2003－ARC－155　(13)
2003／11／28

Way-variable Caches for Static Power Reduction

Luong Dinh Hung,† Chitaka Iwama ,† Niko Demus Barli,†
Shuichi Sakai† and Hidehiko Tanaka†

Power consumption due to leakage increases rapidly as devices scale to smaller geometries. We propose way-variable caches that dynamically adapt the number of active ways according to runtime requirements. By entirely gating the unused ways from the voltage supply, the leakage can be significantly reduced. We then apply an original algorithm utilizing data access locality to make proper resizing decisions. Performance evaluations are done with a superscalar processor model having 16-KB, 4-way set-associative L1 instruction and data caches. The results verified that, on average, 1.7 ways of the instruction cache can be disabled with only 1.3% performance degradation in the case of instruction cache. The values are 1.5 ways and 1.1% in the case of the data cache.

1. Introduction

Shrinking of CMOS devices following Moore's law offers us smaller, faster but often leakier transistors. The static power consumption due to leakage current has been becoming a large fraction of total power consumption of recent microprocessors. It has been reported that static power dissipation counts for more than 20% of entire power consumption in Itanium processor, implemented in 0.13um process (3.4 times the percentage of static power of previous generation implemented in 0.18um process) [1]. Due to the exponential growth in the magnitude of leakage, static power is likely to be the limiting factor in power-constraint microprocessor designs in the future.

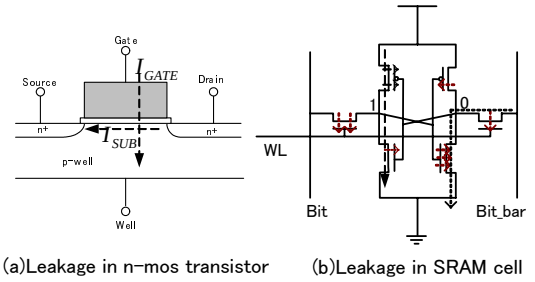

Fig. 1 Leakage in (a) CMOS transistor and (b) in SRAM cell

Figure 1-a shows two most important components of leakage in a CMOS transistor. As the gate length becomes shorter, the potential barrier could be lowered by the source-drain voltage (Vds). This causes significant *subthreshold leakage* current flowing between the source and drain even when the applied gate voltage(Vgs) is far below the threshold voltage (Vth). At the same time, the gate dielectric is demanded to be thinner according to the scaling rule. This increases the capability of electrons tunneling through the dielectric, resulting in larger *gate leakage*.

Our research focuses on reducing the leakage in caches. Modern microprocessor designs devote a large fraction of chip to multi-level caches, making them an reasonable target for attacking leakage. Figure 1-b shows several main leakage paths existing in a typical 6-transistor SRAM cell. There are two main subthreshold leakage paths; one leakage path inside the cell and one coming from the bitline. Each passes over two transistors in which one is on and the other is off. There is some amount of gate leakage at every transistor. It is known that the nmos exhibits more gate leakage than the pmos and the nmos transistor in on state causes much gate leakage [10].

It is widely known that the characteristics of cache requirements in microprocessors is quite different among applications as well as among phases of individual application execution. In this research, we propose *way-variable caches* in which some ways of the set-associative cache can be disabled during periods of modest cache activity, while more cache ways may remain operational for more cache-intensive periods. High degree of leakage reduction (both subthreshold leakage and gate leakage) can be achieved in the disabled ways. We then propose a novel algorithm considering the locality characteristic of the data access in order to make proper way controlling decisions.

The remainder of this paper is organized as follows. Section 2 describes some related work. In section 3, we explain the structure organization of way-variable caches as well as the proposed resizing algorithm. Evaluation is described in section 4. Finally, section 5 concludes the paper.

