Way Memoization to Reduce Fetch Energy in Instruction Caches

Albert Ma, Michael Zhang, and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{ama|rzhang|krste}@lcs.mit.edu

Abstract

Instruction caches consume a large fraction of the total power in modern low-power microprocessors. In particular, set-associative caches, which are preferred because of lower miss rates, require greater access energy on hits than direct-mapped caches; this is because of the need to locate instructions in one of several ways. Way prediction has been proposed to reduce power dissipation in conventional set-associative caches; however, its application to CAM-tagged caches, which are commonly used in low-power designs, is problematic and has not been quantitatively examined. We propose *way memoization* as an alternative to way prediction. As in way prediction schemes, way memoization stores way information (*links*) within the instruction cache, but in addition maintains a valid bit per link that when set *guarantees* that the way link is valid. In contrast, way prediction schemes must always read one tag to verify that the prediction is correct.

Way memoization requires a link invalidation mechanism to maintain the coherence of link information. We investigate several invalidation schemes and show that simple conservative global invalidation schemes perform similarly to exact invalidation schemes but with much lower implementation cost. Based on HSPICE simulations of complete cache layouts in a 0.25 μm CMOS process, and processor simulations running SPECint95 and MediaBench benchmarks, we show that way memoization reduces the energy of a highly-optimized 16 KB 64-way CAM-tag low-power instruction cache by 21%, an additional 13% savings compared to a way-predicting cache.

1 Introduction

Energy dissipation has become a key constraint in the design of microprocessors. In simple pipelined RISC microprocessors common in embedded applications, instruction cache energy accounts for a significant fraction of total processor energy. For example, the StrongARM SA-110 low-power RISC microprocessor dissipates 27% of total processor power in the instruction cache [13]. Microprocessors usually avoid using direct-mapped caches since higher conflict miss rates lead to larger miss energy penalties as well as reduced performance. However, set-associative instruction caches require greater access energy than direct-mapped caches because of the need to locate instructions in one of several possible locations or ways. In conventional set-associative caches, all ways are accessed in parallel, i.e., includes reading out and comparing cache tags as well as reading out instruction words from each way. Thus, for a fixed cache subbank size, a conventional n-way set-associative cache access requires almost n times the amount of energy as the access of a direct-mapped cache (excluding I/O driver energy).

There are several existing techniques that aim to reduce instruction lookup energy in set-associative caches. One simple optimization is not to perform this search for sequential fetch within each cache line (*intra-line sequential flow*) since we know the same line is being accessed, but for non-sequential fetch such as branches (*non-sequential flow*) and sequential fetch across a cache line boundary (*inter-line sequential flow*) [14, 16], full instruction lookups are performed. This eliminates around 75% of all lookups. This reduces the number of tag readouts and comparisons as well as the number of instruction words read out. We include this optimization in the baseline cache to which we will compare our results.

Another approach uses a two-phase associative cache: access all tags to determine the correct way in the first phase, and then only access a single data item from the matching way in the second phase. Although this approach has been proposed to reduce primary cache energy [8], it is more suited for secondary cache designs due to the performance penalty of an extra cycle in cache access time [2].

A higher performance alternative to phased primary cache is to use CAM (content-addressable memory) to hold tags. CAM tags have been used in a number of low-power processors including the StrongARM [13] and XScale [5]. Although they add roughly 10% to total cache area, CAMs perform tag checks for all ways and read out only the matching data in one cycle. Moreover, a 32-way associative cache with CAM tags has roughly the same hit energy as a two-way set associative cache with RAM tags, but has a higher hit rate [3, 13, 20]. Even so, a CAM tag lookup still adds considerable energy overhead to the simple RAM fetch of one instruction word.

Way-prediction can also reduce the cost of tag accesses by using a way-prediction table and only accessing the tag and data from the predicted way. Correct prediction avoids the cost of reading tags and data from incorrect ways, but a misprediction requires an extra cycle to perform tag comparisons from all ways. This scheme has been used in commercial high-performance designs to add associativity to off-chip secondary caches [19]; to on-chip primary instruction caches to reduce cache hit latencies in superscalar processors [4, 10, 18]; and has been proposed to reduce the access energy in low-power microprocessors [9]. Since way prediction is a speculative technique, it still requires that we fetch one tag and compare it against the current PC to check if the prediction was correct. Though it has never been examined, way-prediction can also be applied to CAM-tagged caches. However, because of the speculative nature of way-prediction, a tag still needs to be read out and compared. Also, on a mispredict, the entire access needs to be restarted; there is no work that can be salvaged. Thus, twice the number of words are read out of the cache.

