Low Static-Power Frequent-Value Data Caches

Chuanjun Zhang*, Jun Yang and Frank Vahid
Department of Computer Science and Engineering
*Department of Electrical Engineering
University of California, Riverside
czhang@ee.ucr.edu, {junyang/vahid}@cs.ucr.edu,
F. Vahid is also with the Center for Embedded Computer Systems at UC Irvine

Abstract: *Static energy dissipation in cache memories will constitute an increasingly larger portion of total microprocessor energy dissipation due to nanoscale technology characteristics and the large size of on-chip caches. We propose to reduce the static energy dissipation of an on-chip data cache by taking advantage of the frequent values (FV) that widely exist in a data cache memory. The original FV-based low-power cache design aimed at only reducing dynamic power, at the cost of a 5% slowdown. We propose a better design that reduces both static and dynamic cache power, and that uses a circuit design that eliminates performance overhead. A designer can utilize our architecture by simulating an application and then synthesizing the FVs into an application-specific FV cache design when values will not change, or by simulating and then writing to an FV-cache with configuration registers when values could change. Furthermore, we describe hardware that can dynamically determine FVs and write to the configuration registers completely transparently. Experiments on 11 Spec 2000 benchmarks show that, in addition to the dynamic power savings, 33% static energy savings for data caches can be achieved.*

1. Introduction

As the speed gap between processors and memories continues to increase, modern processors tend to include larger on-chip caches to reduce the number of accesses to long latency memories. For this reason, on-chip caches remain a major chip power consumer. For example, the Intel Pentium Pro dissipates 33% [5] and the StrongARM-110 dissipates 42% [12] of its total power in caches. Moreover, as the number of on-chip transistors continues to grow and the threshold voltage continues to decrease in nanoscale technologies, leakage power (i.e., static power) accounts for a larger portion of total power consumption.

As a result, numerous techniques have been proposed to conserve cache memory power. To reduce dynamic energy dissipation, techniques including filtering mechanisms[10], on-demand reconfiguration [14][18], way selection and prediction [2][8], cache decomposition [6], and value based low cache activity [16] have been proposed. At the same time, many techniques have also appeared to reduce cache static power consumption [1][2][9][11][13]. Most of the proposed techniques are based on turning off portions of the cache at the cost of losing data and increasing miss rates. Even techniques that try to control this negative effect incur performance overhead to a certain extent [9]. In this paper, we propose a technique that can turn off cache portions with zero information loss and no performance degradation with proper design of cache cell circuitry.

Recently, a frequent value low power data cache design was proposed based on the observation that a major portion of data cache accesses involves frequent values that can be dynamically captured [16]. Frequent values are stored in cache in an encoded form occupying only a few bits. For example, 32-bit frequent values, say "FFFFFFFF" and "FFFF0000" in hex, might be encoded using 5 bits as "00000" and "00001" respectively. The cache data array bank is separated into two sub-arrays such that all the frequent value encodings are located in a smaller sub-array. Non-frequent values occupy both the smaller and larger sub-arrays. On each frequent value access, only the smaller sub-array is activated, saving dynamic power by not driving the larger sub-array.

In this paper, we propose to reduce static power by shutting off the unused bits in the larger sub-array for encoded frequent values. Since frequent values are stored in encoded form using only the few bits in the smaller sub-array, the remaining bits in the larger sub-array serve no purpose as long as the value stays frequent. Such shutoff may be very beneficial since FVs occupy many words in data caches [16].

Furthermore, the original FV low power cache design suffers from an extra cycle when reading non-FVs [16], which account for 68% of all data cache accesses, resulting in a 5% increase in execution time. We were able to use circuit design to remove the extra cycle.

Our low power FV cache can be designed in various ways. For application-specific processors, the FVs can be first identified offline through profiling, and then synthesized into the cache so that power consumption is optimized for the hard coded FVs. For processors that run multiple applications, the FVs can be located in a register file to which different applications can write a different set of FVs. To obtain maximum flexibility, a third approach is to embed the process of identifying and updating FVs into registers, so

that the design dynamically and transparently adapts to different workloads with different inputs automatically.

