A History-Based I-Cashe for Low-Energy Multimedia Applications

Inoue, Koji
Dept. of Elec. Eng. and Computer Science Fukuoka University

Moshnyaga, Vasily G.
Dept. of Elec. Eng. and Computer Science Fukuoka University

Murakami, Kazuaki
Dept. of Informatics Kyushu University

出版情報：Proc. of 2002 International Symposium on Low Power Electronics and Design (ISLPED'02), pp.148-153, 2002-08. Association for Computing Machinery
バージョン：
権利関係：

A History-Based I-Cache for Low-Energy Multimedia Applications

Koji Inoue
Dept. of Elec. Eng. and Computer Science
Fukuoka University
8-19-1 Nanakuma, Jonan-ku,
Fukuoka 814-0180 JAPAN
inoue@tl.fukuoka-u.ac.jp

V.G. Moshnyaga
Dept. of Elec. Eng. and Computer Science
Fukuoka University
8-19-1 Nanakuma, Jonan-ku,
Fukuoka 814-0180 JAPAN
vasily@fukuoka-u.ac.jp

K. Murakami
Dept. of Informatics
Kyushu University
6-1 Kasuga-Koen, Kasuga,
Fukuoka 816-8580 JAPAN
murakami@i.kyushu-u.ac.jp

ABSTRACT

This paper proposes a history-based tag-comparison scheme for reducing energy consumption of direct-mapped instruction caches. The proposed cache efficiently exploits program-execution footprints recorded in the Branch Target Buffer (BTB), and attempts to detect and eliminate unnecessary tag checks at run time. Simulation results show that our approach can eliminate up to 95% of tag checks, saving the cache energy by 17%, while affecting the processor performance by only 0.2%.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; C.1 [**Computer Systems Organization**]: Processor Architectures

General Terms

Design

1. INTRODUCTION

1.1 Motivation

As the popularity of multimedia applications increases, the workload of microprocessors employed in battery-operated computing devices such as laptop and hand-held computers is dominated by media programs (e.g., video, 3-D graphics, audio, and speech). In order to satisfy the high performance requirements imposed by these workloads, under the growing speed discrepancy between microprocessors and external memory, there is a tendency to use larger and larger on-chip caches. However, their large load capacitance accounts for significant energy dissipation. Instruction caches (or I-caches) particularly affects total energy consumption due to their high access frequency. The Strong-ARM SA110, for example, dissipates 25% of its total energy in the I-cache [11]. Therefore, reducing the I-cache energy is critical for lowering the overall energy consumption of microprocessors.

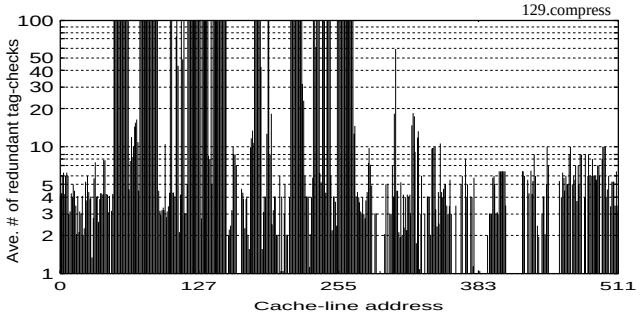

Figure 1: Redundant tag-check distribution

In this paper, we focus on direct-mapped instruction caches which employ the simplest control and exhibit the fastest access time in comparison to set-associative organization. On an I-cache access, a tag check and data read are performed in parallel. The tag check is required to determine whether the current reference hits the cache. Thus, we can consider at least two energy sources: the energy used for the tag check and that for the data read. Cache subbanking is a well known approach to reducing the data-read energy, in which the data SRAM array is partitioned into several subbanks and only one subbank including the target instruction is activated [6]. However, since the whole tag is required for the tag check, this kind of partitioning approach can not be applied to the tag SRAM array. Therefore, we need to devise another approach to reducing the tag-check energy.

Consider a direct-mapped I-cache behavior. Only when a cache miss takes place, some instructions are evicted from the cache to fill the missed instructions. Thus, once an instruction is loaded into the cache, it stays there at least until the next cache miss occurs. Suppose an instruction is executed repeatedly, e.g., loop executions. At the first reference of the instruction, the tag check has to be performed to examine its presence in the cache. However, at and after the second reference, if no cache miss has occurred, that target instruction is guaranteed to currently reside in the cache. In this case, we can detect a cache hit without performing the tag check. We refer to the period between two consecutive cache misses as a *cache-miss interval*. When an instruction

Fukuoka Univ. Technical Report
(appears in ISLPED'02)

is accessed repeatedly during a cache-miss interval, the tag check for the first reference has to be performed, but after that it is not unnecessary. However, conventional caches perform the tag check at every cache access, therefore a large amount of energy is unnecessarily dissipated.

