A Way-Halting Cache for Low-Energy High-Performance Systems

Chuanjun Zhang*, Frank Vahid**, Jun Yang and Walid Najjar
Department of Computer Science and Engineering
*Department of Electrical Engineering
University of California, Riverside
** Also with the Center for Embedded Computer Systems, UC Irvine
{chzhang/vahid/junyang/najjar}@cs.ucr.edu

Abstract: *Caches contribute to much of a microprocessor system's power and energy consumption. We have developed a new cache architecture, called a way-halting cache, that reduces energy while imposing no performance overhead. Our way-halting cache is a four-way set-associative cache that stores the four lowest-order bits of all ways' tags into a fully associative memory, which we call the halt tag array. The lookup in the halt tag array is done in parallel with, and is no slower than, the set-index decoding. The halt tag array pre-determines which tags cannot match due to their low-order four bits mismatching. Further accesses to ways with known mismatching tags are then halted, thus saving power. Our halt tag array has an additional feature of using static logic only, rather than dynamic logic used in highly associative caches. We provide data from experiments on 17 benchmarks drawn from MediaBench and Spec 2000, based on our layouts in 0.18 micron CMOS technology. On average, 55% savings of memory-access related energy were obtained over a conventional four-way set-associative cache. We show that energy savings are greater than previous methods, and nearly twice that of highly-associative caches, while imposing no performance overhead and only 2% cache area overhead.*

Categories and Subject Descriptions
B.3.2 [Memory Structure]: Cache memories
General Terms: Design
Keywords
Low power techniques, cache design

1. Introduction

Caches may consume nearly 50% of a microprocessor's power [12][16]. Cache designers, for both high-end and embedded processors, must compromise between performance, cost, size, and power/energy dissipation. A fundamental cache design tradeoff is between a direct-mapped cache and set-associative cache. A conventional direct-mapped cache accesses only one tag array and one data array per cache access, whereas a conventional four-way set-associative cache accesses four tags arrays and four data arrays per cache access. Thus, a conventional direct-mapped cache consumes much less dynamic power per access than a set-associative cache. However, a direct-mapped cache may have a higher miss rate than a set-associative cache, depending on the access pattern of the executing application, with a higher miss rate meaning more power consumed in off-chip accesses and stalled processor cycles. Therefore, a direct-mapped cache may or may not result in less overall power and/or energy consumption for a particular application.

Generally, we can view the low-dynamic-power cache design goal as that of minimizing the internal activity during a cache access. That activity comes from reading and comparing tags in tag arrays, and from reading/writing data in data arrays. Ideally, on a hit, we would have only read and compared one tag entry and accessed one data entry – we cannot do much better than that. Furthermore, on a miss, we would have only read and compared one tag entry, and accessed no data entries. In fact, on a miss, we do not even have to access a complete tag entry – seeing even one mismatched tag bit is enough to determine a miss. This last point provides the motivation for our way-halting cache.

In this paper, we introduce a new cache design, which we call a way-halting cache, that reduces the cache's internal activity to nearly the ideal minimums described above, without any performance overhead – neither in the critical path, nor in the hit rate.

Our cache is four-way set-associative, though the method can be applied to any number of ways. We divide each of the four tag arrays into two sub-arrays: the first sub-array (the halt tag array) holds only the low-order four bits of each tag (we'll explain later why we chose to use four bits), and the other sub-array (the main tag array) holds the remaining bits of each tag. A way-halting cache checks all the (four-bit) tags in the halt tag array in parallel with set index decoding, in contrast to other approaches that only check the tags in the cache set specified by the set index. The decoded index activates only the main tag array and data arrays of ways that have *not* been predetermined by the halt tag array check to be a mismatch – predetermined mismatches effectively *halt* the access to a way's main tag array and data array. Note that way-halting does not impact the hit rate, as the hit rate is identical to that of a four-way cache – we've merely caused early termination of accesses to ways that are pre-determined to be misses. Furthermore, through careful design, we can create the halt tag array access and comparisons so they do not extend the cache's critical path. We will show that a way-halting cache comes very close to halting three ways on a hit (and hence accessing only one way), and to halting all ways on a miss (and hence accessing no data) – both approaching the ideal minimums of cache access described earlier.

A way-halting cache makes use of a fully associative memory for the four-bit-wide halt tag array. A key question is whether the power consumed by the fully-associative comparison in the halt tag array outweighs the power savings in the rest of the cache. Our experiments clearly show that the power savings are far greater. We

ISLPED'04, August 9–11, 2004, Newport Beach, California, USA.

also took special care to design that memory using static circuits, in contrast to the dynamic circuits used in the content-addressable memories (CAMs) found in some modern highly associative cache architectures of embedded processors. Thus, our cache does not need special tools or libraries, and is therefore more widely usable by designers.

