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

CHUANJUN ZHANG
San Diego State University
and
FRANK VAHID, JUN YANG, and WALID NAJJAR
University of California, Riverside

Caches contribute to much of a microprocessor system's power and energy consumption. Numerous new cache architectures, such as phased, pseudo-set-associative, way predicting, reactive-associative, way-shutdown, way-concatenating, and highly-associative, are intended to reduce power and/or energy, but they all impose some performance overhead. We have developed a new cache architecture, called a way-halting cache, that reduces energy further than previously mentioned architectures, 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 predetermines which tags cannot match due to their low-order 4 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, making our cache simpler to design with existing tools. We provide data from experiments on 29 benchmarks drawn from Powerstone, Mediabench, and Spec 2000, based on our layouts in 0.18 micron CMOS technology. On average, we obtained 55% savings of memory-access related energy over a conventional four-way set-associative cache. We show that 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 Descriptors: [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Design, Experimentation, and Performance

Additional Key Words and Phrases: Cache, low power, low energy, embedded systems, dynamic optimization

This work was supported by the National Science Foundation under NSF: CCR-0203829 and by the Semiconductor Research Corporation: SRC: 2003-HJ-1046G.
Frank Vahid is also with the Center for Embedded Computer Systems, UC Irvine.
Authors' addresses: Chuanjan Zhang, Department of Electrical and Computer Engineering, San Diego State University, 5500 Campanile DR, San Diego, CA 92182-1309; email: czhang@mail.sdsu.edu; Frank Vahid, Jun Yang, and Walid Najjar, University of California, Riverside.

1. INTRODUCTION

Reducing the power consumption of both high-end processors and embedded processors is an increasingly important task. High-end processor chips are becoming very high in temperature, calling for lower power solutions. Embedded processor chips often have little cooling capability and thus require low power, and are often powered by batteries and thus require low energy to prolong battery life.

Caches may consume nearly 50% of a microprocessor's power [Malik et al. 2000; Segars 2000]. Most of that power is due to dynamic power—the switching of bits during accesses. Some of that power is due to static power—current leaking even when the cache is not being accessed. Cache designers, for both high-end and embedded processors, must compromise between performance, cost, size, and power/energy dissipation. A fundamental cache design trade-off 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, roughly a 55% reduction per access [Reinmann and Jouppi 1999]. A direct-mapped cache has the additional benefit of not requiring a multiplexor to combine multiple accessed data items from multiple ways, and hence can have faster access time. 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, the cache 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, ideally the cache would have only read and compared one tag entry, and accessed no data entries. In fact, on a miss, the cache does 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 subarrays: the first subarray (the halt tag array) holds only the low-order 4 bits of each tag (we will explain later why we chose to use 4 bits), and the other subarray (the main tag array) holds the remaining bits of each tag. A way-halting cache checks *all* the (4-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 have merely caused early termination of accesses to ways that are predetermined 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 the ideal of halting three ways on a hit (and hence accessing only one way), and of 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 4-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 also took special care to design the fully associative 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 a way-halting cache are presented in Section 5. We compare way halting with other low-power caches in Section 6. We conclude in Section 7.

2. PREVIOUS ENERGY-EFFICIENT CACHE DESIGNS

Numerous attempts to reduce cache dynamic power have been proposed in recent years. The technique that is closest to our design is partial address matching [Efthymiou and Garside 2002; Juan et al. 1996; Liu 1994]. These previous efforts seek to reduce the access time or energy dissipation of set-associative caches. Liu [1994] 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 et al. [1996] 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 [Efthymiou and Garside 2002] reduces power by first checking only the least four significant tag bits of each tag, and then only checking the remaining bits if the first four match. The method thus reduces tag comparison power only, at the expense of performance (a 25% slowdown is reported in Efthymiou and Garside 2002]). The cache can operate in that serial mode, or in the traditional parallel mode in which the complete tag is checked at once (but with no power savings). In contrast, our way-halting cache targets a conventional four-way

set-associative cache, reduces power consumed by both the tag and data ways, and has no performance overhead.

Among other schemes, most of them create designs that are a compromise between set-associative and direct-mapped caches. A *phased-lookup* set-associative cache [Calder et al. 1996; Hasegawa et al. 1995] 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 [Inoue et al. 1999] [Powell et al. 2001] first accesses only the tag array and data array of one way that is predicted to be a hit. If a misprediction occurred, the rest of the ways are accessed in the following cycle. The prediction accuracy for instruction and data caches is reported to be 90% and 80%, respectively [Powell et al. 2001]. However, the mispredictions result in performance overhead. A *reactive-associative* cache (RAC) [Batson and Vijaykumar 2000] uses both selective direct mapping and way prediction. Selective mapping places most of the blocks in direct-mapped positions that are accessed first. If those miss, the other ways are accessed following way prediction. The tag array in a RAC is arranged as a set-associative cache; however, the data array is arranged as a direct-mapped cache without the multiplexors as needed in set-associative cache. A *pseudoset-associative* cache [Huang et al. 2001] 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. Panwar and Rennels [1995] proposed a method to skip tag comparisons when accessing the same cache line as the last cache access.

