Low Cost Instruction Cache Designs for Tag Comparison Elimination

Youtao Zhang
Computer Science Department
University of Texas at Dallas
Richardson, TX 75083

Jun Yang
Computer Science and Engineering Department
University of California at Riverside
Riverside, CA 92521

ABSTRACT

Tag comparison elimination (TCE) is an effective approach to reduce I-cache energy. Current research focuses on finding good tradeoffs between hardware cost and percentage of comparisons that can be removed. For this purpose, two low cost innovations are proposed in this paper. We design a small dedicated TCE table whose size is flexible both horizontally (entry size) and vertically (number of entries). The design also minimizes interactions with the I-cache. For a 64-way 16K cache, the new design reduces the tag comparisons to 4.0% with a fraction only 20% of the hardware cost of the way memoization technique [5]. The result is 40% better compared to a recent proposed low cost design [2] of comparable hardware cost.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]

General Terms

Design

Keywords

Tag Comparison Elimination, Low-Power Instruction Cache

1. INTRODUCTION

Instruction cache in modern processors consumes a significant portion of the chip power. As an example, the StrongArm SA110 spends 27% of its total power in instruction cache [11]. For highly associative caches, the power consumption is a particular concern since large number of tag comparisons are carried in parallel. Therefore, reducing instruction cache power is of great interest in low power processor design community.

Tag comparison elimination (TCE) techniques have been proposed to remove unnecessary tag comparisons to achieve power reduction. A tag comparison is considered unnecessary if we can determine a match or a mismatch without a real comparison. The decision is made based on the runtime information that is maintained in low cost hardware.

Panwar and Rennels [3] classified tag comparisons into intra- and inter- cache line ones. Inter-cache line tag comparisons are from instructions that switch from one cache line to another: the instructions at the end of a cache line and taken branch instructions. A compiler approach is proposed to annotate these instructions such that all intra-cache tag comparisons can be removed. In *way memoization* technique proposed by Ma, Zhang and Asanvoic [5], various linking information and way numbers are maintained per cache line. Although it achieves good tag comparison reduction, the extra hardware required is overly high and the cache needs substantial amount of modifications. These limitations made the design undesirable for highly associative caches.

Therefore, recent research focuses on finding good trade-offs between hardware cost and TCE amount. The history-based TCE is such a scheme [2]. This technique is closely coupled with the design of BTB. Especially, it needs to store the target addresses of *not-taken branches*, as opposed to storing targets for only the taken branches that is normally used. Consequently, the performance degradation occurs due to less effective BTB. On average, we observed 2% increase in execution time which means that besides the instruction cache, the rest of the processor burns out 2% of extra power. Alternatively, if the not-taken branch targets are not stored, the required tag comparisons increase twice as much.

A number of other techniques are also proposed to reduce the instruction cache power consumption. "Way prediction" technique predicts a cache way and accesses it as a direct-mapped cache [1, 4]. This technique has been realized in commercial processors [8]. Our design is non-speculative and thus different from way prediction. Phased cache separates tag and data access into two stages. After tag access stage, only one data line is accessed if a tag match has been found. Filter cache [10] adds another cache hierarchy in front of L1 cache with a smaller L0 cache so that a large percentage of instruction cache accesses hit in L0 cache. It is a design that trades performance for power.

In this paper, we propose two innovative TCE designs for the instruction cache. A small table is specifically designed for TCE only. In this way, the cache or the BTB is minimally affected. In addition, the energy consumed to access a small table is less than the energy to access the information stored in the cache. In our design, both horizontal (bits per entry) and vertical (table entries) sizes of the table are flexible which provides the flexibility when integrating the TCE mechanism into the processor design.

The rest of the paper is organized as follows. In section 2, we discuss the details of our designs. Experimental results are given in section 3. Section 4 concludes the paper.

2. DESIGN DETAILS

We will first introduce the design of the small, dedicated table used for TCE and then describe how we can reduce the hardware cost using a *way mask*.