The rest of this paper is organized as follows. In Section 2, we briefly review related work. We introduce our way-halting cache architecture in Section 3. The design of the halt tag array is discussed in Section 4. Energy savings of way halting cache are presented in Section 5. We compare way halting with other low power caches in Section 6. We summarize the paper in Section 7.

2. Previous Energy-Efficient Cache Designs

Numerous attempts to reduce cache dynamic power have been proposed in recent years. Most of them create designs that are a compromise between set-associative and direct-mapped caches. A *phased-lookup* set-associative cache [5] accesses the tag arrays in the first phase, and then accesses only the one data array corresponding to the matching tag (if any) in a second phase. Power is saved by accessing at most only one data array, but at the cost of performance overhead due to two-phases and hence longer cache access time. A *way-predicting* set-associative cache [14] first accesses only the tag array and data array of one way that is predicted to be a hit. If a miss-prediction occurred, the rest of the ways are accessed in the following cycle. The prediction accuracy for instruction and data caches are reported to be 90% and 80%, respectively [14]. However, the miss-predictions result in performance overhead. A *pseudo-set-associative* cache [7] is a set-associative cache having one tag array and one data array like a direct-mapped cache. On a miss, an index bit is flipped and a second cache entry is checked for a hit – the first and second locations thus form a pseudo-set. Again, dynamic power is reduced at the expense of performance.

Highly associative caches using CAM (content-addressable memory) tags have been utilized in low-energy embedded processors [18]. The high associativity in such caches is not for performance – beyond four or eight ways, the hit rate does not increase much with higher associativity for most applications – but rather due to the use of CAMs for tag comparisons. Our way-halting cache can save on average 20% more energy than the highly associative cache.

Each approach reduces power in exchange for some cost. Note that any method that reduces power per cache access but increases time per access and/or increases the miss rate may actually result in *higher* overall energy, since energy is power times time. This fact makes the total memory-access related energy, not just the power per access, an extremely important metric when evaluating cache designs, and thus the metric we will utilize.

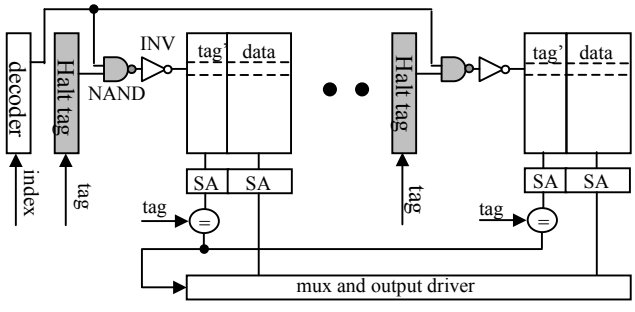

Figure 1: Way-halting four-way set-associative cache architecture. Four bits of each tag is stored in a separate halt tag array for each way. The first inverter of the word line driver is replaced by a NAND gate. (SA= Sense Amplifier; tag'= main tag array).

Partial address matching [11][9] has been proposed to reduce the access time of set associative caches. Liu [11] investigated the possibility of improving the access time of a set associative cache to an approximation of a direct mapped cache with faster matches of five tag bits. Using the same observation, Juan [8] used one tag bit to distinguish the two ways of a two-way set associative cache to achieve an access time close to or equal to that of a direct mapped cache. To reduce energy dissipation, an adaptive serial-parallel highly-associative cache [4] reduces power by first checking only the least four significant tag bits of each tag, and only checking the remaining bits if the first four match, thus reducing tag comparison power only, at the expense of performance (25% slowdown is reported in [4]). Our way halting cache targets the conventional four ways set associative cache and the power consumed by both the tag and data ways are reduced without any performance overhead.