2. Related Work

There is a vast body of literature focusing on cache leakage reduction in recent years. The

† Graduate School of Information Science and Technology, The University of Tokyo

drowsy cache [2] utilizes Dynamic Voltage Scaling (DVS) technique. In drowsy cache, cache lines are periodically turned to *drowsy mode* (low leakage mode) by connecting them to the low voltage source. Only those cache lines being accessed are woken up and restored to the normal value of Vdd. However, the subthreshold leakage from bitlines cannot effectively reduced as well as additional cycles is needed to wake up the cache lines in drowsy mode.

The *decay cache* [4] employs gated-Vdd technique [8] that is widely used for leakage reduction in logic circuits. In this technique, an large nmos transistor is introduced in the ground path of each cache line. The gating transistor is turned on when the corresponding cache line is active. If a cache line has not been accessed for some predetermined amount of time, the gating transistor is turned off. Because the data is lost when gating, decay cache can increase the miss rate if it disables the data that is accessed again later. Although the gated-Vdd technique can effectively eliminate leakage, inserting the gating transistor in the critical pull-down path inevitably increases the cache access latency. The *DRI cache* [3] employs gated-Vdd technique at larger granularity. Entire subarray that contains several sets can be disabled at a time. Applying gated-Vdd at subarray level enables more degree of leakage reduction.

The *selective cache ways* [5] has originated the idea of adapting number of cache ways to runtime cache requirements. The work focus on saving dynamic power consumption. Little modifications to the conventional SRAM are added to prevent precharges, wordlines and sen-amps of disabled subarrays from switching. Our work extends the idea to support leakage reduction. The details of SRAM structure as well as an effective algorithms controlling number of enabled cache ways is described in the next section.

3. Way-variable Caches

3.1 Concept and Structure

In modern designs, large on-chip caches are often partitioned into multiple subarrays. The subdivision of wordlines and/or bitlines in such designs helps reduce the access latency. This is especially useful for associative caches for which the wordlines can be exceeding long. There are tools, when provided which parameters such as cache size, associativity, number of read/write ports, etc, analyze and choose the partitioning configuration that offers optimal performance [11]. By carrying experiments using such tools with various parameters as well as consulting real cache designs in recent commercial products, it has been verified that the number of time the wordline segmented is usually larger than the cache associativity [5]. It is, therefore, reasonable to assume the associative cache structure in which each cache way locates to seperate subarrays. Each subarray has its own decoder, precharger, sense-amplifier circuits so the subarrays can be controlled independently from each others.

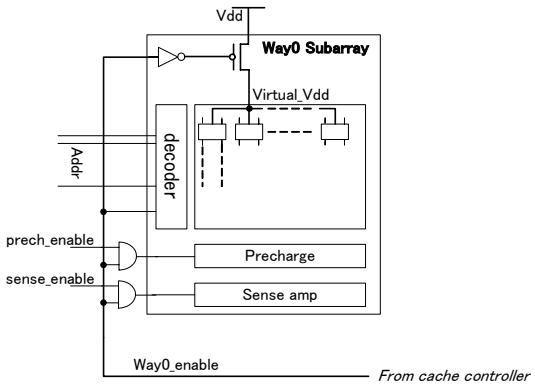

Fig. 2 The structure of an subarray

Figure 2 shows the overall structure of an subarray of the way-variable cache. This subarray is assummed to belong to way0 (it could be data or tag portion of the way). There is *way0_enable* signal that goes to the decoder, precharge and sense-amp of the subarray. A large pmos transistor connects the voltage supply Vdd to the cell array and also controlled by *way0_enable* signal. This transistor serves the functioning of gated-Vdd transistor. The cache controller manages the current states of ways and signifies *way0_enable* HI or LOW depending on whether way0 is enabled or disabled.

The circuit organization described here is quite similar to the one found in [5]. Because the purpose is to reduce dynamic power consumption in [5], gating the decoder, precharge and sense-amp way to avoid any switching activity in the disabled way is sufficient to that purpose. Our originality is that we add the gating transistor in order to be able to separate the cells in disabled way from supply voltage to reduce the leakage.