ISLPED'03, August 25–27, 2003, Seoul, Korea.

2.1 TCE Table

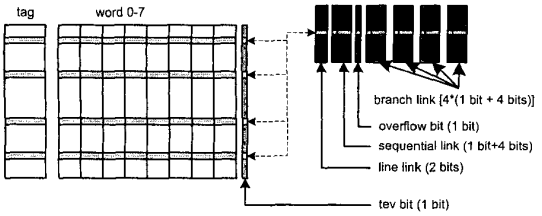

Figure 1: A small dedicated table.

Our first step is to construct a small table (TCE table) that is separated from instruction cache and stores necessary information for TCE. The entry fields are very much like those in [5] but require much less space in both dimensions. Vertically, there are fewer number of entries. Therefore, multiple cache lines are mapped to a single table entry. At anytime, a single table entry represents only one line in a group. Thus, a handshake between the table entry and the cache line is needed and it is established as the following: (1) each cache line contains a table-entry-valid (tev) bit that indicates if the information in the corresponding table entry belongs to the line, and (2) each table entry keeps a link that remembers which line in a group is currently being linked, e.g. a table that has 1/4 of the cache lines number of entries needs to keep 2 bits for each such a link (Figure 1). We will discuss the economic way of organizing table entry horizontally in section 2.2.

Next, we will elaborate the fields maintained by the small table. Suppose we have a 16KB cache with 32 bytes per cache line. There are in total 512 cache lines. Assume that we let the number of entries in the table be 1/4 of the number of cache lines, i.e., 128, the information maintained in each table entry is described as follows.

- Line link. A 2-bit line link is used to distinguish which of the 4 possible cache lines is currently linked with this table entry.
- Sequential link. A sequential link contains a valid bit and $log(A)$ bits to memorize the way number of its next cache line, where A is the cache associativity. For a direct-mapped cache, sequential link contains only one bit ($log(A) = 0$).
- Branch links. We use 4 links for branch targets per table entry. The number of such links is the instructions per cache line divided by two since there are at most half of instructions are branches per cache line [5]. Each of the link is again $log(A)$ bits plus a valid bit. Similarly, the branch link reduces to one bit when the cache is direct-mapped.
- Overflow bit. A 1-bit overflow flag is used to indicate if the corresponding cache line is pointed by some branch links in other table entries.

2.2 Way Mask

The above design can be applied to both directed mapped and set-associative caches. However, when the cache is highly associative, the hardware cost is still significant. In this section, we propose to reduce bits used in each table entry, i.e. reduce the hardware cost horizontally.

Let us first analyze the bit usage of each table entry. Using the same cache configuration as before, suppose now the set associativity is 64. Then, to remember a way number, 6 bits are needed. Therefore, each table entry has 38 bits$(2+(1+6)+1+4\times(6+1))$ in total, in which 24 are used for remembering way numbers for branches. We studied the distribution of branches in the cache by sampling the cache every 10,000 instructions. Figure 2 plots the breakdown of the percentage of cache lines that hold zero to four branches. We found that most cache lines contain zero or one branch, and on average, 68.8% lines have no branches, 16.1% lines contain only one branch.

This study suggests that many bits used for remembering way numbers are wasted. In other words, compacting the branch links would hardly degrade the overall TCE amount. We therefore propose to use a pair of *way masks* that encodes all the information of branch links per entry but use only the number of bits that equal to 2 branch links. For cache lines that contain a single branch, the mask pair itself reveals the way number. For multiple branch links, the mask pair covers only a slightly larger superset of all remembered way numbers.