In this paper, we present an alternative to way-prediction — *way memoization*. Way memoization stores tag lookup results (links) within the instruction cache in a manner similar to some way prediction schemes. However, way memoization also associates a *valid bit* with each link. These valid bits indicate, prior to instruction access, whether the link is correct. This is in contrast to way prediction where the access needs to be verified afterward. This is the crucial difference between the two-schemes, and allows way-memoization to work better in CAM-tagged caches. If the link is valid, we simply follow the link to fetch the next instruction and no tag checks are performed. Otherwise, we fall back on a regular tag search to find the location of the next instruction and update the link for future use. The main complexity in our technique is caused by the need to invalidate all links to a line when that line is evicted. The coherence of all the links is maintained through an invalidation scheme.

Way memoization is orthogonal to and can be used in conjunction with other cache energy reduction techniques such as sub-banking [12], block buffering [6], and the filter cache [11]. Another approach to remove instruction cache tag lookup energy is the L-cache [1], however, it is only applicable to loops and requires compiler support.

The remainder of this paper is structured as follows. Section 2 describes the implementation and operation of the way-memoizing cache and shows that way-memoizing instruction cache avoids most tag lookups. Section 3 presents several invalidation schemes that maintain the correctness of way-memoization. Section 4 describes in detail one invalidation scheme that was chosen. Section 5 presents energy and performance estimates for the different variations of the way-memoizing cache and compares the cache with other low-energy instruction caching schemes. Section 6 discusses future work to be explored and Section 7 summarizes the paper.

2 Way-Memoizing Instruction Cache

The *way-memoizing instruction cache* keeps links within the cache. These links allow instruction fetch to bypass the tag-array and read out words directly from the instruction array. Valid bits indicate whether the cache should use the direct access method or fall back to the normal access method. These valid bits are the key to maintaining the coherence of the way-memoizing cache. When we encounter a valid link, we follow the link to obtain the cache address of the next instruction and thereby completely avoid tag checks. However, when we encounter an invalid link, we fall back to a regular tag search to find the target instruction and update the link. Future instruction fetches reuse the valid link.

Way-memoization can be applied to a conventional cache, a phased cache, or a CAM-tag cache. Table 1 compares a way-memoizing cache with the other low-power caches that have been proposed. Of particular interest is the comparison between way-predicting and way-memoizing caches. On a correct way prediction, the way-predicting cache performs one tag lookup and reads one word, whereas the way-memoizing cache does no tag lookup, and only reads out one word. On a way misprediction, the way-predicting cache is as power-hungry as the conventional cache, and as slow as the phased cache. Thus it can be worse than the normal non-predicting caches. The way-memoizing cache, however, merely becomes one of the three normal non-predicting caches in the worst case. However, the most important difference is that the way-memoization technique can be applied to CAM-tagged caches.

To see how the way-memoizing cache works, we need to examine the different cases of how program execution proceeds. The flow of program execution (and thus instruction fetch) at each point can be classified as either sequential, that is, proceeding from one address to the next sequentially adjacent address, or non-sequential. A MIPS RISC ISA is assumed in the following discussion and also in the results.

	normal			way-predicting	way-memoizing		
	conventional	phased	CAM		conventional	phased	CAM
Cache type	I/D-Cache	I/D-Cache	I/D-Cache	I/D-Cache	I-Cache	I-Cache	I-Cache
Speculative?	N/A	N/A	N/A	yes	no	no	no
Predicted/ memoized cost	n tag n word *1* cycle	n tag *1* word *2* cycles	*1* search *1* word *1* cycle	*1* tag *1* word *1* cycle	*0* tag *1* word *1* cycle	*0* tag *1* word *1* cycle	*0* search *1* word *1* cycle
Mispredicted/ unmemoized cost	N/A	N/A	N/A	n tag n word *2* cycles	n tag n word *1* cycle	n tag *1* word *2* cycles	*1* search *1* word *1* cycle

Table 1: N-way set-associative cache comparison.