3. Way-Halting Cache Architecture

3.1 Baseline architecture

Our way-halting cache architecture is shown in Figure 1. We utilize a four-way set-associative cache as our baseline architecture, since four ways yields a sufficiently good hit rate for most applications. We use an 8 Kbyte total cache size and a 32-byte line size, though our approach can be applied straightforwardly to caches with other numbers of ways, total size and line size. Our baseline cache thus has 64 sets. The architecture includes a 6x64 decoder, word line drivers, 4 tag arrays, 4 data arrays, sense amplifiers (SA), comparators, 1 multiplexor, and output drivers. The architecture also includes precharging circuits and write circuits that are not shown in the figure. For such a cache, a memory address will be divided into a 6-bit index to determine the set, a 21-bit tag to determine a match, and a 5-bit offset to extract the appropriate bytes from a line. The index bits from a desired address are fed into the decoder. One decoder output will become high and is strengthened by a word line driver consisting of a pair of cascaded inverters (not shown in the Figure), activating four cache lines of the one set of the cache. Four tags and data arrays are thus read out simultaneously through the sense amplifiers. Four comparators compare the desired address tag with the tags read from the tag array to see which way (if any) is a hit. The data of the hit way is sent to microprocessor through the mux and output driver.

3.2 Main idea – early detection of misses

Given a four-way set-associative cache, four tags are checked for each cache access. At most, one of those tags may match, with the other three being mismatches. Usually, the mismatches occur in the low order bits. The intuition behind this phenomenon is the spatial locality of memory accesses with identical high-order address bits and hence identical high-order tag bits.

Therefore, if we can somehow check the low-order bits of a tag *early*, we can detect most misses early, and so we can terminate the access to the full 21 bits of tag as well as to the data arrays before they consume power. We will show the impact of the number of low-order bits on the early miss detection shortly.

3.3 Basic architecture

To enable early detection of misses, we store the low-order four bits of each tag in a separate n-bit-wide memory. We call this memory the *halt tag array* in Figure 1. We call the remaining (21-*n*)-bit-wide tag array the *main tag array*, shown as *tag*' in the figure. In a conventional cache, the desired address' index is decoded, and the resulting decoder output line activates the read of the appropriate tag from the tag array, which is then compared with

the desired tag. Decoding takes some time, during which we have the opportunity to check the *halt tag array* without increasing delay. Since the index has not been decoded yet, we do not know which tag in the halt tag array to read and compare – we therefore compare *all* the halt tags with the lower *n* bits of the desired address' tag. We accomplish this by implementing the *halt tag array* as a fully associative memory, which we point out is only *n* bits wide (and 64 rows long) where *n* is small, making such a memory feasible in terms of size and power. We will study the value range of *n* in Section 4.1.

In a conventional cache, the address decoder would assert a single output line high, and that line would be strengthened by a pair of cascaded inverters to enable reading the appropriate row from the tag and data arrays. In our way-halting cache, that output line should be ANDed by the results of the halt tag array comparison for that row. In other words, only if the low-order four bits match should the cache continue to access the main tag array and the data array; if the halt tag was a mismatch, the output line should *not* go high.

Adding an AND gate after the double inverters would lengthen the critical path. Instead, we can achieve the same logic by replacing the first inverter by a NAND gate as shown in Figure 1; the second inverter makes the total logic an AND. A NAND gate would normally be slower than an inverter. However, the first inverter of the cascaded inverters is typically small – the second inverter is instead appropriately sized larger to drive the signal. Thus, when replacing the first inverter by a NAND gate, we can increase the size of the NAND gate (actually, of its transistors) so that the gate's switching speed is the same as the original inverter.

3.4 Issues on virtually/physically addressed/tagged caches

Our scheme requires that the tags are available no later than the set index. If the tag, but not the set index, needs to be first translated by a translation lookaside buffer (TLB), we have a problem since the halt tag array lookup cannot proceed. Such situations happen in a virtually-indexed and physically-tagged (V/P) cache as in the AMD K6 0, etc. Here, we briefly summarize four combinations of tag and data array addressing using either the virtual address or the physical address: virtually-indexed, virtually-tagged (V/V); virtually-indexed, physically tagged (V/P); physically indexed, virtually tagged (P/V), and physically-indexed, physically-tagged (P/P) cache.

Apart from V/P, all other three cases, namely, the V/V, P/V, and P/P caches, meet our requirement that the tags are available before or at the same time with the index. For V/P caches, the physical tag will not be available until the address translation is finished through the translation lookaside buffer. This will influence the access time of our way halting cache. To solve this problem, we use a technique called *page alignment* or *page coloring* [17].

The main idea of page alignment is that we require the least four bits of the virtual tags from the processor to be equal to the least four bits of the physical tags stored in the cache tags with the help of operating systems. For example, the version of UNIX from Sun Microsystems guarantees the virtual address and physical address are identical in the last 18 address bits [6]. With such an implementation, the halt tag array lookup can proceed before the physical tag is obtained from the TLB, avoiding delays in the original design. Therefore, the way halting cache can be used in all four types of caches. The main drawback of page coloring is that the cache cannot be larger than a page (for each way), but this is not a limitation for embedded systems.