When the way0 is disabled, the gated-Vdd transistor is turned off, gating all cells in the subarray from the supply voltage. The internal nodes of each cell (node A, B in Figure 1-b) will soon drop to some intermediate voltage levels. Because leakage current highly depends on voltage level, such reduction in voltage level results in significant reduction of the leakage inside the cells. Moreover, the precharge circuits do not operate the disabled ways, causing the bitlines dropping from high precharged voltage level. This helps

reduce the leakage currents coming from bitlines.

When applying gated-Vdd techniques in SRAM, we have two possible choices: using pmos transistors to gate the Vdd paths or using the nmos transistors to gate the ground paths. The *decay cache* needs to insert nmos transistors in the ground paths to suppress the leakage current coming from the bitlines. In SRAM access, the pull-down paths, rather than pull-up paths, is considered the critical paths. The gating transistor sitting in ground path inevitably adds some amount of resistance to the pull-down path and consequently degrades the cache access latency. In our way-variable cache, because the bitlines of disabled ways eventually drop to low voltage levels, the amount of leakage coming from bitlines is small and can be dismissed. Adding gating pmos in Vdd path to suppress leakage inside the cell is sufficient and avoids the access latency degradation. Moreover, it is known that gate leakage in pmos has one order of magnitude smaller than that of nmos of the same size. Using pmos as gating transistor results in smaller *active* leakage (active leakage is the leakage exhibited by enabled ways or cells).

When a way is turned off, all the data in that way will be destroyed. We must ensure that any modified data is saved before turning the way off. The mechanisms to maintain the data integrity may incur overheads in terms of clock cycles as well as additional supporting hardwares. In [5], authors advocates the use of *read-and-refill* operation to move modified data to the other ways. It costs no additional hardware. We can also apply the idea of eager-writeback [9] that try to write modified data back whenever the bus to lower cache is free. Such a technique can help increase the bus utilization as well as reduce the overhead of writing back modified data when the way is going down. In this research, we limit the application of way-variable cache to L1 caches. We assume *write-through* data caches which are common in recent commercial processors (IBM Power4, Intel Itanium, etc,). However, when considering the applications of way-variable caches to write-back caches or low lever caches, the techniques mentioned above can help us to reduce the overheads.

3.2 Resizing Algorithm

As described in previous section, the disabled cache ways can exhibit very small leakage. For way-variable caches to be successful, we need to develop an algorithm that can properly determine when and to which direction (enable or disable more ways) the resizing will be taken.

Set associative caches usually employ the Least-Recently-Used (LRU) algorithm to choose the candidate block whenever replacement is required. The replaced block is the one that has been unused for the longest time. LRU algorithm is implemented by providing some state bits in each cache set. Accesses to the cache set will update the corresponding state bits. The LRU block can be identified by the contents of these bits.

We propose a resizing algorithm that takes advantage of these bits. Whenever an access comes to the cache, the state bits of the corresponding set are also read. If the access is a cache hit, by examining the contents of state bits we can determine whether the hitting block is the LRU block or not. We count the number of accesses to LRU blocks ($N_{LRU\_accesses}$) over the total number of cache accesses($N_{total\_accesses}$) over specific execution window.

The value of $N_{LRU\_accesses}$ allows us to make some predictions about how the cache missrate will vary if we enable or disable more ways. Specifically, as we decrease the number of disabled ways by one in the next execution window, we expect the cache missrate will be increased by about $\frac{N_{LRU\_accesses}}{N_{total\_accesses}} \times 100$ percent. The prediction is made on the assumption that the characteristics of cache accesses remain relatively stable over the period that quite longer than the length of single execution window. The same access that hit an LRU block in the previous window otherwise likely experiences a miss on the following windows (the block has already been evicted due to fewer enabled ways). Large $N_{LRU\_accesses}$ consequently results in more cache misses, and in turn, more performance degradation when we reduce the enabled cache ways. Conversely, if $N_{LRU\_accesses}$ is small, the benefit of allocating more cache ways to exploit temporal locality of the data becomes small.

```
//collect N_LRU_accesses N_total_accesses for the current window
if( N_LRU_accesses / N_total_accesses > threshold1 && nway_current_window < nway_MAX )
    nway_next_window = nway_current_window + 1;
else if( N_LRU_accesses / N_total_accesses < threshold2 && nway_current_window > 1 )
    nway_next_window = nway_current_window - 1;
else
    nway_next_window = nway_current_window;
```