The rest of the paper is organized as the following. In Section 2, we briefly summarize existing static power reduction techniques. In Section 3 and Section 4, we describe the low static power FV data cache techniques and its design choices. In Section 5, we present experimental results, and we conclude the paper in Section 6.

2. Previous Work

There have been a number of techniques proposed to reduce cache static power consumption. Gated-V_{dd} [13] inserts an extra transistor between the voltage source and the SRAM cells to shut off the unused on-chip cache lines, achieving 62% energy-delay product reductions. Gated-V_{dd} has been widely used by many researchers to shutoff part of the cache. The DRG [1] cache dynamically resizes itself by monitoring the miss rate of the cache and shutting off part of the sets according to the application's dynamic behavior. Cache line decay [11] dynamically turns off cache lines that have not been visited for a designated period, reducing the L1 cache leakage energy dissipation by 4x in SPEC2000 applications. Zhou [19] proposed to dynamically determine the time interval for deactivating cache lines, achieving an average of 73% instruction cache lines and 54% of data cache lines being put into sleep mode, with an average IPC impact of 1.7%.

In contrast to shutoff mechanisms that lose the information in the cache, a drowsy cache [9] keeps the unused cache line in a low power mode by lowering the SRAM source voltage while retaining the contents of the cache. 80% to 90% of the cache lines can be put into drowsy mode without affecting the performance by more than 1% for SPEC2000.

Notice that all previous works increased the program execution time, mostly due to higher level-one cache misses. As a result, more power is consumed in the higher-level caches or memories. The significance of our proposed technique is that no additional cache misses are introduced, and neither the core nor the higher-level cache memories are affected. In other words, we reduced the cache energy consumption (dynamic + static) without slowing down the processor. Furthermore, our technique can be also applied jointly with other methods to achieve further power savings.

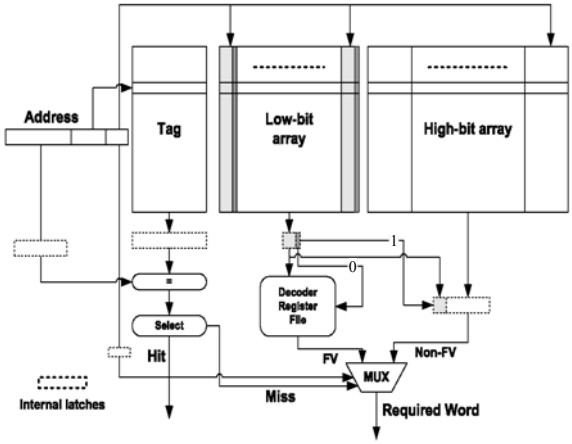

Figure 1: Original frequent value data cache architecture [16].

3. FV Data Cache Design

3.1 Overview of Original FV Cache Design

In this section, we give a brief overview of the original FV data cache design. More details can be found in [16].

The FV cache was proposed based on the observation that a small number of distinct frequently occurring data values often occupy a large portion of program memory data spaces and therefore account for a large portion of memory accesses [16]. This frequent value phenomenon was exploited in designing a data cache that trades off performance with energy efficiency.

From the perspective of the frequent value cache, data values are divided into two categories: a small number of frequent values, in our case 32 FVs, and all remaining values that are referred to as non-frequent values. The frequent values are stored in encoded form and therefore can be represented in 5 bits; the non-frequent values are stored in unencoded form in 32 bit words. The set of frequent values remains fixed for a given program run.

The cache data array is partitioned into two data arrays as shown in Figure 1. The low-bit array contains the lower order 5 bits of each word and the high-order array contains the remaining 27 bits. Frequent values are stored in encoded form in the low-bit array while non-frequent values are split into the lower order 5 bits and the higher order 27 bits using the space in both arrays. The 5-bit encoding is used to retrieve the full 32-bit value from the "Decoder Register File" in the figure for frequent values. To distinguish between a code for a frequent value and a trailing part for a non-frequent value in the low-bit array, an additional flag bit is needed corresponding to each word in a cache line. The extra bit was stored along with every word in the low-bit array so that the word width becomes 6 bits.