Some techniques require software assistance. A *programmable tag size* cache [Petrov and Orailoglu 2001] allows software to define the number of tag bits that should be compared—a useful situation in loops where only a few bits change from one address to the next. A *direct-addressed cache* [Witchel et al. 2001] utilizes a special compiler that eliminates tag checking when the compiler knows the accessed line is the same as the previous access, resulting in 9–40% data cache energy savings at the expense of the need for a special compiler and special tag check directives.

Some methods, complementary to those above, seek to tune the cache configuration to a particular application. A *way-shutdown* cache [Albonesi 2000; Malik et al. 2000] is a configurable set-associative cache where some ways can be shutdown. For applications not needing all the available ways and total cache size, shutting down ways can save dynamic power per access without much performance overhead. Furthermore, by adding sleep transistors to the cache's SRAM cells, way shutdown can also save static power [Zhang et al. 2003], although sleep transistors may slow SRAM access and hence impose performance overhead. A *way-concatenation* cache [Zhang et al. 2003] is a set-associative cache where ways can be logically concatenated to result in a four-way, two-way, or direct-mapped cache, all of the same total size. Way concatenation and way shutdown can be combined to allow for even better tuning of the number of

ways and total size to a particular application, resulting in an average 35% reduction in memory-related energy compared to a conventional four-way cache [Zhang et al. 2003]. Further combining such a cache with a configurable line size, having a 16 byte physical line size but up to a 64 byte logical line size, improves energy reductions to 40% [Zhang et al. 2005].

Highly associative caches using CAM (content-addressable memory) tags have been utilized in low-energy embedded processors, such as the StrongArm processor [Santhanam et al. 1998]. A highly associative cache is a phased cache. In a traditional four-way set-associative phased-cache access, four tags are read from four tag arrays corresponding to the particular set determined by the address index, and compared in parallel using four comparators. With the advent of fast, low-power fully associative CAMs, the four tags of a set could instead be stored in a single four-word CAM, so that the four parallel comparisons could be done more efficiently. Of course, a four-word CAM is quite small—a 32 or 64 word CAM is still very efficient, and thus increasing the set size to 32 or 64 (or even higher) makes sense. So 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. Although our cache also uses a fully associative memory, our parallel comparisons are among all the tags in an array (actually, among just 4 bits of all those tags), rather than among all the tags in a particular set as in a highly associative cache.

Each previous approach reduces power in exchange for some cost. A phased-lookup cache requires more time for every access. A way-predicting, reactive-associative, or pseudo-set-associative cache requires extra time whenever the first way accessed results in a miss. Way-shutdown and way-concatenation caches require profiling, and also save no power if four ways are needed (though the approaches can be combined with the other approaches), or could instead create more misses if configured with too low of associativity. Methods that require special compilers impose tool cost. Highly associative caches are phased cache and hence require more time to access, and they also require special CAM tag memories and clocking. 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 equals 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.

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

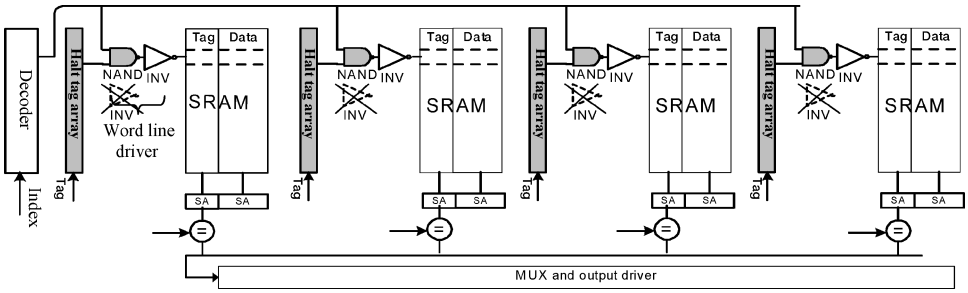

Fig. 1. Way-halting four-way set-associative cache architecture. Four bits of each tag is stored in a separate halt tag array for each way. The first inverter of the word line driver is replaced by a NAND gate.