2.1 Sequential Flow Optimization

Sequential flow can be classified as either intra-line, in which adjacent addresses are within the same cache line, or inter-line, in which adjacent addresses lay across a cache line boundary. These two forms of sequential accesses can be differentiated simply by checking the low-order bits of the instruction address. In the case of intra-line flow, the tag access never needs to be performed, as the current word is guaranteed to be found in the same cache line as the previous word. We include this optimization in all of our cache designs considered in this paper.

To handle the cases of inter-line flow, we augment each cache line with a *sequential link*. A sequential link consists of a *valid bit*, which indicates whether this link is valid, and a *way field*, which points to the way within a cache set that holds the next word to be fetched. The program counter provides the rest of the information to locate the word. Together, these form a unique address in the cache which is guaranteed to hold the desired word when the valid bit is set. If the cache associativity is n, then the way field requires $\log_2 n$ bits.

2.2 Non-sequential Flow Optimization

Non-sequential flow can be further classified as fixed or variable. In fixed non-sequential flow, corresponding to branches and absolute jumps, the branch target address to which instruction fetch flows is fixed. In variable non-sequential flow, corresponding to indirect jumps, the address to which instruction fetch flows depends on program execution. These instructions are less often encountered than standard branches, thus we do not include any optimizations for them in this paper. However, it is possible to optimize this case also, such as by tagging the link register or a return address stack.

To handle the cases of fixed non-sequential flow, we use *branch links*. A branch link, like a sequential link, holds a valid bit and a way field. However, it points to the way

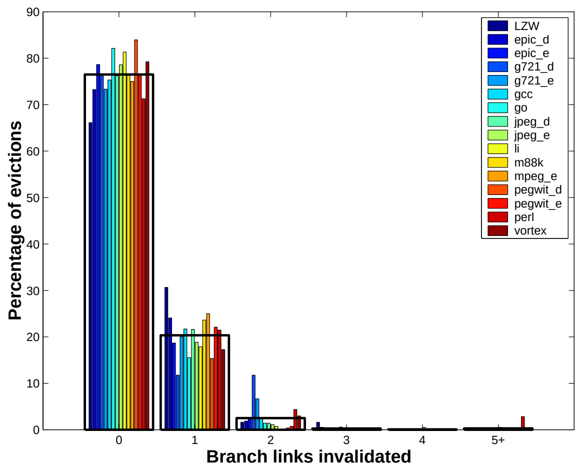

Figure 1: Distribution of the number of branch links cleared per line eviction. The unfilled rectangle overlay shows the average across the benchmarks.

holding the target of the branch or jump, rather than the way hold holding the next sequential instruction. In some ISAs, we need a branch link on every instruction word, since it is possible for each instruction to be a branch. In the MIPS ISA we only need a branch link on every other instruction word, since back-to-back branches are disallowed. These branch links are stored alongside the instruction words in the cache line.

3 Link Invalidation

When a cache line is evicted, there may be valid links elsewhere in the cache that point to the victim line. These links need to be invalidated; otherwise, when they are traversed, instructions from the wrong cache line will be fetched. A *sequential* link to a cache line is easy to find since it can come from only one cache line (corresponding to the preceding cache line address). A tag search locates that cache line (if it is in the cache) and clears the link. However, any *branch* link in the entire cache can point to the victim cache line. We can either search the entire cache for these broken branch links, or we can build a data-structure in the cache to record which lines point to a particular line in the cache, so that only those branch links that need to be are invalidated. We call such schemes exact invalidation schemes. They all require large area overhead and complex control logic.

Figure 1 shows the distribution of the number of branch links that need to be invalidated on each cache line eviction. This data was collected from a MIPS processor simulator with a 16KB I-cache with 32-byte lines running a subset of SPECint95 and MediaBench benchmarks. Around 74% of the time, there are no links to be invalidated. Around 23% of the time, exactly one link needs to be invalidated. The rest represent about 3% of evictions. This graph strongly hints that we can use an inexact but conservative invalidation scheme since there are usually no more than one branch link to invalidate.