The key observation in constructing the masks is that many branch links share same bits. For the same example, we compact all 4 branch links into a pair of way masks. The way mask pair consists of a "0-mask" and a "1-mask", each having 6 bits. The mask simply records on each bit position whether 0 or 1 appears in any of the branch links. Thus a "1" in the i^{th} bit of 0-mask means that 0 appears at the i^{th} bit for at least one branch link; a "0" means all the i^{th} bits are 1. The meaning of 1-mask mask is analogous. Figure 3 shows an example of a way mask constructed from two branch links, 110110 and 110011. The common bits between the two are, from left to right, the 1^{st}, 2^{nd}, and 5^{th} bit having 1's and the 3^{rd} bit having a 0. Therefore, the 1^{st}, 2^{nd}, and 5^{th} bit in 0-mask should be 0, but 1 in 1-mask. The 3^{rd} bit in 0-mask should be 1, but 0 in 1-mask. All the rest bits in both masks should be 1. Thus the 0-mask should be 001101 and the 1-mask should be 110111.

To generate way numbers from the masks, we first need to find out what are the bits that can be either 0 or 1. These bits are represented as 1's in *both* 0- and 1- masks. If there are n such bits, the number of ways we should compare is 2^n. To get those bits, we simply take the XOR of both masks and look for the 0's in the result. Replace those 0's with x's which means that they are nondeterministic. The rest bits are deterministic—either 0 or 1. They can be simply retrieved from the 1-mask by ANDing it with the XOR results. In the above example, the XOR of the 0- and 1-mask is 111010. Replacing 0's with x's yields 111x1x. Then AND this value with 1-mask, we have 110x1x. Therefore, the ways we need to compare tags are: 110010, 110011, 110110, and 110111. Instead of 64 tag comparisons, we are only comparing 4. Each table entry now contains 26 bits, as opposed to 38 bits before.

2.3 Tag Comparison Elimination

The TCE table is used for the elimination of inter- cache line tag comparisons. Similar to [3, 2, 5], intra-line tag comparisons are eliminated with compiler or hardware support.

As shown in Figure 4, TCE of current I-cache access is controlled by the output x of the TCE control logic. Meanwhile, we map a table entry c using the instruction address and evaluate two conditions: (a) current instruction is the end of a cache line; (b) current instruction is a taken branch instruction. After the current TCE decision is made, x is updated based on a and b.

x is set to true when (1) a is true, the tev bit is true, and the sequential link in the table entry is valid; (2) b is true, the tev bit is true, and its corresponding branch link is true; (3) neither a or b is true. x is set to false for rest cases: (4) a is true but the sequential link is invalid; (5) b is true but the corresponding branch link is invalid. Specifically, if the tev bit is false, both sequential link and branch links are not valid. The fact that tev bit is false indicates the corresponding table entry is either empty or used by another cache line. If it is used by another line, we reset the tev bit of that line and flush the table entry, then set the tev of the current cache line.

Accordingly, TCE is based on previously calculated x. If x is true, tag comparisons are eliminated for (1)(3) and (2) if the branch

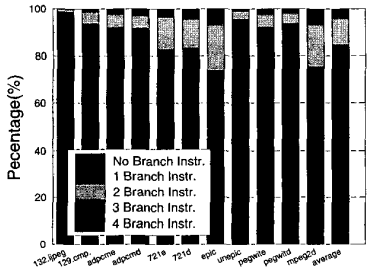

Figure 2: Branch Distribution in I-Cache.

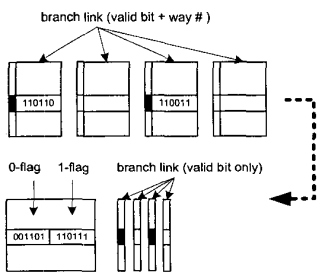

Figure 3: Branch link mask.

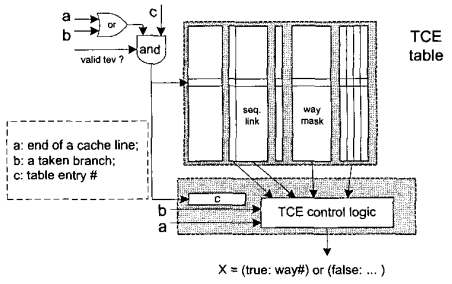

Figure 4: TCE Table Access.

mask indicates a single way number, and are reduced otherwise.