When reading a word from the cache, initially we simply read from the low-bit array. Since every word read out contains a flag bit, the flag is examined to determine what comes next. The flag being 1 means the desired word is in un-encoded form, so the remaining bits should be read out from the high-bit array to form the original value. On the other hand, the flag being 0 means that the desired word is a frequent value and stored in encoded form. In this case, the access proceeds to decode the value. Since the access to the high-bit array is avoided, cache activity is reduced.

A write to the FV cache is performed as follows. Before a value is written, it is first encoded through an encoder. If

encoding is successful, it means that the value is a frequent value and thus a 5-bit code is stored in the low-bit array and the flag bit is cleared. In this case, accessing the high-bit array is avoided. If the encoding fails, the value to be written is a non-frequent value and thus both low-bit and high-bit data arrays are accessed as well as the flag bit being set. Note that writing non-FVs does not need to take two cycles as does reading non-FVs, because the value is encoded early in the pipeline and thus the decision of driving one array or two is clear before the access.

In a traditional cache implementation, the entire line is first read out from the data array, and then the desired word within the line is selected at the time it reaches the output multiplexor. If the same scheme is used in the FV cache design, the decoding of frequent values cannot begin until the required word is selected out, which is the very end of a cache access. Consequently, decoding will increase the cache access time, which is not desirable. Therefore, the subbanking scheme proposed by [4] was adopted, in which the subbank containing the target word can be read independently. The width of each subbank is the physical word width and each subbank can be activated independently. This design facilitates the FV cache implementation in that the decoding for frequent values can begin immediately after accessing the low-bit array since only the word at the desired location is read out instead of the whole line.

3.2 Shutting Off Unused Bits of FV

The FVs are not only *accessed* frequently, but also *distributed* widely in caches [17]. This phenomenon provides a good opportunity for reducing static power. Our approach is the following. Since the 32-bit FVs are encoded in 5 bits, the remaining 27 bits do not store any useful information. Therefore, they can be shut down to save static power. This shutdown is performed as soon as a coded FV is written into the cache through write or refill operations. The unused bits will stay off until the word is turned into a non-FV. At that time, all the off bits are awakened to function as in a normal cache. Thus, as long as a value stays frequent, static power is saved. The overall savings depend on the occupancy of FVs in the cache. Our studies show that on average nearly half of the cache content contains FVs (Section 4), which indicates the benefit of reducing static power through finding FVs.

We used a "Gated-V_{dd}" technique [13] to control the open and close of a cell. Either a gated pMOS or nMOS transistor can be used to gate the voltage, with different tradeoffs among delay, area, and static power savings. An nMOS gated-V_{dd} can eliminate 97% of static power with 5% area increase and 8% access time increase. A pMOS gated-V_{dd} has almost no area overhead and no delay increase, but the tradeoff is that the static power savings is reduced to 86% [13]. Our goal is to minimize

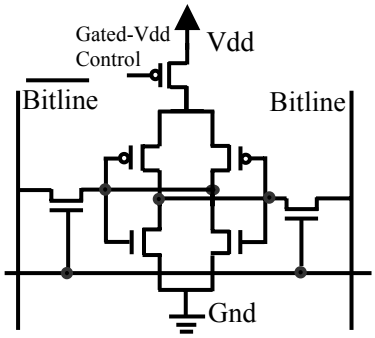

Figure 2: SRAM cell with a pMOS gated Vdd control.

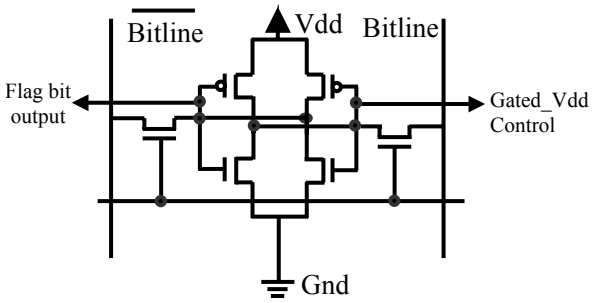

Figure 3: Flag bit SRAM cell

static power for frequent values without impacting the performance. Thus, a pMOS gated-V_{dd} transistor is more suitable for us. The nMOS cell can still be used in our FV cache when maximum static power saving is desired and when the 8% increase time is acceptable, especially when the 8% does not extend the system's critical path.