In the first inexact invalidation scheme we examine, we augment the cache line with a single *invalidate* pointer plus an *overflow bit* for each cache line. The new cache line structure is shown in Figure 2. The sequential and branch links are shown associated with the tag and with the instruction pairs respectively. The invalidate pointer points to the first branch link that is made to the line. The overflow bit for a line is set when a link is being created to that line but the invalidate link is already occupied. If the overflow bit is set on a line that is being evicted, all branch links and all invalidate links in the entire cache are cleared. This conservative clearing of all links potentially erases a lot of useful way information. However, it avoids needing to keep track of all possible links. Further, from Figure 1 we can see that this should only happen on at most 3% of evictions. We call this the *one-link* scheme since it reserves space for one link.

The next simplification removes the invalidate link entirely, leaving only the overflow bits. Any time a branch link is created, the overflow bit is set on the target line. This causes all the links to be cleared on at most 26% of evictions. We call this the *zero-link* scheme. Finally, for completeness, we can remove the overflow bit and clear all the links on any eviction. We'll call this the *oblivious* scheme.

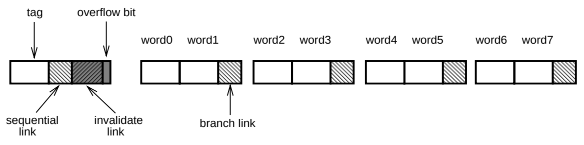

Figure 2: Cache line augmented with the one-link structure implementing an inexact invalidation scheme.

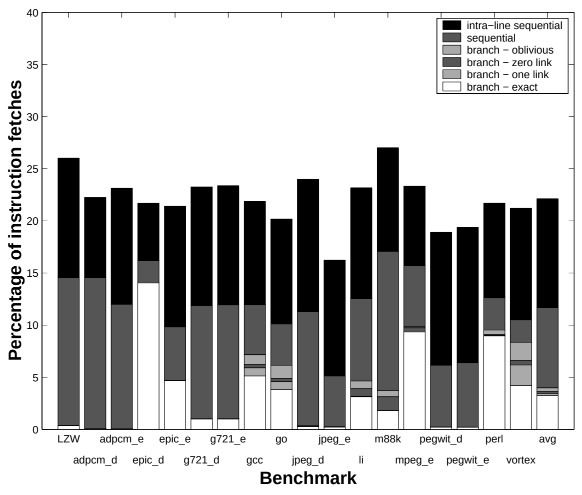

Figure 3: Tag lookups for 4-way set-associative cache.

Figure 3 shows the performance of all the schemes on a 4-way set-associative 16KB cache across the benchmark set (X-axis). Figure 4 shows the same results on a 64-way cache using CAM tags. The rightmost bar is the average across the benchmarks. The top of each vertical subbar represents the fraction of tag lookups performed per instruction fetch for each variation of the cache. The topmost subbar (22% on average) measures the baseline cache, which implements the intra-line sequential optimization. The second subbar (12%) measures a way memoizing cache which implements only the sequential links. The third through sixth bars (approximately 3%) are different variations of the complete way memoizing cache. The bottommost implements the exact invalidation scheme.

The surprising result is that even the oblivious scheme does fairly well compared to the exact scheme. Based on this result, we chose the *zero-link* scheme to implement and study in greater detail. The zero-link scheme has only one extra bit of overhead per cache line compared to the oblivious scheme. The area overhead for the zero-link scheme is 6% on a 4-way set-associative cache and 13% on a 64-way CAM-tagged cache, including the sequential and branch links. Since the MIPS ISA has a branch delay slot, we only need one branch link for every pair of instructions.

4 Operation of the Zero-Link Cache

In this section, we describe the operation of the zero-link cache in detail. The operation of the cache depends on whether it is CAM-tagged. On a non-CAM-tagged cache, a write requires two cycles - the tag-readout and data-write

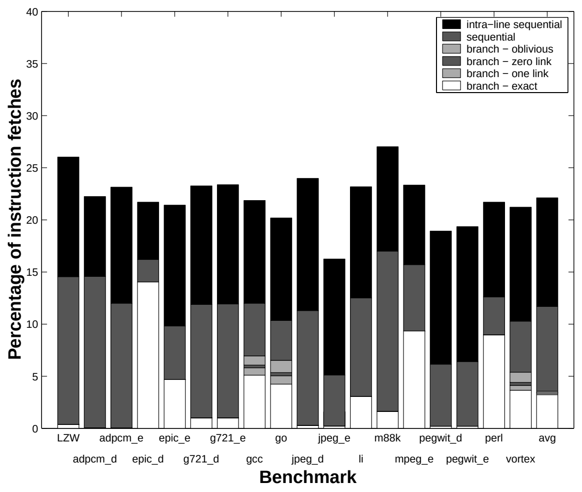

Figure 4: Tag lookup reduction for 64-way set-associative cache.