4. Designing the Halt Tag Array

The most important component in a way-halting cache is the *halt tag array*, which must be designed not only to be faster than the index decoder does, but also to consume low enough energy so that we obtain overall energy savings. The two most important considerations in the design of the halt tag array are: (1) the bit width of the array, and (2) the implementation of the fully associative comparison circuitry.

4.1 Bit width of the halt tag array

We examined the impact of the halt tag array's bit width on the number of ways that can be halted. Our goal was to find the minimum number of bits that halts nearly three of the four ways per hit, or conversely stated, activates only one of the four ways per hit. Theoretically, there are at least two bits across the four tags that can be used to differentiate any one from the others. However, to determine which such two bits to use is not an easy job since they vary from set to set. Dynamically determining the two bits is even more expensive in both delay and energy. Thus, a better solution is to use more bits in fixed positions to accommodate all the cache sets.

The spatial locality of memory accesses results in the address sequences sent to the cache tending to be in the "near neighborhood." In other words, only the low-order bits of addresses toggle most of the time. Thus, it is reasonable to use the low-order

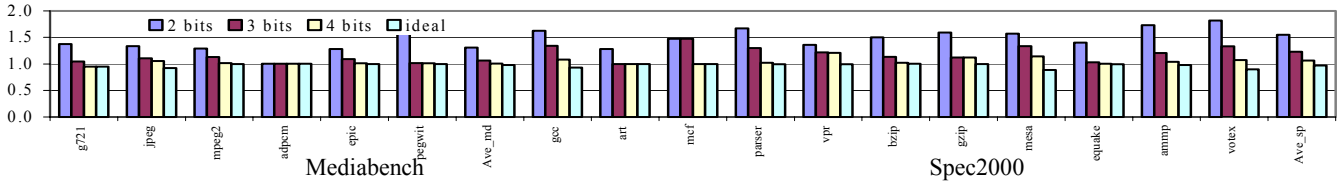

Figure 2: Average number of ways of instruction cache opened when 2-bit, 3-bit, and 4-bit are compared in parallel with address decoder. *Ave_md* and *Ave_sp* stand for average of Mediabench, and Spec 2000 respectively.

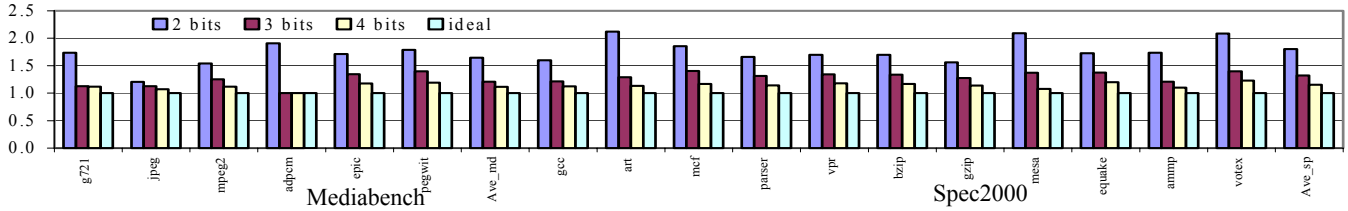

Figure 3: Average number of ways of data cache opened when 2-bit, 3-bit, and 4-bit are compared in parallel with address decoder. *Ave_ps, Ave_md* and *Ave_sp* stand for average of Powerstone, Mediabench, and Spec 2000 respectively.

tag bits in the halt tag array. Subsequently, the number of low-order tag bits, i.e., the halt tag array bit width needs to be determined. The wider the array is, the more accurate the way filtering is, yet the higher the energy and time. We varied the bit width from 2 to 4 and measured the average number of ways that are activated. We simulated a variety of benchmarks for an 8 Kbyte, 32-byte line size cache using SimpleScalar [2]. The benchmarks included programs from Motorola's Powerstone suite [12] (*pjpeg, padpcm, auto2, bcnt, bilv, binary, blit, brev, crc, ucbqsort, fir, g3fax, and v42*), MediaBench [10] (*g721, jpeg, mpeg2, adpcm, epic, and pegwit*) and eleven programs from Spec2000 [8] *(gcc, art, mcf, parser, vpr, bzip, gzip, mesa, equake, ammp, and votex)*. We used the reference input vectors with each benchmark as program stimuli. We include data for every benchmark that we ran; we did not run all benchmarks from the various suites simply due to time constraints.