Figure 2 shows the cell shutdown logic. An extra pMOS transistor is integrated into the original SRAM cell. When the "Gated-V_{dd} Control" goes high, the SRAM cell's voltage is floated, turning off the entire cell.

The impact of the extra pMOS transistor on area is tiny. Since the 5-bit array will never be shut off, we only need to consider the 27-bit array. Further, we only need to add one extra pMOS transistor for an entire block of 27 bits for each cache word. We measured the extra area for a 32 Kbyte, four-way 32-byte line size cache. The total area overhead obtained from the layout [18] of our cache is less than 1% of the total data cache area.

The gated-V_{dd} control signal of the pMOS transistor, "Gated_V_{dd} Control" as shown in Figure 3, is the inversion of the flag bit's output that indicates whether or not the cache word is an FV. The flag bits are initially set to 1, which means initially all words are non-FVs. Any data to the data cache is checked with the FV encoder. If the word is an FV, the corresponding flag bit is set to 0 and this cache word is encoded and stored in the 5-bit array. At the same time, the flag bit turns off the 27-bit portion of the word. Similarly, on reading FVs, only the 5-bit portion is read and the 27-bit portion is gated off. On a non-FV read or write, the flag bit is set to one and the original 32 bits are written into the cache as usual. Our new circuit design improves the original FV cache design in that there is no extra delay in determining

accesses of the 27-bit portion. This is explained in more detail next.

3.3 Non-Delayed FV Cache Design

Reading non-FVs in the initial FV cache design incurs extra cycles since the flag bits are clustered as an array, and reading out its corresponding flag bit precedes reading the 27-bit portion. This feature increases the average L1 data cache access latency and slows down the program execution time. To gain true energy savings, cache accesses due to FVs should well exceed non-FVs. However, not all applications we analyzed held this property favorably. As shown in [16], some benchmarks consume more energy than the base case (e.g. 107.mgrid in Spec95 [16]), mitigating the general applicability of the initial FV cache design.

3.3.1 Design of the flag bit

While it is not obvious how to increase FV access locality, revising the circuit design to eliminate the necessity of extra cycles for non-FV is feasible. We observed that the management of the flag bit is the critical point in the design. Note that the flag bit is stored in a single SRAM cell, as shown in Figure 3. After the bit value is written into the SRAM cell, its content is stable as long as power is supplied. Therefore, the operation of reading out the SRAM cell bit is unnecessary since this bit can be a stand-alone cell attached to each word. Since every word is split into 5 bits and 27 bits, the flag bit can be used to gate the open/close of the 27 bits' drivers. The design of the flag bit plays an important role in both word portion shut-off and non-delayed access.

To see the feasibility of our flag bit design, we laid out the flag cell together with the new driver, and we also extracted the circuits. Since the reading of the flag bit is not performed, we care only about its write operation. Our SPICE simulation verified that writing to the flag bit would work properly without modifying the transistors' parameters of the flag bit. The dimension of our flag bit is 2.8um×4.8um in 0.18um technology.

3.3.2 Cache Line Structure and Timing Analysis

The conventional cache line architecture is shown in Figure 4. The output of the index decoder is strengthened by a word line driver that is composed of two inverters as illustrated in Figure 5(a)[15]. The new cache line architecture is shown in Figure 6 (only four cache words are drawn). Here we employed the new word line driver shown in Figure 5(b). A NAND gate replaces the original

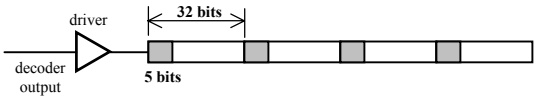

Figure 4: The conventional cache line architecture, four words per cache line are drawn.