cannot occur in parallel since it is not known beforehand which way should be written. On a CAM-tagged cache, however, a write can be performed in one cycle since only the matching way is enabled for a write. [20] Figures 5 and 6 describes the timing of both CAM and non-CAM designs. However, for simplicity, we will only describe the CAM-tagged cache here.

When an instruction is fetched from the cache, the corresponding branch link is also read out. In addition, if the instruction is fetched from a different cache line from the last fetch, the sequential link is read out. If the instruction does not take a branch and is not at the end of the cache line these links are not used; on the fetch of the next instruction, the succeeding word on the cache line is read out. Otherwise, if a branch is taken, the branch link is used for the fetch of the next instruction. If a branch is not taken and the instruction is at the end of the cache line, then the sequential link is used. There are three cases to consider: a valid link, invalid link with cache hit, and invalid link with cache miss. Let *T* be the next instruction (or its address) to be fetched. Assume that in cycle *0*, a instruction fetch of instruction *T* is initiated.

In the case when the link to instruction *T* is valid, we skip the tag lookup and directly read out the instruction in cycle *0* and complete the instruction fetch.

In the second case when the link is invalid, a standard cache access (with tag lookup) is performed during cycle *0*. Simultaneously, for a jump or taken branch, the overflow bit is written to indicate that a link to this line will be created. By this time, hardware knows that there was a hit in the cache. During cycle *1*, while the next instruction, instruction *T+1*, is being fetched, the link on the referencing instruction to instruction *T* is created. This relies on the fact that a low-power cache design would divide the

	cycle *0*	cycle *1*	cycle *2*
valid	fetch *T*		
invalid hit	lookup *T* fetch *T* set overflow	fetch *T+1* write link	
invalid miss	lookup t	readout *E* write link	lookup *E-1* invalidate *E-1* seq. invalidate all br.

Figure 5: CAM-tagged cache.

	cycle *0*	cycle *1*	cycle *2*
valid	fetch *T*		
invalid hit	lookup *T* fetch *T*	fetch *T+1* set overflow write link	
invalid miss	lookup *T*	lookup *E-1* write link invalidate all br.	invalidate *E-1* seq.

Figure 6: Non-CAM-Based Cache

cache into multiple sub-banks that could be accessed independently given the right circuitry [12]. There is some probability that the fetch and the rewrite occur to the same bank. In this case, the pipeline is stalled for one cycle.

In the last case, a standard cache access is performed in cycle *0* and results in a cache miss (thus no words are read out). A cache line, line *E*, is selected for eviction. During cycle *1*, its tag field and overflow bit is read out. The tag bits are used to invalidate the sequential link to *E* and the overflow bit is used to determine whether to invalid all links. Also during cycle *1*, the link on the referencing instruction to *T* is created. During cycle *2*, using the tag of the evicted line, the sequential link pointing to line *E* is invalidated through a tag search for line *E-1*. Finally, if the overflow bit on the evicted line was set, we clear of all branch link valid bits using a flash clear. The fetch completes some time later when the target line is loaded into the cache from secondary memory. The added latency for invalidating links is insignificant compared to the cache refill latency.

5 Performance and Energy Evaluation

To evaluate the performance and energy dissipation of alternative designs, we built performance and energy models based on a machine with a similar configuration to a StrongARM microprocessor [13]. We use a low-power 16 KB instruction cache design with 8 sub-banks and 32-byte cache lines. We use SPECint95 and MediaBench benchmarks for our simulations and assume MIPS ISA in