Fig. 3 Resizing algorithm

The proposed resizing algorithm is shown in Figure 3. $N_{LRU\_accesses}$, $N_{total\_accesses}$ are collected for each execution window. We then calculate $\frac{N_{LRU\_accesses}}{N_{total\_accesses}}$. If the ratio is smaller than

a predefined value *threshold1* and the number of enabled ways is more than one, we decrease the number of enabled ways by one. If the ratio is larger than *threshold2* (*threshold1*<*threshold2*), we increase number of enabled ways by one. Otherwise, the number of enabled ways is unchanged. The *threshold1* and *threshold2* parameters control the "aggressiveness" of the resizing algorithm. Small *threshold1* and large *threshold2* make the algorithm "conservative" in the sense that they bias the upsizing direction and enable many cache ways. On the other hand, algorithms with large *threshold1* and small *threshold2* have the strong tendency of trading performance for reduction of more ways. Due to its simplicity, the proposed algorithm could be implemented with very modest hardware resource.

4. Evaluation

4.1 Leakage Evaluation

4.1.1 Evaluation Method

We carried out Spice simulation to measure the amount of leakage exhibited by individual SRAM cells in various cache configurations: 1)normal cache without any leakage consideration, 2) decay cache, 3)drowsy cache and 4)way-variable cache. Transistor parameters at 65nm process technology, obtaining from Berkeley Predictive Technology Model [7], was used for the evaluation. Transistor threshold voltage is `|Vth|`=0.2V. Vdd is set at 1V. In case of drowsy cache, voltage supply at drowsy mode is 0.5V. Detail circuit setups using in the evaluation are shown in Figure 4. Transistor model level 51, which considers both subthreshold leakage and gate leakage, is used.

4.1.2 Result

We measure the total leakage current of the cells when they are in active state and when they are putted in low leakage mode (or standby mode). The results are shown in Figure 5-a and 5-b, respectively. The leakage in the figure is further broken down into subthreshold leakage and gate leakage.

Because no leakage reduction measure is taken, normal cache cell has the same value of leakage current in Figure 5-a and 5-b. The active leakage values are almost equal (95 nA) for all cells except the decay cache cell which is 20 nA larger. The excessive leakage in decay cache cell is caused by the gating nmos transistor. This transistor is on when the cell is enabled and the gate leakage of nmos in on state is significant. The way-variable cache cell, which uses pmos instead of nmos gating transistor, shows almost no increase in active leakage. The result is consistent with the fact that p-type transistors have one degree of magnitude lower gate leakage than n-type transistor [10].