If x is false, a normal I-cache access is performed. Besides, we update the table by writing the way number of the current access back to the previously accessed table entry (remembered in c). Sequential link is updated for case (4) and the corresponding branch link and way mask are updated for case (5).

If a normal cache access results in a cache miss, the TCE table as well as all tev bits are flushed. This design simplifies the control logic although selective flushing is also possible.

2.4 Implementation Issues

Table Designs. The table used for TCE is organized as a cache structure. While its associativity can be decided independently of I-cache, we choose a direct-mapped structure because of its simplicity and energy efficiency. Note that there is no "tag comparison" for this table structure. To index the TCE table, a portion of the instruction address is extracted directly. The one-to-one correspondence between the cache line and the table entry is done through the handshaking between *tev* and the line link.

Power Model. Our power model is modified from CACTI [7] to reflect the new table design. The total energy consumed contains the energy consumed in the cache as well as in the table. Cache access energy includes decoding, tag access, data access, and data output drive. For the table access, there is no tag access portion.

$$\begin{aligned} E_{all} &= E_{cache} + E_{table} \\ &= E_{c_{tag}} + E_{c_{data}} + E_{c_{output}} + E_{c_{decoding}} + \\ &\quad E_{t_{decoding}} + E_{t_{data}} + E_{t_{output}} \end{aligned}$$

For highly associative caches, with the way mask design, the number of tags needed to be checked is dynamically decided by the mask. Thus, we dynamically model tag comparisons in the instruction cache. The dedicated small table is modeled as a cache without tag comparison and with energy for invalidation operations.

Performance Overhead. As the table is separated and used for TCE only, it has minimized interaction with the instruction cache. Cache performance is only affected when the *tev* needs to be updated. There are two possibilities: (1) when a *new* line is accessed, and it is not linked with a table entry, the *tev* needs to be set and the line indicated by the *line link* needs to reset its *tev*. That is, there are two updates in this case. (2) when a table entry whose overflow bit is set changes its owner, or when there is a cache miss, we need flush all bits in the table and all *tev* bits in the cache.

3. EXPERIMENTAL RESULTS

3.1 Settings

We implemented the proposed techniques using Simplescalar 3.0 [6] and integrate the CACTI cache energy model [7]. The base configuration is a 16K bytes, 32 bytes per cache line direct-mapped instruction cache and we vary both the cache line size and the associativity in our experiments. A set of multimedia programs are picked up from SPEC95 and Mediabench [9] benchmark suites. A decoder and an encoder are evaluated separately for most Mediabench programs.

For comparison purpose, we also implemented zero-link way memoization (WM) [5] and the history-based tag comparison scheme (HBTC) [2]. HBTC has been augmented with two enhancements: (1) removal of intra-cache line tag comparisons, termed as ITC in [2]; and (2) storing not-taken branch target addresses in BTB. We denote it as HBTC+ in the rest of the paper.

3.2 Hardware Cost

The hardware cost with respect to bit size increase is studied in this section. We first vary the number of table entries and study its impacts on TCE effectiveness. We use the base configuration and vary the table entries from 128 to 32, i.e. 1/4 to 1/16 of I-cache entries. Figure 5 reports the remaining tag comparisons as a percentage of tag comparisons of the traditional cache.

Benchmark	128 entries (%)	64 entries (%)	32 entries (%)
132.ijpeg	0.4168	0.4399	2.3746
129.cmp.	0.5660	6.8235	8.7412
adpcme	0.0015	0.0796	0.0844
adpcmd	0.0017	0.0787	0.0788
721e	8.2218	8.9392	9.0219
721d	7.9120	8.9369	9.0872
epic	9.9448	10.4661	10.4150
unepic	0.2114	0.2332	0.2299
pegwite	1.8198	3.1713	5.4390
pegwitd	1.0814	3.3089	5.7010
mpeg2d	1.0992	1.1317	1.6154
Average	2.8433	3.9645	4.7989

Figure 5: Impact of Number of Table Entries.