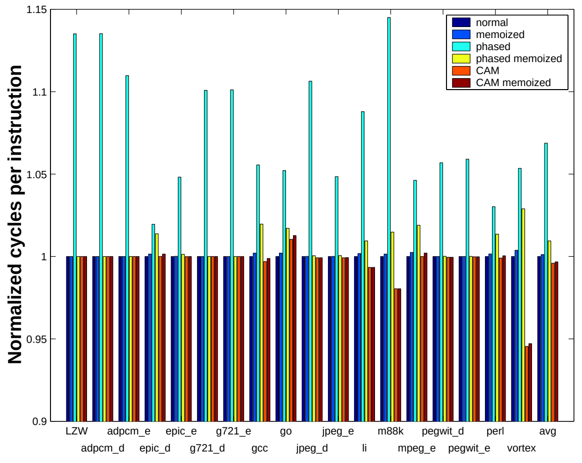

Figure 7: Processor performance overhead for different set-associative caches.

all our results.

5.1 Performance Modeling

Figure 7 shows the performance impact for a range of cache configurations, measured as CPI relative to the normal non-CAM cache configuration. We assume an instruction cache miss penalty of 20 cycles (data caches are not modeled). The first bar shows a conventional 4-way set-associative RAM-tag cache with an LRU replacement policy. The second bar shows the effects of adding way-memoization to the 4-way RAM-tag cache. Performance is slightly reduced because when a link is created, the cache sub-bank which is being accessed cannot simultaneously be used for instruction fetch. Since there are eight sub-banks, this situation occurs about one in eight times a branch link is created. In rare cases, this also causes a stall when a sequential link is created. We see that overall there is only a very slight performance degradation due to the link update activity.

The third bar shows the performance penalty of using a 4-way set-associative phased RAM-tag cache. The phased cache reads tags then data sequentially, and so adds a bubble into the fetch pipeline whenever a branch is taken. This adds a performance penalty of around 7% CPI for our simple pipeline. This number is highly dependent on the cache miss rate and miss penalty. A higher miss rate or miss penalty overshadows the effect of branch penalties such as is the case with `epic_d`.

The fourth bar shows the effect of adding way-memoization to the phased cache. Most of the performance penalty of using phased-caches is eliminated because we avoid the branch-taken penalty whenever the branch target's way is memoized. However, performance is still worse than the non-phased cache schemes.

The fifth bar shows the performance of the 64-way CAM-tagged cache. As in the StrongARM design, we use a FIFO replacement policy within each sub-bank [13]. As expected, the CAM-tagged cache gives the best performance overall with a lower miss rate than the 4-way associative schemes. The sixth bar shows the very slight performance degradation from adding way-memoization to the 64-way CAM tag cache.

5.2 Energy Modeling

We base our energy models on data extracted from circuit layout of actual cache designs (without the way-memoization modifications) in a TSMC 0.25μm CMOS process with 5 Aluminum metal layers. A number of circuit-level power reduction techniques are used in this design, such as sub-banking, low-swing bitline, and segmented wordline. For the RAM-tag cache variants, we assume RAM tags are held in separate smaller arrays to improve speed and reduce power dissipation. For the CAM-tagged cache variants, we use a low-power CAM design that has separate search and write bitlines and which employs a reduced voltage swing on the precharged match lines [20]. We ran HSPICE simulations on circuit netlists obtained with a two-dimensional circuit extractor (SPACE) to obtain energy dissipation numbers for reads and writes for the various components of the cache. These energy simulation numbers are factored by the activity simulation numbers to generate total energy numbers. The CACTI model was not used here since it does not include many of the circuit level energy reduction techniques used in our design. When we used the CACTI model [17] to obtain energy numbers for our cache configuration, we observed that the CACTI result is ($10\times$) greater than our result [20].

Figure 8 shows the energy per access for each of our schemes. The data are grouped into three groups. The first group, the first three bars, are comparisons between 4-way set-associative caches. The middle group, the next four bars, are comparisons between 4-way phased caches. The third group, the last four bars are comparisons between 64-way set-associative CAM-tagged caches. The top section of each bar, labelled overhead, represents the energy used in doing things not required in an unmodified cache such as reading out links and overflow bits.