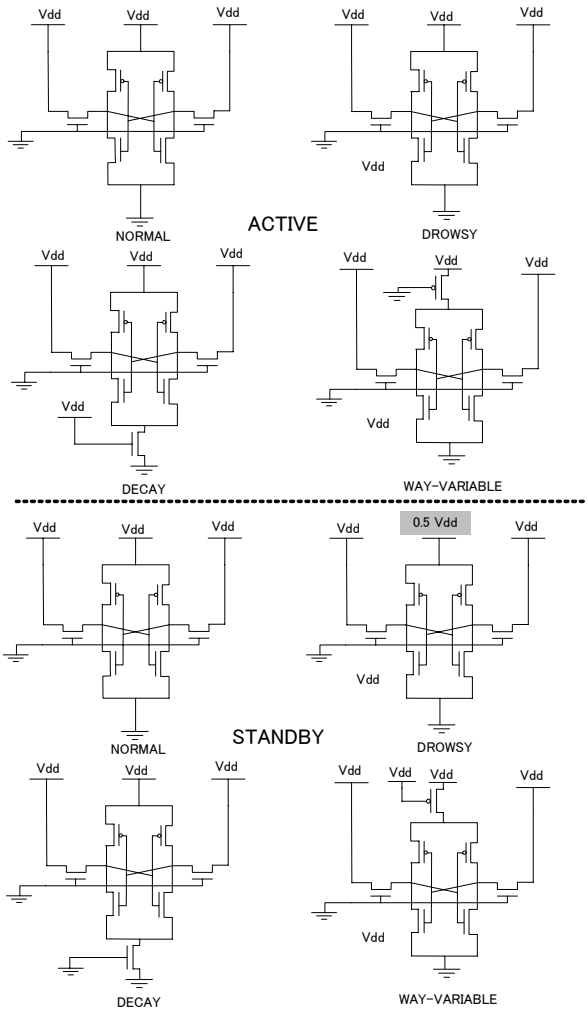

Fig. 4 Circuit Setups for Evaluation

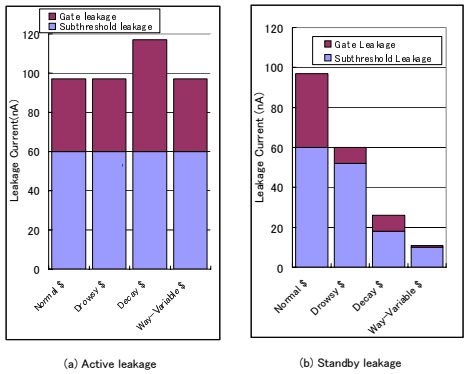

Fig. 5 Leakage in active mode and standby mode

The standby leakage of drowsy cache cell is 60 nA. While gate leakage is reduced much when Vdd changing from 1V to 0.5V in drowsy cache cell, subthreshold leakage is not changed much. It is because the bitlines are still kept at 1V and the subthreshold leakage current from bit-

line are almost unchanged. The decay cache cell, due to its capability of suppressing the leakage paths from bitlines, offers higher degree of standby leakage reduction than the drowsy cache cell. The standby leakage in this case is 27 nA. Way-variable cache cell shows the best performance. The standby leakage is reduced to only 11 nA, which is tenth times smaller than the leakage of the normal cell.

The way-variable cache is good because it offer us lowest standby leakage. Additionally, contrary to the decay cache, implementation of way-variable cache does not incur the increase in active leakage of the cell.

4.2 Performance Evaluation

4.2.1 Evaluation Method

We carried out simulations to investigate the impacts of variations in cache ways on the processor performances. Simulations were done using SimpleScalar simulator [6]. We modeled an 6-stage superscalar processor, having 16KB 4-way set associative I-cache and D-cache and 256KB 4-way L2 cache. L1 cache hit, miss latency are 1 and 8 clock cycle, respectively. Simulation were done against applications in the SPECCPU2000 benchmarks with the *reference* inputs. We fast-forwarded first one billion instructions and simulated the next four billion instructions.

4.2.2 Results

Performance Degradation. Figure 6 and Figure 7 show the performance degradation incurred for each benchmark as the number of enabled ways of the instruction cache and data cache varied. The amount performance degradation quite different among applications. In Figure 6, performance degradations of applications like *mcf, ammp, swim* are very small. We can execute these applications from the beginning to the end using the instruction caches with minimum number of enabled ways.

On the other hands, performances of *crafty, eon, vortex, equake ,mesa* are very sensitive to the number of enabled ways of instruction caches. Performances tend to degrade abruptly as we disable more caches. For way-variable caches still being useful in such situations, we need dynamic approaches that monitor variations of cache requirements during the lifetimes of applications and allocate appropriate number of ways accordingly. Such observation can also be obtained in the case of the data caches in Figure 7. On average, performance is more sensitive to instruction cache than the data cache.