The results are shown in Figure 2 and Figure 3 for instruction and data caches, respectively, with averages for each benchmark suite circled. Taking the Spec2000 result of an instruction cache (Figure 2) as an example, the ideal average number of ways that should be opened is 0.97 (1 for hits and 0 for misses). Using 2, 3, and 4 bits in our halt tag array, the result is 1.55, 1.23, and 1.06 respectively. The reading for other benchmarks and Figure 3 is similar. We see that a bit width of 4 comes very close to the ideal situation of only accessing one way per hit and zero way per miss.

We also did the experiments using cache sizes of 16 Kbyte and 32 Kbyte for four way set associativity, obtaining similar results. We did not do experiments for caches with associativity more than four, because when associativity is higher than four, the benefits of hit rate tend to diminish but with a longer cache access time that impacts microprocessor's performance.

4.2 Halt tag array fully-associative memory design

Each halt tag array is a 64x4 fully associative memory. If we do not design that array properly, it may consume too much energy and hence mitigate savings obtained from halting ways.

We first designed the halt tag array using traditional 10-transistor CAM cells utilizing dynamic circuit techniques, as found in highly-associative CAM-tag based caches. We laid out the halt tag array, as well as the rest of the cache including the main tag array and the data array SRAM, in a TSMC 0.18 micron CMOS technology obtained through MOSIS [13].

However, we found that designing the halt tag array as a fully associative memory built using *static* circuit (SRAM-based) techniques resulted in a lower energy per access. This is because the

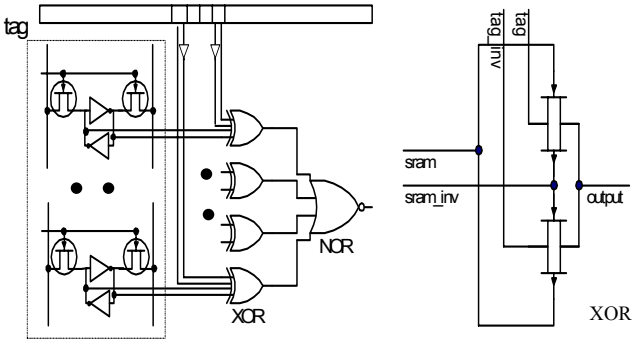

Figure 5: Design of a fully associative memory for the halt tag array, based on a static circuit only. The sixteen input static comparator is composed of four XOR gates and one NOR gate. Eight inputs come from the SRAM cells that store the halt tag, while the other eight inputs come from the desired address' least four-tag bits.

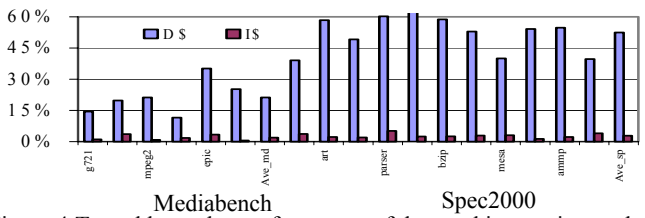

Figure 4:Tag address change frequency of data and instruction cache.

switching activity in the halt tag array is not very high. Static circuits only consume power (dynamic power, that is) when the circuit's inputs change, while dynamic circuits consume power even when there is no switching activity. We measured the percentage of the tag changes from the address streams sent to the instruction and data caches respectively. The results are shown in Figure 4. We observed that the data cache tag changes more frequently due to less spatial locality than the instruction cache, but changes are still not high on average. Furthermore, even when there is a change in tag bits, only a few comparator output bits change in the halt tag array, keeping the dynamic power low for our static circuit. Using a static circuit approach also has the advantage of adopting standard SRAM and logic tools off-the-shelf.

The design of our halt tag array is shown in Figure 5. Here only one word of the array is depicted, which consists of four standard SRAM cells (two are shown). We also show a static comparator on the right hand side. The static comparator component must execute as fast as the address decoder component to avoid lengthening the critical path. We employed the same decoder architecture as in CACTI [15]. Both components have two levels of gates. We designed our XOR and NOR gates of the comparator to be as fast as the address decoder, through laying out our cache using Cadence in technology of 0.18um. We extracted the cache layout, obtained the net lists, and did the time simulation of the cache using Spectra, a tool from Cadence. The size of one static comparator is 3 µm x 16µm. The total area overhead is less than 2% of the total cache area.

5. Way-Halting Cache Energy Savings Results

5.1 Energy modeling

In order to evaluate the difference of energy dissipation between a way-halting and conventional set-associative cache, we consider only the energy dissipation per cache access in this section. In a later section, we will consider energy dissipation related to cache misses such as off chip memory and microprocessor stall energy.