Figure 1 shows the average number of unnecessary tag checks performed at each cache line (or block)[1], i.e., the average access count to each cache line per cache-miss interval. As we can see, the number of redundant tag-checks is more than four for the majority of the cache lines, with some lines having as many as one hundred. If we detect and eliminate the unnecessary tag checks at run time, then we can save significant amounts of energy.

1.2 Related Research

Several architectural approaches to reducing the frequency of I-cache tag-checks have been proposed. A popular one is to augment the I-cache with a small Level-0 memory (or L0-cache) which stores the frequently executed instructions [3][4][8]. Since the I-cache is accessed only on L0-cache misses, the number of I-cache tag-checks is reduced.

Panwar et al. [10] proposed to perform tag-checks only for instructions that belong to different cache lines. If two instructions i and j reside in the same cache line, and instruction j is executed immediately after i, then the tag check for j can be omitted. Although this method can be controlled via the Program Counter, it requires tagging each branch instruction to indicate whether a branch transfers control to an nninstruction outside the current line. We refer to this technique as *interline tag-comparison or ITC*.

Witchel et al. [13] presented a *Direct Addressed* scheme that allows software to access cache data without hardware cache tag checks. The key idea is to store the tag-check results in *DA-register file*, and then reuse them based on compiler supports. If the DA-register file is valid, the compiler analyses its content to ensure that the current access is to the same cache line as an earlier access. The main shortcoming of this method is the special compilation scheme that might affect code compatibility.

Ma et al. [9] suggested a dynamic way-memoization for eliminating tag checks. The idea is to record within I-cache both the tag check results (*links*) and the *valid bits* to show whether the link is correct. If the link is valid, it is followed to fetch the next instruction without tag checks. This approach can be applied regardless of cache associativity but implies hardware overhead in the cache for keeping the branch links.

1.3 Contribution

This paper proposes an alternative approach for detecting and removing unnecessary tag-checks in I-cache: –"*history-based tag-comparison (HBTC)*[2]. Unlike existing techniques, the HBTC cache exploits execution footprints recorded in the Branch Target Buffer (BTB) for reducing the frequency of tag checks. In contrast to [10], our technique is more feasible because it can omit redundant tag checks for instructions which belong to different cache lines. Moreover, it requires neither a large timing-area overhead in the cache nor additional compiler supports. The proposed approach is orthogonal to, and can be used in conjunction with, other cache energy reduction techniques such as L0-cache. We present the HBTC implementation scheme for direct-mapped I-caches, and evaluate performance/energy efficiency by using various benchmark programs.

The paper is organized as follows: The next section presents the HBTC architecture. Section 3 outlines the simulation environment and the experimental results. Conclusions are drawn in Section 4.

[1]Usually, tag checks are performed at cache-line granularity.
[2]In [5], the concept of the HBTC cache and rough evaluation results in terms of the total tag-check count performed have been reported. This paper describes an extended architecture which does not require the rollback operation for execution footprints on branch mis-prediction. In addition, the total energy consumption including BTB overhead and the impact on processor performance are evaluated.

2. THE HISTORY-BASED TAG-COMPARISON CACHE

2.1 Main Idea

At and after the second reference to the same instruction in a cache-miss interval, performing tag checks is unnecessary, as explained in Section 1.1. In other words, we can omit the tag check if: (i) the target instruction has been referenced before, and (ii) no cache misses have occurred since the previous reference to the target instruction.

In modern microprocessors, a Branch Target Buffer (BTB) is employed to obtain a branch-target address as soon as possible. In order to detect the above two conditions, the HBTC cache records execution footprints in the BTB. An execution footprint indicates whether or not the target-instruction block (or the fall-through-instruction block) associated with the branch currently resides in the I-cache. The execution footprint is recorded after all instructions in the corresponding block are referenced without any cache misses. Whenever a cache miss takes place, all the recorded footprints are erased (or invalidated). Therefore, when the microprocessor fetches an instruction block, if the corresponding footprint is valid, the two conditions for dynamic tag-check omission are satisfied. In this case, we can omit tag checks for all instructions in the block.

This approach separates the hardware for dynamic tag-check omission from the I-cache structure, therefore it does not degrade the cache-core-access time, energy consumption, or transistor budget. In addition, since we exploit the BTB employed for branch prediction, our scheme can be implemented with small hardware overhead.

2.2 Organization

The proposed HBTC cache combines four components: BTB, Previous Branch Address register (PBAreg), a mode controller, and I-cache, as shown in Figure 2. Unlike existing designs, the following 1-bit flags associated with the branch are added to each BTB entry.