Resizing Algorithm Evaluation. We implemented our proposed resizing algorithm into the SimpleScalar simulator. Application executions are divided into multiple windows. Each execution window comprises of hundred thousands of consecutive dynamic instructions. *Threshold1* and *threshold2* parameters are respectively set at 1% and 2% for the way-variable data cache. As we observed above, the instruction cache are more sensitive to cache size than the data cache. To make resizing algorithm of instruction cache to be less aggressive than that of data cache, its *threshold1* and *threshold2* parameters is set at 0.5% and 1%, respectively.

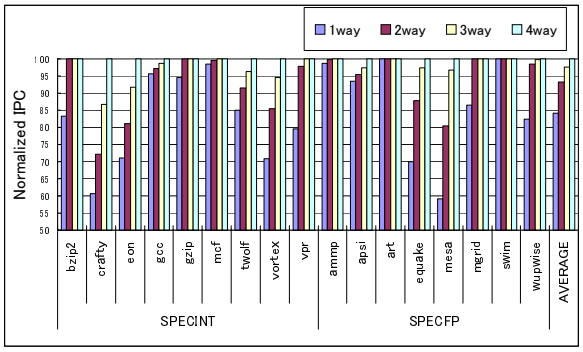

Fig. 6 Performance degradation when number of enabled ways of instruction cache varying from one to four

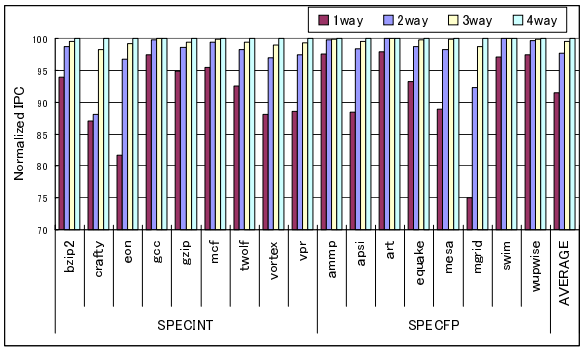

Fig. 7 Performance degradation when number of enabled ways of data cache varying from one to four

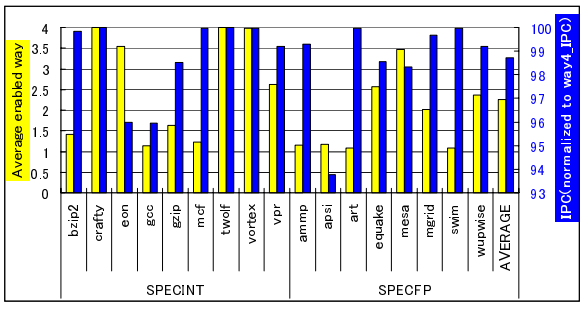

Fig. 8 Way-variable instruction cache: average active way & performance degradation

Figures 8, 9 show the average number of enabled ways as well as normalized IPC degradation of the way-variable instruction cache and way-

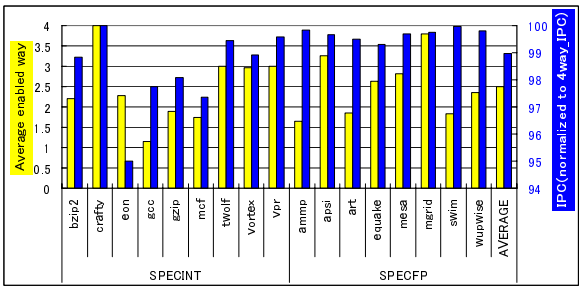

Fig. 9 Way-variable data cache: average active way & performance degradation

variable data cache. With our resizing algorithm, applications are executed with quite reasonable values of enabled ways. Applications whose performances are not sensitive much to cache sizes are executed with minimum number of enabled ways. They are *gcc, mcf, ammp, art, swim* in case of the instruction cache, as well as *gcc, ammp, swim, wupwise* in case of the data cache. For those application whose performances are very sensitive to cache size, almost full cache ways are enabled during the execution lifetimes. They are *crafty, twolf, vortex* in case of the instruction cache and *crafty, mgrid* in case of the data cache.