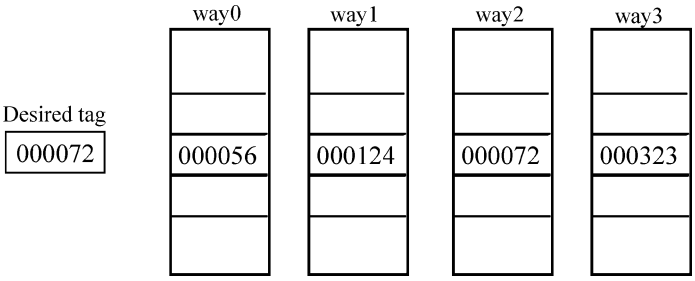

Fig. 2. Tags of a set commonly do not match in their low-order bits.

cache thus has 64 sets. The architecture includes a 6 × 64 decoder, word line drivers, four tag arrays, four data arrays, sense amplifiers (SA), comparators, one 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 can be detected in the low-order bits. This phenomenon is illustrated in Figure 2, which represents a 21-bit tag in hexadecimal. Because memory accesses tend to exhibit spatial locality, the high-order tag bits of cache items frequently match. Instead, the differences between two

tags tend to appear in the low-order bits. If the probability of a match in a single low-order bit is 50%, then the probability of the four lowest-order tag bits matching would be $0.5^4 = 6.25\%$. In other words, the probability of a mismatch in the least four tag bits would be $100\% - 6.25\% = 93.75\%$. This result matches our experimental results shown in Figures 3 and 4.

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 4 bits of each tag in a separate n-bit-wide memory. We call this memory the *halt tag array*, as shown in Figure 1. We call the remaining (21-n)-bit-wide tag array the *main tag array*. 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 the cache can check the *halt tag array* without increasing delay. Because the index has not been decoded yet, the cache does not know which tag in the halt tag array to read and compare—the cache therefore compares *all* the halt tags to 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 4 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. The identical technique of replacing the first inverter by a resized NAND gate was used by Zhang in the way-concatenate cache in [Zhang et al. 2003], with detailed layout and timing analysis results showing no lengthening of the critical path.

We point out that we have not changed the cache subarray organization to implement the way-halting cache. Based on the CACTI model, the data memory is divided into four subarrays for cache sizes of 8 Kbytes, 16 Kbytes, and

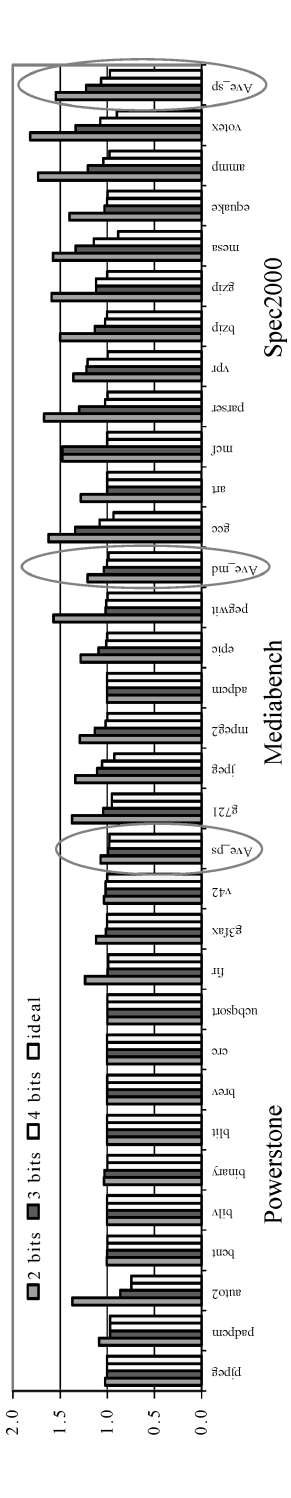

Fig. 3. Average number of ways of instruction cache opened when 2 bits, 3 bits, and 4 bits are compared in a halt-tag array in parallel with the address decoder. *Ave_ps, Ave_md, and Ave_sp* stand for average of Powerstone, Mediabench, and Spec 2000, respectively.

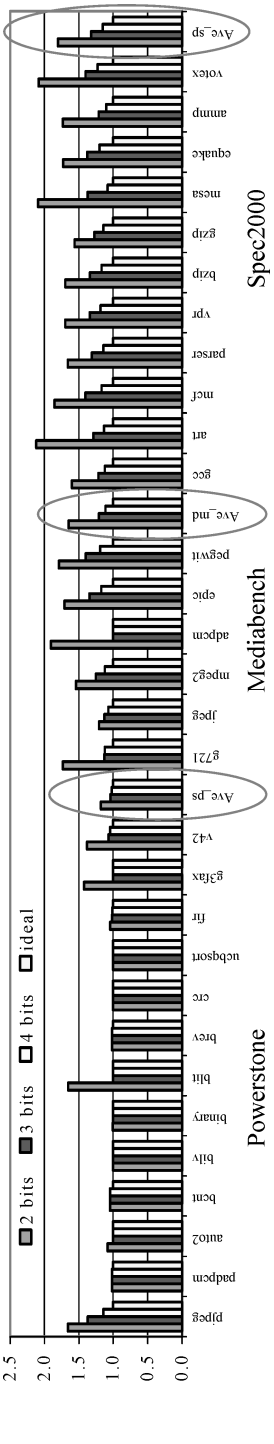

Fig. 4. Average number of ways of data cache opened when 2 bits, 3 bits, and 4 bits are compared in a halt-tag array in parallel with the address decoder. *Ave_ps, Ave_md, and Ave_sp* stand for average of Powerstone, Mediabench, and Spec 2000, respectively.