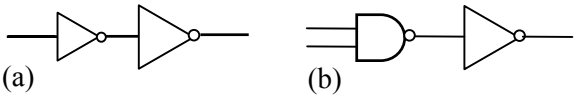

Figure 5**:** Design of the original and new word line driver

inverter. The content of the flag bit and the output of the index driver are the two inputs of the NAND gate, as shown in Figure 6. A flag bit of '0' indicates an FV, and '1' a non-FV. Note that the decoder will still drive all the sub-data array banks. The partial words with '0' flag bits are automatically gated off and those with '1' flag bits will be turned on or off based on the index decoder output.

Changing the inverter into a NAND gate could increase the wordline's driving delay since a NAND gate contains more transistors than an inverter. However, tuning the transistor size can reduce the NAND delay to that of the original inverter. We performed SPICE simulations to compare the delay before and after resizing. Our results show that if transistor sizes are maintained the same, the total increase to the cache critical path is 2%. Fortunately, this increase can be avoided if the NAND gate transistor size is tripled, which does not represent a significant overall area increase since those transistors didn't occupy much area to begin with. Thus, we can maintain the same cache access delay with the new cache line driver.

4. Designers' Choices of Using the FV Cache

We have described a low static power FV cache. When utilized into a processor system, the FV cache can be designed with different degrees of complexity and flexibility. In this section, we provide three approaches that are suitable for a variety of processors targeting different types of applications. Essentially, the complexity comes from how FVs are identified and if they are allowed to vary for different applications. As always, the more flexibility the processor provides, the more complex the FV cache is.

The first approach is appropriate to application specific processors. Since only a single type of application runs on the processor, its FVs tend to be stable over time. In such cases, the FVs can be first obtained from a profiling run through simulations, and then synthesized into the cache as part of the cache data storage. The advantage of this approach is that once the FVs are hard coded on-chip, the

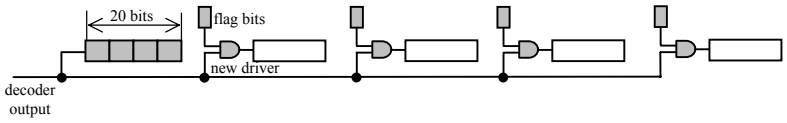

Figure 6: FV cache line architecture. Flag bit equals '0' means the corresponding cache word is an FV. The leftmost five bits are used to encode the FVs.

cache does not perform operations other than reads. Thus, the logic of this component is simple and can be designed to consume minimum power.

The second approach extends the first one with the ability of changing the FVs according to different applications. This approach is suitable for a multi-task environment in which the processor runs multiple programs instead of single program. Since different program exhibits different behavior, and thus generates different FVs, it is necessary to let FVs change when a different program is activated. Each program's FVs are still obtained off-line. Instead of synthesizing the FVs on-chip, a register file may be used to store FVs so that they can be rewritten on each activation of a different program. The size of the register file depends on the number of FVs of interest to the designer, which is heavily dependent on each program's behavior. A study showed that for SPEC95 benchmark suite, 32 FVs are generally satisfactory across different programs [17]. This will account for 3.9% of area overhead (obtained from CACTI3.0 [15]) if a 32 Kbyte L1 D-cache is used (as in Section 5).

The third approach provides the maximum flexibility in maintaining FVs. According to a previous study [17], some programs' FVs are sensitive to different inputs. This suggests that another dimension of varying FVs might be added into the design. Since it is infeasible to profile every program on all possible inputs to catch FVs, detecting FVs on-line would be useful. Thus, on top of the second approach, the register file could be extended to dynamically capture FVs using extra logic. In the scheme proposed in [16], an inexpensive hardware FV finder was developed that monitored cache accesses. The FV finder was turned on for only the first 5% of memory accesses assuming that the total memory access numbers are known a priori. After that, the FVs were captured in the finder and transmitted to the cache so that the cache starts operating as an FV cache. The energy overhead of the finder was estimated to be 0.3%-6.1% of the L1 D-cache (8 Kbyte to 64 Kbyte caches were tested). The area overhead is similar to our second approach, and thus modest. One potential issue is that the FV finder described detects frequently *accessed* values, which may or may not correspond to frequently *distributed* values in memory, though they usually are the same. We leave an FV finder for frequently distributed values for future work.