We computed energy as follows. We use E_{dec}, E_{tag}, E_{data}, E_{pre}, E_{com}, E_{mux}, E_{SA}, *and* E_{CMP} to represent the energy dissipation of the address decoder, one tag array, one data array, one way's precharging circuit, one way's comparator circuit, the mux and output driver, one way's sense amplifier circuit, and a comparator, respectively. We use E_{con} and E_{wh} to represent the energy dissipation of a conventional four-way set-associative cache and of our way-halting cache, respectively. Thus, the energy of a way-halting and conventional four-way set-associative cache can be computed as follows:

Equation 1:

$$E_{con} = E_{dec} + E_{mux} + 4*(E_{tag} + E_{data} + E_{pre} + E_{com} + E_{SA})$$

$$E_{wh} = E_{dec} + E_{mux} + n*(E_{tag} + E_{data} + E_{pre} + E_{com} + E_{SA}) + 4* E_{tha}$$

n is the average number of ways that are activated of way-halting cache. It is easy to see that way halting and conventional four-way set-associative caches share the common decoder and mux. The difference is that a way-halting cache may access less than four ways of data and tag arrays. In addition, in our way halting cache,

we have to consider the energy overhead of *halt tag array,* which is $E_{tha.}$ We laid out the cache using Cadence at technology of 0.18um.

5.2 Energy savings

Figure 6 provides energy savings, compared to a conventional four-way set-associative cache (whose energy is represented as 100%), of our way halting cache using a four-bit wide halt tag array. We show data for both the static and dynamic circuit implementations of the halt tag array. We see that a way-halting instruction cache using a static halt tag array (I$-static) consumes only about 30% of the energy of a conventional cache, meaning a 70% savings. Likewise, the data cache (D$-static) results in a 65% savings. In contrast, the designs using dynamic halt tag arrays yield only about 45% savings for instruction and data caches. Energy savings were lower for all caches when we used 3 or 2 bit wide halt tag arrays – ranging from 2% to 18% lower.

6. Comparison with Other Low Power Cache Architectures

In this section, we compare the performance and energy consumption of the way-halting cache with previously proposed low-power cache architectures, including CAM-based highly associative, direct-mapped, way predicting, phased, and pseudo-set-associative caches, in terms of performance and energy.

6.1 Performance

Zhang et al. [18] argues that a CAM-based tag array in a highly-associative cache has comparable access latency with an SRAM-based tag array. In order to improve the speed of the CAM tag comparison, they split the match line and employ single-ended sense amplifiers on both segments of the split match lines. Their CAM timing process was not described in detail though a special timing pulse may be employed in their scheme.

From the above discussion, we can conclude that a conventional cache, and hence our way-halting cache, has better, or at least as good, performance compared with a highly associative cache.

Way-prediction does not prolong the access latency, but incurs extra cycles when there is a miss prediction. On average, the correct way is predicted for instruction and data accesses 90% and 80% of the time respectively [14]. The performance overhead of way prediction is around 3% due to miss prediction. A direct mapped cache has a faster access time than a four-way set-associative cache. In fact, it can be as high as 20% faster than a same size four-way set-associative cache [15]. A phased cache will not prolong the access time but needs two cycles instead of one cycle in a four-way set-associative cache. A pseudo-set-associative cache requires two extra cycles when there is a miss prediction. Figure 7 compares the normalized number of cycles needed to execute each benchmark we simulated. A CAM-based highly associative cache needs the lowest number of cycles, 97.9% of the conventional four-way set-associative cache. A way-halting cache needs the same number of cycles as the conventional four-way cache. Way prediction is the next best performing, followed by pseudo-set-associative and then phased caches.

6.2 Energy

Section 5.2 only considers energy per cache hit. In this section, we compute the overall energy consumption taking into account the off-chip memory and the processor core. The energy model is given in the following equations:

1. $overall\_energy = no\_of\_hits * \underline{hit\_energy} + \underline{no\_of\_misses} * miss\_energy$
2. $miss\_energy = offchip\_access\_energy + uP\_stall\_energy + \underline{cache\_block\_fill\_energy}$

In the first equation, the *no_of_hits* and *no_of_misses* are obtained by running SimpleScalar with different cache configurations. The hit_energy is computed through simulation of circuits extracted from our layout of SRAM cache using Cadence [3].