32 Kbytes, to achieve the best trade-offs of cache size, performance, and energy consumption.

3.4 Issues with 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 look aside buffer (TLB), a problem exists since the halt tag array lookup cannot proceed. Such situations happen in a virtually indexed and physically tagged (V/P) cache, as appear in processors like the AMD K6 [Advanced Micro Devices], MIPS R10K [MIPS Technologies], PowerPC [IBM], 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 cases, meet our requirement that the tags are available before or at the same time as the index. For V/P caches, the physical tag will not be available until the address translation is finished through the translation look ahead buffer (TLB). 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* [Taylor et al. 1990].

The main idea of the page alignment here is that we require the least 4 bits of the virtual tags from the processor to be equal to the least 4 bits of the physical tags stored in the cache tags, with the help of the operating system. For example, the version of UNIX from Sun Microsystems guarantees the virtual address and physical address are identical in the last 18 address bits [Hennessy and Patterson 2002]. 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 typically 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 is 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 2 bits across the

four tags that can be used to differentiate any one from the others. However, to determine which such 2 bits to use is not an easy job since the location of those 2 bits may vary from set to set. Dynamically determining the 2 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 implies that the address sequences sent to the cache tend 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 tag bits in the halt tag array. Subsequently, the number of low-order tag bits, that is, 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 having a random replacement policy, using SimpleScalar [Burger and Austin 1997]. The benchmarks included programs from Motorola's Powerstone embedded suite [Malik 1997] (*pjpeg, padpcm, auto2, bcnt, bilv, binary, blit, brev, crc, ucbqsort, fir, g3fax,* and *v42*), MediaBench [Lee et al. 1997] (*g721, jpeg, mpeg2, adpcm, epic,* and *pegwit*), and 11 programs from Spec2000 *(gcc, art, mcf, parser, vpr, bzip, gzip, mesa, equake, ammp,* and *votex)*. To reduce simulation time, we selected a random subset of benchmarks from each benchmark suites. We used the reference input vectors with each benchmark as program stimuli.

The results are shown in Figures 3 and 4 for instruction and data caches, respectively, with averages for each benchmark suite circled. Looking at the Spec2000 results for an instruction cache (Figure 3, the ideal average number of ways that should be opened (i.e., accessed) is 0.97 (1 for hits and 0 for misses). Using 2, 3, and 4 bits in our halt tag array, the average number of ways opened is 1.55, 1.23, and 1.06, respectively. The other benchmarks and Figure 4 display similar averages. 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. Further increasing the number of halt-tag array bits to 5 or more cannot significantly reduce the number of ways opened, but may instead increase area, power, and performance overhead. Therefore, we use 4 bits in the halt-tag array, because that number is the smallest number that achieves close to the ideal.

We also ran the experiments using cache sizes of 16 Kbytes and 32 Kbytes and 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 are small, and instead the cache has a longer cache access time that impacts a microprocessor's performance.

4.2 Halt Tag Array Fully Associative Memory Design

Each halt tag array is a 64×4 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

```
for ( i =1; i<1000; i ++ )
{
   x[i] = y[i] + z[i];
   a[i] = b × c[i];
}
```

Fig. 5. A simple *for* loop example.

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 [The Mosis Service]. We utilized several low-power SRAM design techniques, such as pulse word line control, to limit the bit line swing, and word line segmentation such that only one word (32 bits) is read [Amrutur and Horowitz 1998].

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 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. Locality exists both in instruction and data cache accesses. The tag addresses sent out to the level-one on-chip caches do not change frequently. For example, Figure 5 shows a simple *for* loop. The instructions of this *for* loop may reside in several cache lines (the line size is 32 bytes in our experiments). It is obvious that only the index to the instruction cache would be changed to locate those instructions. On the other hand, the tags will remain the same for about 1000 iterations.

We measured the percentage of the tag changes from the address streams sent to the instruction and data caches, respectively, for the earlier-mentioned benchmarks. The results are shown in Figure 6. 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 enabling adoption of standard SRAM and off-the-shelf logic tools.

The design of our halt tag array is shown in Figure 7. 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 of the figure. 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 [Reinmann and Jouppi 1999]. 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 with a technology of 0.18 μm, as shown in Figure 8. We extracted the cache layout, obtained the net lists, and did the timing simulation of the cache using Spectra, a tool from Cadence. The size of one static comparator is 3 μm $\times$ 16 μm. The total area overhead is less than 2% of the total cache area.