5. Experiments

To determine the benefits of our FV cache architecture in reducing static energy, we ran 11 SPEC2000 [7] benchmarks, *art, ammp, parser, mcf, equake, gcc, gzip, bzip, mesa, votex, and vpr,* through the SimpleScalar tool set [3]. We used a 4-issue out-of-order processor simulator with a 32 Kbyte L1 instruction and data cache. The benchmarks were fast-forwarded for 1 billion instructions and executed for 500 million instructions afterwards, using reference inputs. The configuration of the simulated microprocessor is shown in **Table 1**.

Table 1: Configuration of the simulated processor.

Processor Core	
Instruction Window	80-RUU,40-LSQ
Issue width	4 instructions per cycle
Memory Hierarchy	
L1 Dcache	32KB, four way, 32B line size, WB
L1 Icache	32KB, four way, 32B line size, WB
L2 unified cache	128KB,four way, 64B line size, WB
Memory	100 cycles
TLB Size	128-entry, 30-cycle miss penalty.

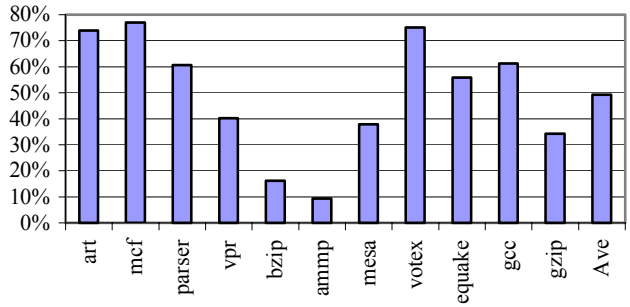

Figure 7: Percentage of data cache words that are FVs.

5.1 Static Energy Savings

Our main goal is to reduce the static energy consumed by the data cache without losing performance. As mentioned earlier, the overall static energy saving depends on the average coverage of FVs inside data cache. Through experiments, we found that there are abundant FVs in the L1 data cache at any time for Spec 2000 benchmarks, as shown in Figure 7. The percentage shown is the average for the 500 million instructions execution time. On average, 49.2% of the total words are FVs, with the highest being 77.0% for benchmark *mcf* and the lowest 9.4% for benchmark *ammp*. The static energy savings are proportional to the number of FVs in the data cache. Thus, the corresponding static energy savings on average are 35% (49.2%×27/33×86%) considering that 27 bits out of 33 bits (we need a flag bit per 32-bit word) are shut off and 86% of static power can be saved using a pMOS Gated-V_{dd}. When compared with the conventional 32-bit per word cache, the static energy savings can be calculated as 100%- (100%-35%)*33/32 = 33%.

5.2 Performance Improvement

Our second achievement is the performance improvement over the original FV data cache design. Recall that the original FV cache performance overhead was due to the prolonged non-FV accesses. The more non-FV accesses, the slower the execution and the less the overall power savings (less energy savings), since the system would consume more energy when the program runs longer. We measured the

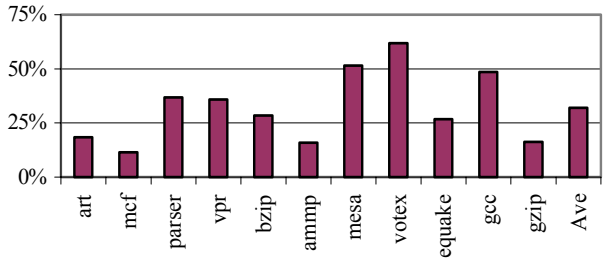

Figure 8: Hit rate of FVs in data cache.

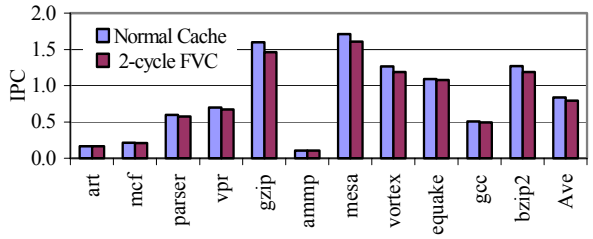

Figure 9:Performance (IPC) degradation of two-cycle FV cache.

average percentage of cache hits that are FVs, as shown in Figure 8. On average, the hit rate on data FVs is 32% with the highest being 62.7% for *votex* and the lowest 11.4% for *mcf*. Therefore, we can see that on average, 68% of cache accesses are non-FVs.

