T. Douseki, N. Shibata, and J. Yamada, "A 0.5–1 V MTCMOS/SIMOX SRAM macro with multi-V_{TH} memory cells," in *Proc. IEEE Int. SOI Conf.*, 2000, pp. 24–25.

[14] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano, "A low power SRAM using auto-backgate-controlled MT-CMOS," in *Proc. IEEE/ACM Int. Symp. Low Power Electronic Devices*, 1998, pp. 293–298.

[15] M. Ketkar and S. Sapatnekar, "Standby power optimization via transistor sizing and dual threshold voltage assignment," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 2002, pp. 375–378.

[16] J. Kao and A. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEE J. Solid-State Circuits*, vol. 35, pp. 1009–1018, July 2002.

[17] A. Agarwal, L. Hai, and K. Roy, "A single-V_t low-leakage gated-ground cache for deep submicron," *IEEE J. Solid-State Circuits*, vol. 38, pp. 319–328, Feb. 2003.

[18] N. Azizi, A. Moshovos, and F. Najm, "Low-leakage asymmetric-cell SRAM," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, 2002, pp. 48–51.

[19] S. Heo, K. Barr, M. Hampton, and K. Asanovic, "Dynamic fine-grain leakage reduction using leakage-biased bitlines," in *Proc. IEEE/ACM Int. Symp. Computer Architecture*, 2002, pp. 137–147.

[20] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. Clark, "Managing leakage for transient data: Decay and quasi-static 4T memory cells," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, 2002, pp. 52–55.

[21] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. IEEE/ACM Int. Symp. Computer Architecture (ISCA28)*, 2001, pp. 240–251.

[22] W. Zhang, J. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-directed instruction cache leakage optimization," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2002, pp. 208–218.

[23] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: *Simple* techniques for reducing leakage power," in *Proc. IEEE/ACM Int. Symp. Computer Architecture*, 2002, pp. 148–157.

[24] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Drowsy instruction caches—Leakage power reduction using dynamic voltage scaling and cache subbank prediction," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO-35)*, Nov. 2002, pp. 219–230.

[25] A. Keshavarzi, K. Roy, and C. Hawkins, "Intrinsic leakage in low power deep submicron CMOS IC's," in *Proc. IEEE Int. Test Conf.*, 1997, pp. 146–155.

[26] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[27] H. Qin and J. Rabaey. Deep sleep mode: SRAM leakage suppression using ultra low standby data retention voltage. presented at Gigascale Silicon Research Center Workshop. [Online]. Available: http://www.gigascale.org/pubs/talks/2003/oakland/

[28]. [Online]. Available: http://www-device.eecs.berkeley.edu

[29] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[30] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model WRL Res. Rep.,", 2002.

[31] K. Ghose and M. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proc. IEEE/ACM Int. Symp. Low Power Electronics and Design*, 1999, pp. 70–75.

[32] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer Mag.*, vol. 35, pp. 59–67, Feb. 2002.

[33] Standard Performance Evaluation Corp. [Online]. Available: http://www.specbench.org

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, Oct. 2002, pp. 45–47.

Nam Sung Kim was born in Seoul, Korea, on May 16, 1974. He received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1997 and 2000, and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2003.

He was with the VLSI system laboratory in KAIST, participating in the 386, 486, and Pentium-compatible microprocessor design and verification projects from 1995 to August 2000. His work has focused on computer architecture, digital circuit, and computer-aided design (CAD), with particular emphasis on low-power and low-complexity computer design at the circuit and microarchitecture boundary, and power and complexity estimation tools for the embedded systems.

David Blaauw (M'93) received the B.S. degree in physics and computer science from Duke University, Durham, NC, in 1986, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1988 and 1991, respectively.

He was with the Engineering Accelerator Technology Division, IBM Corporation, Endicott, as a Development Staff Member, until August 1993. From 1993 till August 2001, he worked for Motorola, Inc., Austin, TX, were he was the Manager of the High Performance Design Technology group. Since August 2001, he has been on the faculty of the University of Michigan, Ann Arbor, as an Associate Professor. His work has focused on VLSI design and CAD, with particular emphasis on circuit analysis and optimization problems for high-performance and low-power designs. He was the Technical Program Chair and General Chair for the International Symposium on Low Power Electronics and Design in 1999 and 2000, respectively, and was the Technical Program Cochair and member of the Executive Committee the ACM/IEEE Design Automation Conference in 2000 and 2001.

Trevor Mudge (S'74–M'77–SM'84–F'95) received the B.Sc. degree from the University of Reading, Reading, U.K., in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively.

Since 1977, he has been on the faculty of the University of Michigan, Ann Arbor. He recently was named the first Bredt Family Professor of Electrical Engineering and Computer Science after concluding a 10-year term as the Director of the Advanced Computer Architecture Laboratory, a group of eight faculty and about 70 graduate students. He is author of numerous papers on computer architecture, programming languages, VLSI design, and computer vision. He has also chaired about 30 theses in these research areas. His research interests include computer architecture, computer-aided design, and compilers. In addition to his position as a faculty member, he runs Idiot Savants, a chip-design consultancy, and he is technical advisor to several venture firms.

Dr. Mudge is a Member of the ACM, the IEE, and the British Computer Society.

Krisztian Flautner (M'00) received the B.S.E., M.S.E., and Ph.D., degrees in computer science and engineering from the University of Michigan, Ann Arbor.

He is currently the Director of Advanced Research at ARM Limited. His research has explored the relevance of multithreading for interactive desktop workloads, architectural, and circuit techniques for low-power processors and automatic power-management algorithms for controlling dynamic voltage and threshold scaling. He is also the Architect of ARM's Intelligent Energy Manager (IEM) technology, his research interests are focused on simple ideas that enable high-performance low-power processing platforms with sophisticated software environments.