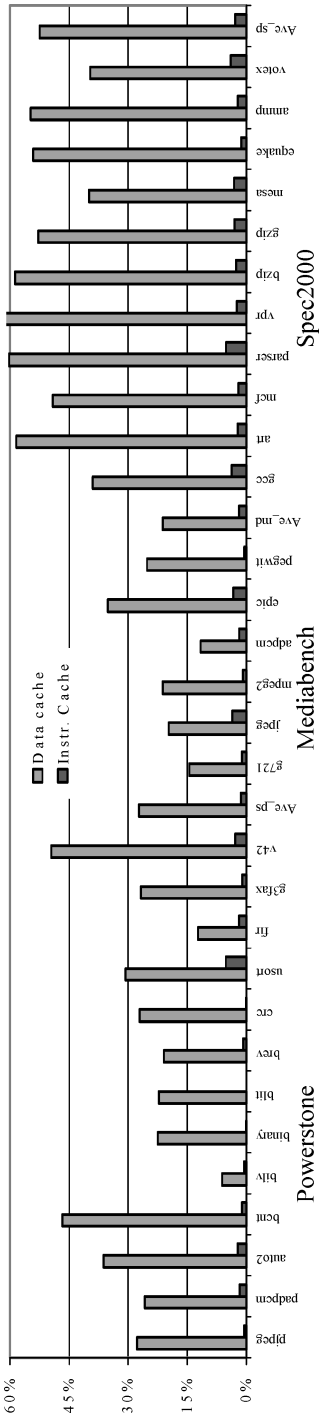

Fig. 6. Tag address change frequency of data and instruction cache.

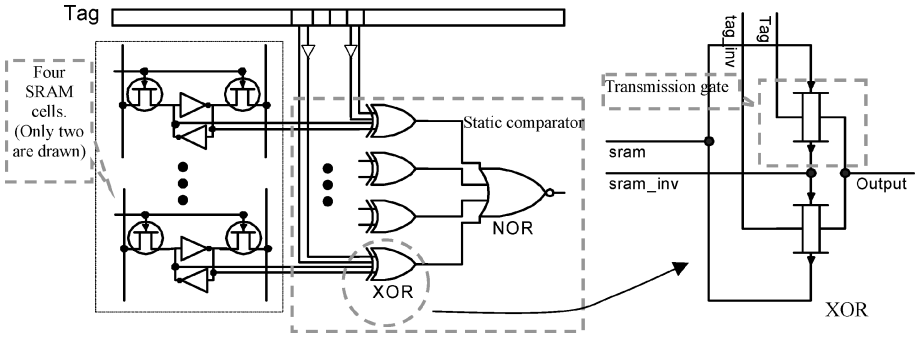

Fig. 7. Design of a fully associative memory for the halt tag array, based on a static circuit only. The 16 input static comparator is composed of 4 XOR gates and 1 NOR gate. Eight inputs come from the SRAM cells that store the halt tag, while the other eight inputs come from the desired address' least four-tag bits.

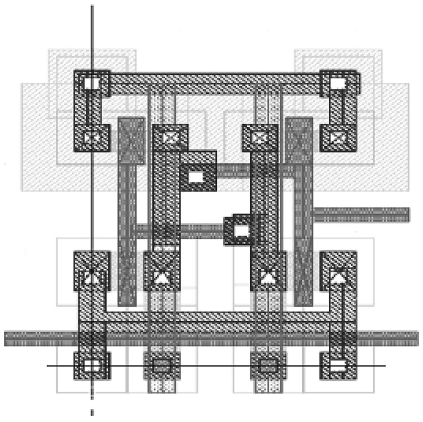

Fig. 8. Layout of an SRAM cell.

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 $E_{\text{dec}}, E_{\text{tag}}, E_{\text{data}}, E_{\text{pre}}, E_{\text{com}}, E_{\text{mux}}, E_{\text{SA}}$, 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:

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

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

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 in a technology of 0.18 μm and measured the energy consumption of each cache component using SPICE simulations.

5.2 Energy Savings

Figure 9 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 4-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

A highly associative CAM-based cache is inherently a phased cache. This phased feature has been described by designers of the StrongARM [Santhanam et al. 1998] and Amulet2e [Garside et al. 1996]. A dynamic circuit (which is usually favored due to low power) CAM needs a precharging phase and an evaluation phase. In the high phase of the clock, the tag CAM evaluates whether there is a hit. The comparison result is stored in a latch and will be used in the low phase of the clock to read data from SRAM. In addition, in the low phase of the clock, the tag CAM is precharged to prepare for the evaluation in the following high phase of the clock. Normally, a clock is a 50% duty pulse, so the critical time of accessing cache should not be the simple addition of the CAM comparison time and data read time. The access time should be the double of the longer phase time that is the low phase for data reading. In a conventional four-way set-associative cache, normally the critical path is on the tag