Let us consider the *bzip2* application in case of the instruction cache. Referring to Figure 6, performance degradation of *bzip2* is negligible when we disable one or two ways. However, performance quickly drops by 15% when only one way are enabled. The result in Figure 8 shows that we can disable 2.6 ways while paying only 0.7% performance degradation. During the execution lifetime, there must be some periods in which two ways are enabled and some periods in which only one way is enabled. Such result implies that our resizing algorithm has been able to effectively adapt the cache ways to runtime cache requirements during execution of *bzip2*. Similar results can also be observed in *gzip, vpr, equake, wupwise* in case of the instruction cache and in most of applications in case of the data cache.

On average, we can disable 1.7 ways with only 1.3% performance degradation in the case of instruction cache. The values are 1.5 ways and 1.1% in the case of the data cache.

5. Conclusion

Power consumption due to leakage increases rapidly as devices scale to smaller geometries. Our research focus on reducing the leakage in microprocessor caches. We propose way-variable caches that dynamically changing the number of active ways according to the runtime requirements. By entirely gating the unused ways from the voltage supply, the leakage can be significantly reduced. We then apply an original algorithm utilizing data access locality to make proper resizing decisions. We verified that, on average, nearly half of the cache ways of 4-way set associative instruction as well as data cache can be disabled with very limit performance degradation.

So far, leakage reduction in way-variable caches has been only considered at the level of the SRAM cells. We plan to investigate the effectiveness of way-variable caches in terms of total power consumption could be reduced at the system level.

Acknowledgement

This research is partially supported by Grant-in-Aid for Fundamental Scientific Research B(2) #13480077 from Ministry of Education, Culture, Sports, Science and Technology Japan, and by Semiconductor Technology Academic Research Center (STARC) Japan, CREST project of Japan Science and Technology Corporation, and by 21st century COE project of Japan Society for the Promotion of Science.

References

[1] Jason Stinson, Stefan Rusu, "1.5 Ghz Third generation Itanium Processor", *ISSCC 2003*

[2] Krisztian Flautner, Nam Sung Kim, et,al., "Drowsy caches: simple techniques for reducing leakage power, *ISCA 2002*

[3] Michael Powell, Se-Hyun Yang, et,al., "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memory", *ISLPED 2000*

[4] Zhigan Hu, Stefanos Kaxiras, Margaret Martonosi, "Let Caches Decay: Reducing Leakage Energy via Exploitation of Cache Generational Behavior", *ISCA 2001*

[5] David H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", *MICRO 1999*

[6] D. Burger and T. Austin, "The Simple-Scalar Tool Set", *University of Wisconsin-Madison Computer Sciences Departmen Technical Report*

[7] Berkeley Predictive Technology Model, http://www-device.eecs.berkeley.edu/ ptm/

[8] Y. Ye, S. Borkar, and V. De, "A technique for standby leakage reduction in high-performance circuits, *Symposium on VLSI Circuits, 1998*

[9] Hsien-Hsin Lee, Gary Tyson, Matthew Farrents, "Eager Writeback a Technique for Improving Bandwidth Utilization", *MICRO 2000*

[10] Fatih Hamzaoglu, Mircea R. Stan, "Circuit-level techniques to control gate leakage for sub-100nm CMOS", *ISLPED 2002*

[11] Glen Reinman and Norman P. Jouppi, "CACTI 2.0: An Integrated Cache Timing and Power Model",In *WRL Research Report 2000/7, DEC Wester Research Laboratory*, 2000