As expected, remaining tag comparisons increase when the number of table entries decreases. However, even with 32 table entries, the results are still very good – on average, 95.2% tag comparisons are removed.

Cache Associativity	128 entries (%)		64 entries (%)		32 entries (%)	
	WM	New	WM	New	WM	New
64-way	14.06	2.93	14.06	1.71	14.06	1.06
16-way	10.16	2.34	10.16	1.42	10.16	0.93
4-way	6.25	1.76	6.25	1.12	6.25	0.78
Directed Mapped	2.34	1.17	2.34	0.83	2.34	0.63

Figure 6: Percentage of Size Increase.

We then compare the hardware cost with WM. Figure 6 lists the percentage of size increase as a percentage of I-cache size when using WM and our design respectively. For example, using a table of 128 entries and the cache is 64-way associative, the size increase is about 2.93% while for WM, the size increases is 14.1%. The new design requires significant less hardware than WM, e.g. about 20%

of that of WM for the above configuration. Since the size increase of HBTC+ is coupled with BTBs, we leave it to the implementation and use comparable or more bits when conducting performance comparison.

3.3 Tag Comparisons Elimination

In this section, we fix the table to be 128 entries and study the TCE performance under both directed mapped and 64-way set-associative cache configurations.

Benchmark	Directed mapped (%)			64-way (%)		
	WM	HBTC+	New	WM	HBTC+	New
132.ijpeg	0.0732	4.6600	1.4875	0.1478	4.8034	0.4168
129.cmp.	0.0285	7.0014	1.1988	0.4053	7.3642	0.5660
adpcme	0.0009	6.0785	2.5543	0.0013	6.0786	0.0015
adpcmd	0.0011	7.4714	2.5939	0.0015	7.4715	0.0017
721e	5.6851	9.9765	9.0921	6.8009	10.2635	8.2218
721d	5.5761	10.0998	8.3124	6.8051	10.2350	7.9120
epic	6.2992	12.1178	10.4427	7.7984	12.2036	9.9448
unepic	0.0792	2.9239	1.5391	0.0973	2.9353	0.2114
pegwite	0.0157	5.5611	2.7188	0.0423	5.5623	1.8198
pegwitd	0.0238	5.9000	1.9958	0.0743	5.9692	1.0814
mpeg2d	0.0080	4.8205	1.8948	0.2159	5.0473	1.0992
Average	2.0355	7.0849	2.8433	1.6173	6.9646	3.9846

Figure 7: Remaining Tag Comparisons.

The remaining tag comparisons using different configurations and techniques are summarized in Figure 7. Compared to WM and on average, TCE table increases 39% and 145% tag comparisons for direct mapped cache and 64-way set associative cache respectively. It is expected since TCE table keeps a subset of the information saved in WM and makes the trade-off between hardware cost and percentage of removable tag comparisons. On the other hand, with comparable hardware as that of TCE table, HBTC increases 248% and 329% tag comparisons respectively from WM. Using the results of HBTC+ as the base, the new design reduces on average 60% and 42% tag comparisons respectively. Thus TCE table is more effective than HBTC+.

3.4 Energy Savings and Performance

Figure 8 reports energy savings and performance results of the new design. Instead of comparing our results to WM, we compare the results to the original cache configuration. The reason is that cache interactions in WM, i.e. updating links in the cache, consume significant energy and direct modeling in CACTI reports very different numbers from WM [5]. They used a different tool instead and performed several circuit level cache modifications.

From Figure 8, the new design saves around 18% of total cache energy. More appealing, we benefit low power consumption from this separated table design, i.e. on average table consumes only 0.8% of the total energy. We observe negligible performance slowdown. On average, it is less than 0.2% (Figure 8). The energy overhead due to performance slowdown is thus negligible.