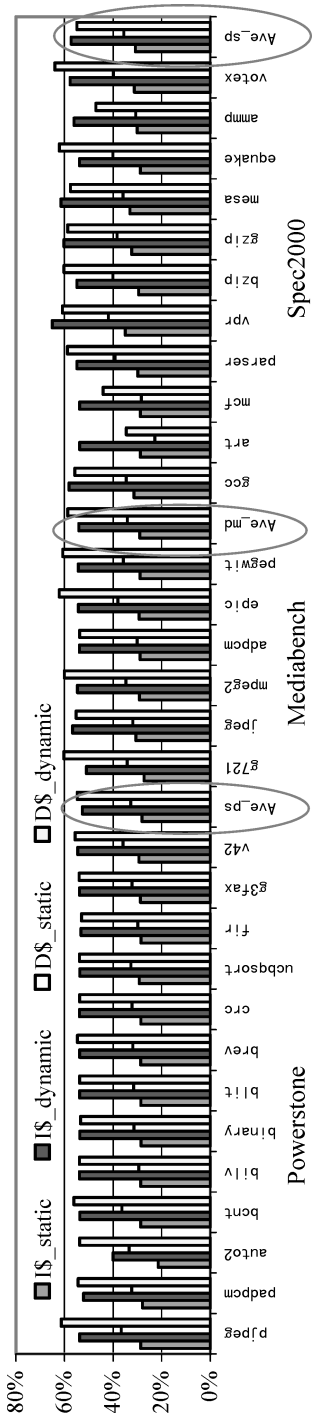

Fig. 9. Normalized energy dissipation when static and CAM comparator are used in parallel with decoder using static comparator.

side, which is around 10% longer than the data read side [Reinmann 1990]. Therefore, we can see that the conventional four-way set-associative cache has a shorter access time than a CAM-based highly associative cache.

Zhang and Asanovic [2000] argues that a CAM-based tag array 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; 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 as good, performance compared with a highly associative cache.

Way-prediction does not lengthen access latency, but incurs extra cycles when there is a misprediction. On average, the correct way is predicted for instruction and data accesses 90% and 80% of the time, respectively [Powell et al. 2001]. The performance overhead of way prediction is around 3% due to misprediction. A direct-mapped cache has a faster access time than a four-way set-associative cache. In fact that access time can be as high as 20% faster than a same size four-way set-associative cache [Reinmann and Jouppi 1999]. 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 misprediction.

Figures 10 and 11 compare number of cycles needed to execute each benchmark using the different cache architectures. 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 (both instruction and data) 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:

$$overall\_energy = no\_of\_hits * \underline{hit\_energy} + \underline{no\_of\_misses} * miss\_energy \quad (1)$$

$$miss\_energy = offchip\_access\_energy + uP\_stall\_energy + \underline{cache\_block\_fill\_energy} \quad (2)$$

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.

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

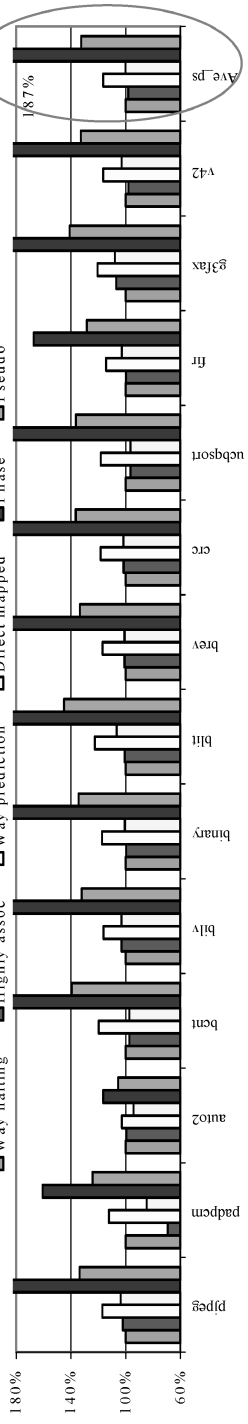

Fig. 10. Total cycles for various cache designs, for Powerstone benchmarks, normalized to a conventional four-way cache.

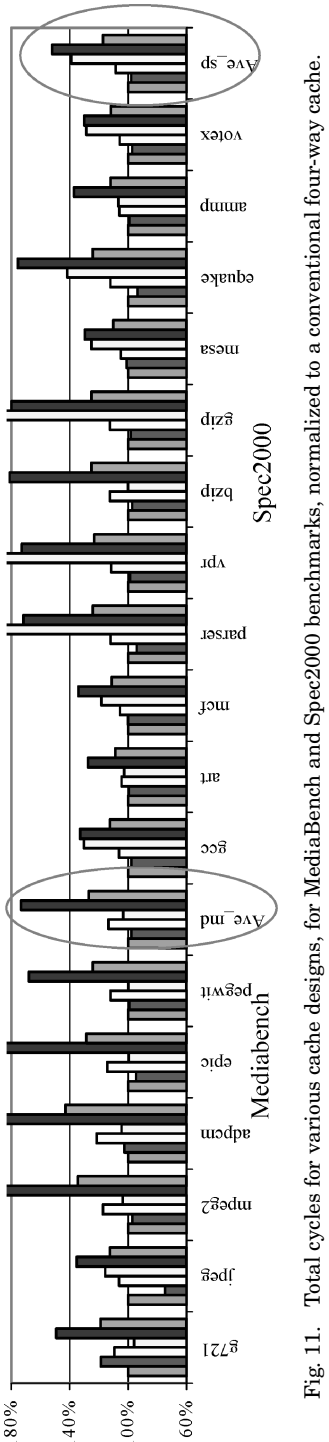

Fig. 11. Total cycles for various cache designs, for MediaBench and Spec2000 benchmarks, normalized to a conventional four-way cache.