In the first group, the first bar (*norm*) is for a conventional cache with only the intra-line sequential tag lookups eliminated. The second bar (*seq*) shows the savings from adding inter-line sequential links. Tag readout energy is almost halved and there are also savings in word read outs, but also some overhead is added to read and write the sequential link fields. The third bar (*br*) shows the energy

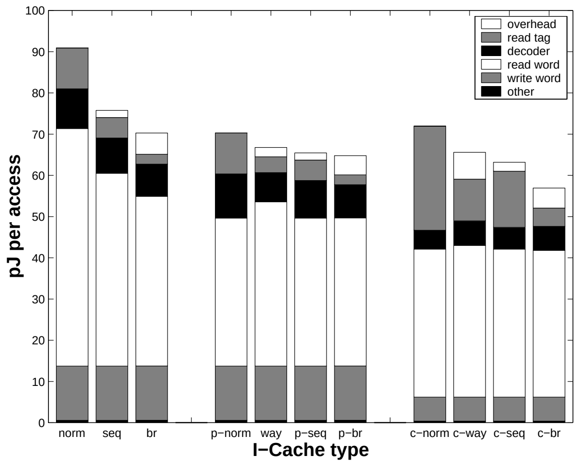

Figure 8: Instruction cache power breakdown.

savings from adding the branch links. Although tag lookup energy is almost eliminated and data word read energy is reduced, there is considerable overhead from accessing the additional branch link state which limits the energy savings. Overall, these optimizations result in an energy saving of around 23% over a low-power cache design that only eliminates intra-line tag lookups.

In the second group, the first bar (*p-norm*) shows the reduction of a phased cache implementation with only intra-line optimization. The next bar (*way*) shows the reduction for a way-predicting cache. The next two bars (*p-seq* and *p-br*) show the additional reduction once sequential and branch links are used.

We make two observations from this group of data. First, the way-predicting cache performs similarly to the phased way-memoizing caches because of its high prediction rate. Intuitively, a way-predicting cache always has a higher prediction rate than that of an equivalent way-memoizing cache, which is already high at 97%. Any valid link in a way-memoizing cache is guaranteed to be a correct prediction in a way-predicting cache. However, even when there is no way information possibly available, the way-predicting cache has a random $1/n$ chance of being correct in its prediction. Thus the worst case of the way-predicting cache is rare. Even though there are a greater number of tag accesses, these do not require a great amount of energy in a conventional 4-way set-associative cache as is evident in the leftmost bar (*norm*).

The second observation is that, because the phased cache already eliminates excess data word access energy, the relative savings are much lower. In fact, adding the non-sequential branch links increases overhead energy per access as shown in column *p-br*. However, as mentioned previously, the branch links improve performance for the phased cache by removing most of the branch taken stalls. We can achieve a total of 7% energy savings over the initial low-power phased cache which only eliminated intra-line tag lookups, while simultaneously improving performance as shown earlier in Figure 7.

The four bars in the last group show the same progression for the 64-way set-associative CAM-tagged cache as in the second group of data. As with the phased cache, the CAM-tagged cache already eliminates excess data word read energy, but the CAM tag search adds considerable energy overhead. Adding the inter-line sequential and non-sequential links gives a large savings in tag energy and results in a total energy savings of 21% over the low-power CAM-tagged cache with only intra-line sequential tag checks eliminated. Since the way links now require six bits (and a valid bit) each, we further reduced the energy overhead by gating the readout of the way links using the valid bit. Thus invalid links do not require as much energy to read. This final CAM-tagged configuration has both the lowest energy and the highest performance of all the configurations considered here.

Finally, we compare the way-memoizing cache (*c-br*) to the way-prediction implemented on the CAM-tagged cache (*c-way*). The tag read energy is higher for the way-predicting cache than for the way-memoizing cache because way-prediction requires that a tag always be read out along with the data in order to verify the correctness of the prediction. This reduces the benefit of reducing the CAM tag searches. In addition, since there are no valid bits to gate the readout of the way links, there is greater energy overhead. Way-prediction only reduces energy consumption by 8%. Thus, way-memoization is able to reduce energy consumption by an addition 13% beyond way-prediction.

6 Future work

There are several modifications that can be performed to reduce the overhead of way-memoization and to further reduce energy consumption in the instruction fetch unit.

When the associativity is high, there is a non-trivial area overhead in keeping the branch links. We can reduce this overhead by storing some of the link bits in the free space in each instruction word. In particular, many branches in the MIPS ISA do not require both register fields. Further, many branch instructions compare one register to the zero register and we could recode these instructions before they are written into the instruction cache [15]. If we do not use branch links for the remaining branch instructions, we can save 5 bits per branch link, reducing the area overhead for a 64-way memoizing cache by 6%. This optimization also reduces the energy overhead that arises from reading out extra state in the cache, which was shown to have a large effect on the energy performance of the way-memoizing cache.