With our improved circuitry (1-cycle latency for non-FVs as well as for FVs), we are able to maintain the same execution speed as the base case. To see how much performance we have gained over the original FV cache, we measured the IPCs for a normal cache and a 2-cycle FV cache and plot them in Figure 9. The IPC for our improved design is the same as the normal cache. Figure 9 shows the slowdowns of the original FV cache design, which is the same value as our performance improvement. We can see that there is a 5.2% difference in the averaged IPCs between the original FV cache and our improved version. This also means that in addition to the static energy we saved by shutting off partial FV words, we also saved more dynamic energy than the original FV cache design.

Another feature in our new design is that it is *safe* in the sense that it does not increase power consumption significantly even when FVs are not abundant. Thus, our improved FV cache design is an appealing approach in reducing both static and dynamic energy of caches.

6. Conclusions

We proposed an efficient static power saving scheme based on the widely existing frequent values in data caches. The scheme reduces data cache static energy by over 33% on average. Our cache cell shut-off based scheme does not increase cache miss rates as other similar techniques do. We also successfully removed the performance overhead that was present in the original FV cache. Our scheme can be combined with other low-power cache methods for further power and energy savings.

7. References

[1] A. Agarwal, H. Li, and K. Roy, "DRG-Cache: A Data Retention Gated-Ground Cache for Low Power," Design Automation Conf., June 2002.

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

[3] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin-Madison Computer Sciences Dept. Technical Report #1342, June 1997.

[4] K. Ghose and M. B. Kamble, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation," IEEE/ACM Int. Symp. on Low Power Electronics and Design, 1999.

[5] S. Gunther and S. Rajgopal, Personal communication.

[6] M. Huang, J. Renau, S.M. Yoo, and J. Torrellas, "L1 Data Cache ecomposition for Energy Efficiency," Int. Symp. on Low Power Electronics and Design, 2001.

[7] http://www.specbench.org/osg/cpu2000/

[8] K. Inoue, T. Ishihara, and K. Murakami, "Way-Predictive Set-Associative Cache for High Performance and Low Energy Consumption," Int. Symp. On Low Power Electronics and Design, 1999.

[9] N. S. Kim, K. Flautner, D. Blaauw, T. Mudge, "Drowsy Instruction Caches, Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction," Int. Symp. on Microarchitecture, Nov. 2002.

[10] J. Kin, M. Gupta and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," Int. Symp. on Microarchitecture, pp. 184-193, Dec. 1997.

[11] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting General Behavior to Reduce Cache Leakage Power," Int. Symp. on Computer Architecture, 2001.

[12] J. Montenaro. et al., "A 160MHz 32b 0.5W CMOS RISC Microprocessor," Int. Solid-State Circuits Conf., 1996.

[13] M. Powell, S.H. Yang, B. Falsafi, K. Roy, and T.N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. Int. Symp. on Low Power Electronics and Design, 2000.

[14] P. Ranganathan, S. Adve, and N.P. Jouppi, "Reconfigurable Caches and their Application to Media Processing," Int. Symp. on Computer Architecture, 2000.

[15] P. Shivakumar, N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model", COMPAQ Western Research Lab, 2001.

[16] J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design," Int. Symp. on Microarchitecture, Nov. 2002.

[17] J. Yang, and R. Gupta, "Frequent Value Locality and its Applications," ACM Transactions on Embedded Computing Systems (inaugural issue), Vol. 1, No. 1, pages 79-105, November 2000.

[18] C. Zhang, F. Vahid, and W. Najjar "A Highly Configurable Cache Architecture for Embedded Systems," Int. Symp. on Computer Architecture, June 2003

[19] H. Zhou, M.C. Toburen, E. Rotenberg, T. M. Cont. Adaptive mode-control: A static-power-efficient cache design. In the 10th Intl. Conf. on Parallel Architectures and Compilation Techniques, 2000.