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 Eq. (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:

$$miss\_energy = k\_miss\_energy * hit\_energy \tag{3}$$

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 subbanks 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 CAM tag based cache has larger area than an SRAM cache. Thus, our four-way way-halting cache will have shorter interconnections, and hence less 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). Figures 12 and 13 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, bear in mind that these savings come with *no performance penalty* compared to a four-way cache—refer back to Figures 12 and 11 to see the performance penalties of 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

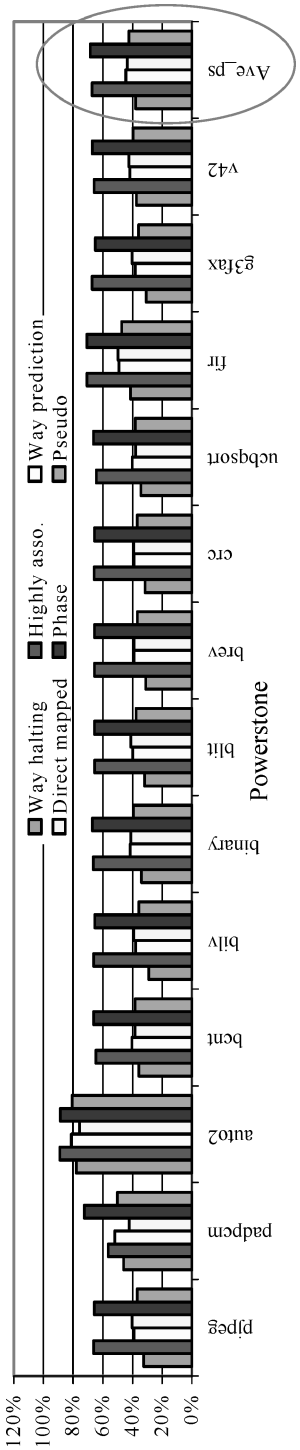

Fig. 12. Energy dissipation for various cache designs, for Powerstone benchmarks, normalized to a conventional four-way cache.

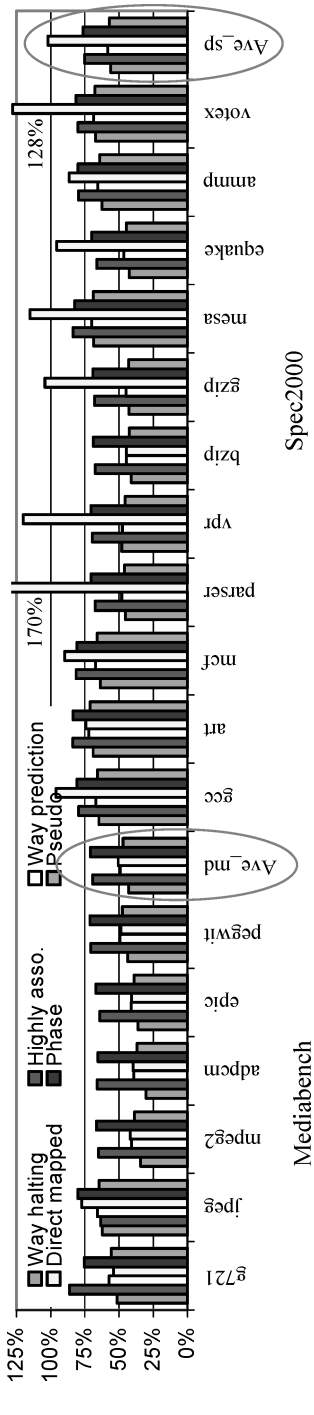

Fig. 13. Energy dissipation for various cache designs, for MediaBench and Spec2000 benchmarks, normalized to a conventional four-way cache.

rather than a dynamic one, with the static circuit saving more power because of the tendency of address tags to stay the same.

REFERENCES

ADVANCED MICRO DEVICES. http://www.amd.com.

ALBONESI, D. H. 2000. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction Level Parallelism*.

AMRUTUR, B. AND HOROWITZ, M. 1998. A replica technique for word line and sense control in low-power SRAM's. *IEEE Journal of Solid-State Circuits 33*, 8, 1208–1218.

BATSON, B. AND VIJAYKUMAR, T. N. 2000. Reactive-associative caches. In *International Conference on Parallel Architectures and Compilation Techniques*.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. University of Wisconsin-Madison Computer Sciences Dept., Technical Report #1342.