Determining the *miss_energy* in the second equation is more involved. The *offchip_access_energy* value is the energy for accessing off-chip memory and the *uP_stall_energy* is the energy for the microprocessor when it is stalled due to cache misses. The *cache_block_fill_energy* is the energy to fill the cache with a new block. The first two terms are highly dependent on the memory model and microprocessor model used in a system. Results from one real system may be entirely different from another. Therefore, we choose instead to create a "realistic" system, and then to vary the configurations to see the impacts on energy distribution. We examined all three terms in equation 2 for typical commercial memories and microprocessors. We found that *miss_energy* is 50 to 200 times the *hit_energy*. Thus, we remodeled the *miss_energy* using the following equation:

3. $miss\_energy = k\_miss\_energy * hit\_energy$

We will consider the situations where *k_miss_energy* is equal to 50 and 200 respectively.

To make a fair comparison with highly-associative caches, we use the same number of sub-banks in our way halting and four-way set-associative cache so that the two caches have mostly the same features, such as data array access time, interconnection areas and length, etc. A highly-associative CAM tag based cache has larger area than an SRAM cache. Thus, our four-way way-halting cache will have shorter interconnections, and hence faster access time. However, we still assume both of them have the same access latency when comparing the energy dissipation and performance (thus, we are giving highly-associative caches an advantage). Figure 8 show the energy dissipation of the way-halting, CAM-based highly-associative, direct mapped, way predicting, phased, and pseudo-set-associative caches, using *k_miss_energy = 50*. The energy is normalized with respect to a conventional four-way set-associative cache equaling 100%. We see that a way-halting cache is most energy efficient. Although the energy difference compared with some of the other cache designs may seem small,

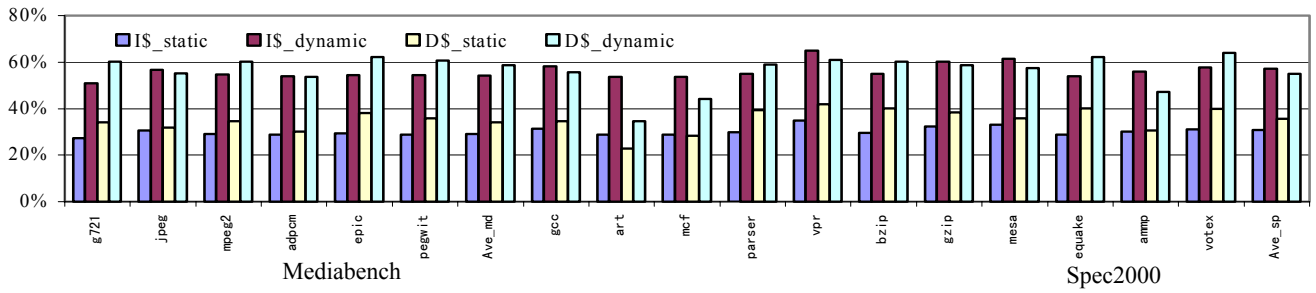

Figure 6**:** Normalized energy dissipation when static and dynamic CAM comparators are used in parallel with decoder.

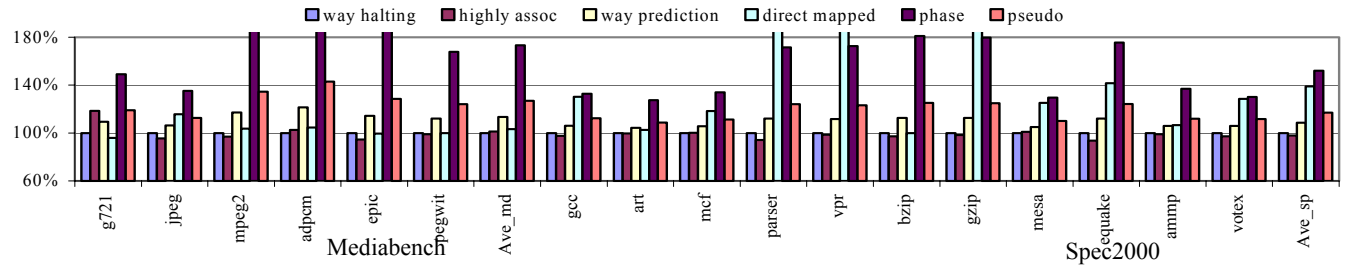

Figure 7: Total cycles for various cache designs, for MediaBench and Spec2000 benchmarks, normalized to a conventional 4-way cache.