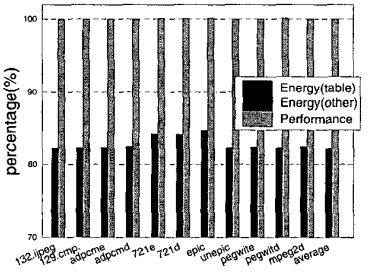

Figure 8: Energy Savings and Performance.

3.5 Branch Mask Impact

In this section, we evaluate the impact of way mask on set associative caches. We choose a 64-way set-associative I-cache, a TCE table with 128 entries and vary the cache line size from 32 bytes to 128 bytes. The results are summarized in Figure 9. From the figure, we observe better TCE results with a larger cache line size. The way mask slightly degrades the TCE performance. However, its impact decreases as the cache line size increases. At 32 bytes per line, the way mask degrades tag comparisons for about 19.9%. While at 128 bytes per cache line, the way mask degrades the tag comparisons for about 12%.

Benchmark	Cache line size					
	32 bytes (%)		64 bytes (%)		128 bytes (%)	
	4 links	1 mask	8 links	1 mask	16 links	1 mask
132.ijpeg	0.4030	1.4875	0.7744	0.7992	0.4212	0.5285
129.cmp.	0.5424	1.1988	0.5576	0.8451	0.4582	0.6255
adpcme	0.0015	2.5543	0.0011	1.1383	0.0010	0.6643
adpcmd	0.0016	2.5939	0.0013	0.8681	0.0011	0.8406
721e	8.1689	9.0921	6.1293	6.3460	5.1941	5.2351
721d	7.8857	8.3124	6.1533	6.2182	4.9261	5.1266
epic	9.9192	10.4427	7.5881	7.9031	6.6991	6.8532
unepic	0.2056	1.5391	0.5532	0.8632	0.4501	0.3478
pegwite	1.8175	2.7188	1.1955	1.4974	0.8407	1.0507
pegwitd	1.0540	1.9958	0.9533	1.2040	0.6906	0.8174
mpeg2d	1.0797	1.8948	0.9924	1.3219	0.8461	0.9014
Average	2.8254	3.9846	2.2636	2.6368	1.8662	2.0901

Figure 9: Branch Mask Impact.

4. CONCLUSIONS

Two design improvements are proposed in this paper to reduce the hardware cost of TCE in the design of a low energy I-cache. A small table is designed for TCE only and its size is flexible both horizontally and vertically. Compared to previous techniques, the proposed design has very few interactions with I-cache and no interactions with BTB, the performance and energy overhead is thus minimized. Our experiments show that a TCE table whose size is 20% of that used in way memoization eliminates 95.2% of tag comparisons in a 64-way set associative I-cache.

5. REFERENCES

[1] M.Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi and K. Roy, "Reducing Set-Associative Cache Energy via Way-Predication and Selective Direct-Mapping," *MICRO'34*, 2001.

[2] K.Inoue, V.G. Moshnyaga, and K. Murakami, "A History Based I-cache for low-Energy Multimedia Applications," *ISLPED'02*, pages 148-153, Monterey, CA, 2002.

[3] R. Panwar and D. Rennels, "Reducing the Frequency of Tag Compares for Low Power I-cache Design," *ISLPED'95*, pages 57-62, 1995.

[4] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," *ISCA-24*, 1995.

[5] A. Ma, M. Zhang, and K. Asanovic, "Way Memoization to Reduce Fetch Energy in Instruction Cache," *Workshop on Complexity-Effective Design, in conjuction with ISCA-28*, June 2001.

[6] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," TR-CS-1342, University of Wisconsin-Madison, June 1997.

[7] P.Shivakumar and N.P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," TR-WRL-2001-2, Dec 2001.

[8] K.C. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE MICRO, 16(2):28-40, April 1996.

[9] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *MICRO*, pages 330-335, 1997.

[10] J. Kin, M. Gupta and W.H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *MICRO*, 1997.

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