As has been proposed in [7], it is possible to precompute the low-order bits of branch destinations and store them in the instruction cache, instead of the original PC relative offset. As a result, low-order bits do not have to be recalculated every time the branch is executed, thus saving power. Combined with way-memoization, this technique allows us to directly locate the branch target without any calculation and without even knowing the upper bits of the current program counter value. Execution can be continued until an invalid link is encountered, at which time we can regenerate the program counter value from the cache tag. In this way, we can reduce the power required for program counter generation. We expect this optimization to fully explore the energy performance potential of way-memoizing caches.

Way-memoizing caches can be adapted to work for superscalar processors. In particular, if instruction fetch is already using way-prediction as in [18], the modifications for way-memoization should add comparatively little incremental area and energy overhead, while decreasing cache read energy.

7 Summary

In this paper, the *way-memoization* technique is proposed to reduce instruction fetch energy. This technique stores precomputed in-cache links to next fetch locations. By following valid links, the instruction fetch unit can bypass the tag lookup, thus reduce tag lookup energy. Though our scheme requires that we invalidate all links to a line when it is evicted on a cache miss, we have shown that for the SPECint95 and MediaBench benchmarks, simple conservative global invalidation schemes perform similarly to exact invalidation schemes, but with much lower implementation cost. There is no performance degradation associated with the implementation. We compare the way-memoizing cache against other schemes, specifically way-prediction, to reduce instruction cache power. Based on HSPICE cache simulations and processor simulations, we show that way memoization reduces the energy of a highly-optimized 16 KB 64-way CAM-tag low-power instruction cache by 21%, an additional 13% savings compared to a way-predicting cache.

References

[1] N. Bellas, I. Hajj, G. Stamoulis, and C. Polychronopoulos. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *ISLPED*, pages 70–75, August 1998.

[2] B. J. Benschneider et al. A 300-MHz 64-b quad-issue CMOS RISC microprocessor. *JSSC*, 30(11):1203–1214, November 1995.

[3] T. Burd. *Energy-Efficient Processor System Design.* PhD thesis, University of California at Berkeley, May 2001.

[4] B. Calder and D. Grunwald. Next cache line and set prediction. In *ISCA-22*, Italy, June 1995.

[5] Intel Corporation. Intel XScale microarchitecture. http://developer.intel.com/design/intelxscale, 2001.

[6] K. Ghose and M. Kamble. Energy efficient cache organizations for superscalar processors. In *Power-Driven Microarchitecture Workshop, ISCA-98*, June 1998.

[7] L. Gwennap. MAJC gives VLIW a new twist. *Microprocessor Report*, 13(12):12–15,22, September 1999.

[8] A. Hasegawa et al. Sh3: high code density, low power. *IEEE Micro*, 1995.

[9] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED*, pages 273–275, August 1999.

[10] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[11] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An energy efficient memory structure. In *Micro-30*, December 1997.

[12] U. Ko, P. Balsara, and A. Nanda. Energy optimization of multi-level processor cache architecture. In *ISLPED*, April 1995.

[13] J. Montanaro et al. A 160-MHz, 32b, 0.5-W CMOS RISC microprocessor. *JSSC*, 31(11):1703–1712, November 1996.

[14] M. Muller. Power efficiency & low cost: The ARM6 family. In *Hot Chips IV*, August 1992.

[15] M. Panich. Reducing instruction cache energy using gated wordlines. Master's thesis, Massachusetts Institute of Technology, August 1999.

[16] R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power I-cache design. In *SLPE*, pages 57–62, October 1995.

[17] G. Reinman and N. Jouppi. An integrated cache and timing model. http://research.compaq.com/wrl/people/jouppi/cacti2.pdf, 1999.

[18] M. Tremblay and J. M. O'Connor. UltraSPARC-I: A four-issue processor supporting multimedia. *IEEE Micro*, 16(2):42–50, April 1996.

[19] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[20] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Koolchip Workshop, MICRO-33*, December 2000.