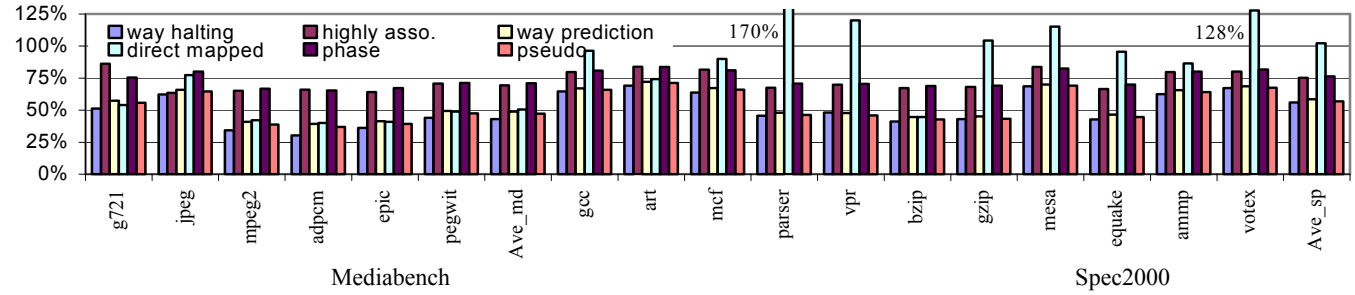

Figure 8: Energy dissipation for various cache designs, for MediaBench and Spec2000 benchmarks, normalized to a conventional 4-way cache.

bear in mind that these savings come with *no performance penalty* compared to a four-way cache – refer back to Figure 7 to see the performance penalties of all the other approaches.

We also generated the data for *k_miss_energy = 200*. Way-halting still dissipated the least energy on average, although highly-associative was more competitive – a high energy penalty (200) for off-chip access means that the slightly lower miss rate due to high-associativity saves more energy than the case of just a high penalty of 50.

7. Conclusion

A way-halting cache is able to save, across three different benchmark suites, an average of 45% to 60% of the energy of a conventional four-way set-associative cache, with only 2% area overhead, and no performance penalty – neither additional cycles nor longer critical path. That energy savings is better than previous approaches using regular associativity. Although the energy savings are only slightly better than some of those approaches, all those other approaches introduce performance overhead. Way-halting also saves energy over a highly associative approach (which has a slight performance advantage on average), while also using static circuits (SRAM) only and hence using standard memory compilers and tools. We designed the way-halting cache using a combination of architectural and layout methods. A key feature of our design is the use of a small fully associative memory for the halt tag array based on a static circuit rather than a dynamic one, with the static circuit saving more power because of the tendency of address tags to stay the same.

Acknowledgements

This work was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (2003-HJ-1046G).

References

[1] Advanced Micro Devices, http://www.amd.com.
[2] 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.
[3] Cadence, http://www.cadence.com
[4] A. Efthymiou and J.D. Garside, "An Adaptive Serial-Parallel CAM Architecture for Low-Power Cache Blocks," Proceedings of the International Symposium on Low Power Electronics and Design, 2002.
[5] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "SH3: High code density, low power," IEEE Micro, Dec. 1995.
[6] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," International Student Edition, third Edition. Morgan Kaufman, 2002.
[7] M. Huang, J. Renau, S.M. Yoo, and J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," International Symposium on Low Power Electronics and Design, 2001.
[8] http://www.specbench.org/osg/cpu2000/
[9] T. Juan, T. Lang, and J. Navarro, "The Difference-bit Cache," in Proceedings of the 27th Annual International Symposium on Computer Architecture, 1996.
[10] C. Lee, M. Potkonjak and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," International Symposium on Microarchitecture, 1997.
[11] L. Liu, "Cache Design with Partial Address Matching," in Proceedings of the 27th Annual International Symposium on Microarchitecture, December 1994.
[12] A. Malik, B. Moyer and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," International Symposium on Low Power Electronics and Design, June 2000.
[13] The MOSIS Service, http://www.mosis.org
[14] M. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," International Symposium on Microarchitecture, 2001.
[15] G. Reinmann and N.P. Jouppi. CACTI2.0: An Integrated Cache Timing and Power Model, 1999. COMPAQ Western Research Lab.
[16] S. Segars, "Low power design techniques for microprocessors," International Solid-State Circuits Conference Tutorial, 2001
[17] G. Taylor, P. Davis, and M. Farmwald, "The TLB slice -- A Low-Cost High-Speed Address Translation Mechanisms," in Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990.
[18] M. Zhang and K. Asanović, "Highly-Associative Caches for Low-Power Processors," Kool Chips Workshop, in conjunction with International Symposium on Microarchitecture, Dec. 2000.