CADENCE. http://www.cadence.com.

CALDER, B., GRUNWALL, D., AND EMER, J. 1996. Predictive sequential associative cache. In *International Symposium on High Performance Computer Architecture*.

CKELOV, M. AND DUBOIS, M. 1997. Virtual-address caches. Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro 17*, 5, 64–71.

EDMONDSON, J. H. AND RUBINFIELD, P. I. 1995. Internal organization of the Alpha 21164 a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal 7*, 1, 119–135.

EFTHYMIOU, A. AND GARSIDE, J. D. 2002. An adaptive serial-parallel CAM architecture for low-power cache blocks. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

FURBER, S. B., EFTHYMIOU, A., GARSIDE, J. D., LLOYD, D. W., LEWIS, M. J. G., AND TEMPLE, S. 2001. Power management in the Amulet microprocessors. *IEEE Design & Test of Computers 18*, 2, 42–52.

GARSIDE, J. D., TEMPLE, S., AND MEHRA, R. 1996. The AMULET2e cache system. In *2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*.

HASEGAWA, A., KAWASAKI, I., YAMADA, K., YOSHIOKA, S., KAWASAKI, S., AND BISWAS, P. 1995. SH3: High code density, low power. *IEEE Micro*, Dec.

HENNESSY, J. L. AND PATTERSON, D. A. 2002. *Computer Architecture: A Quantitative Approach, 3rd ed., International Student Edition*. Morgan Kaufman, San Mateo, CA.

HUANG, M., RENAU, J., YOO, S. M., AND TORRELLAS, J. 2001. L1 data cache decomposition for energy efficiency. In *International Symposium on Low Power Electronics and Design*.

IBM. http://www.ibm.com.

http://www.specbench.org/osg/cpu2000/.

INOUE, K., ISHIHARA, T., AND MURAKAMI, K. 1999. Way-predictive set-associative cache for high performance and low energy consumption. In *International Symposium on Low Power Electronics and Design*.

JUAN, LANG, T., AND NAVARRO, J. 1996. The difference-bit cache. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*.

LIU, L. 1994. Cache design with partial address matching. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and poerformance flexibility. In *International Symposium on Low Power Electronics and Design*.

MIPS TECHNOLOGIES, INC. http://www.mips.com.

MONTANARO, J., WITEK, R. T., ANNE, K., BLACK, A. J., COOPER, E. M., DOBBERPUHL, D. W., DONAHUE, P. M., ENO, J., FARELL, A., HOEPPNER, G. W., KRUCKEMYER, D., LEE, T. H., LIN, P., MADDEN, L., MURRAY, D., PEARCE, M., SANTHANAM, S., SNYDER, K. J., STEPHANY, R., AND THIERAUF, S. C. 1996. A 160 MHz 32 b 0.5 W CMOS RISC microprocessor. In *IEEE International Solid-State Circuits Conference*.

THE MOSIS SERVICE. http://www.mosis.org.

PETROV, P. AND ORAILOGLU, A. 2001. Data cache energy minimizations through programmable tag size matching to the applications. In *International Symposium on System Synthesis*.

POWELL, M., AGARWAL, A., VIJAYKUMAR, T. N., FALSAFI, B., AND ROY, K. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *International Symposium on Microarchitecture*.

PANWAR, R. AND RENNELS, D. 1995. Reducing the frequency of tag compares for low power I-cache design. In *SLPE*, pp. 57–62.

REINMANN, G. AND JOUPPI, N. P. 1999. *CACTI2.0: An Integrated Cache Timing and Power Model*. COMPAQ Western Research Lab.

SANTHANAM, S., ET AL. 1998. A low-cost, 300-MHz, RISC CPU with attached media processor. *IEEE Journal of Solid-State Circuits 33*, 11.

SEGARS, S. 2000. Low power design techniques for microprocessors. In *International Solid-State Circuits Conference Tutorial*.

TAYLOR, G., DAVIS, P., AND FARMWALD, M. 1990. The TLB slice—A low-cost high-speed address translation mechanisms. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*.

WITCHEL, E., LARSEN, S., ANANIAN, C. S., AND ASANOVIC, K. 2001. Direct addressed caches for reduced power consumption. In *International Symposium on Microarchitecture*.

YANG, J. AND GUPTA, R. 2002. Energy efficient frequent value data cache design. In *International Symposium on Microarchitecture*.

ZHANG, C., VAHID, F., YANG, J., AND NAJJAR, W. 2004. A way-halting cache for low-energy high performance systems. In *International Symposium on Low Power Electronics and Design*.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003. A highly-configurable cache architecture for embedded systems. In *International Symposium on Computer Architecture*.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2005. A highly-configurable cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems*.

ZHANG, M. AND ASANOVIC, K. 2000. Highly-associative caches for low-power processors. In *Kool Chips Workshop, in conjunction with International Symposium on Microarchitecture*.

Received October 2004; revised February 2005; accepted February 2005