- Flag **T**: This is the execution footprint of the branch-target-instruction block whose start address is indicated by the target address.
- Flag **F**: This is the execution footprint of the fall-through-instruction block whose start address is indicated by the fall-through address of the branch.

The end address of both the branch-target- and the fall-through-instruction block are given by the other branch-instruction addresses which have already been registered in

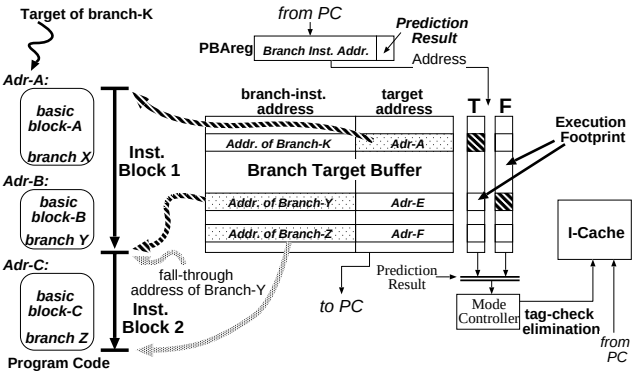

Figure 2: The HBTC implementation scheme

the BTB. A validated footprint ensures that the corresponding instruction block, i.e., all instructions from the branch-target (or fall-through) address to the end address, certainly resides in the cache.

For example, suppose *branch-X* of the basic block A (see Figure 2) has not been executed as a taken branch, and *branch-K, -Y, and -Z* have been registered in the BTB. In this scenario, the **instruction block 1** is the target-instruction block of *branch-K*, because its start address is specified by the target address (*Adr-A*) and the end address is given by *branch-Y*'s address registered in the BTB. On the other hand, the **instruction block 2** is the fall-through-instruction block of *branch-Y*, and its end address is given by *branch-Z*'s address. The size of the instruction block may be changed at run time. For instance, when *branch-X* is executed as taken, it will be registered in the BTB. As a result, the end address of the **instruction block 1** becomes *branch-X*'s address. Thus, the fall-through-instruction block of *branch-X*, whose start address is *Adr-B* and the end address is *branch-Y*'s address, is created.

The PBAreg stores the previous-branch-instruction address and the result of branch prediction (taken or not-taken), and is used as an address register when the BTB is accessed for leaving the execution footprints. The mode controller manages the HBTC behavior. The details of the HBTC operation are explained in Section 2.3.

2.3 Operation

The HBTC cache has the following three operation modes, one of which is activated by the mode controller:

- Normal mode (Nmode): The cache behaves as a conventional I-cache, so that the tag check is performed at every cache access.
- Omitting mode (Omode): The cache omits the tag check in I-cache accesses.
- Tracing mode (Tmode): The cache performs tag checks as in the Nmode, and also set the execution footprints (this operation is not performed in the Nmode).

Figure 3 displays the operation transitions. On every BTB hit, the HBTC cache works as follows:

1. Regardless of the current operation mode, both the **T** and **F** flags associated with the BTB-hit entry are read in parallel.

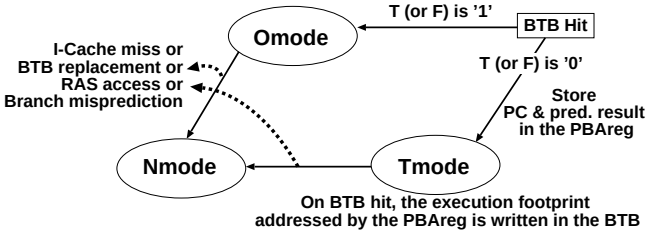

Figure 3: Operation-Mode Transition

2. Based on the branch-prediction result, one of them (**T** on taken or **F** on not-taken) is selected.
3. If the selected execution footprint is '1' (or valid), the operation mode is changed to the Omode. Otherwise, the Tmode is activated, and the current PC (branch-instruction address) and the branch-prediction result (taken or not-taken) are stored in the PBAreg.

Whenever a cache miss takes place, the operation mode is transited to the Nmode. Therefore, a BTB hit in the Tmode means that there have been no cache misses since the previous BTB hit. In other words, the whole instruction block, whose start address is related to the PBAreg and whose end address is specified by the current branch-instruction address (or PC), resides in the I-cache. Note that the access information for the previous BTB hit has been kept in the PBAreg. Thus, when a BTB hit occurs in the Tmode, the execution footprint indexed by the PBAreg is validated (set to '1').

When an execution footprint recorded in the BTB loses accurate information for the presence of corresponding instruction block in the cache, it has to be invalidated to ensure the correct program execution. To simplify the mode control, we have employed a conservative scheme that invalidates all the execution footprints in the BTB. In addition, the cache switches the operation mode to the Nmode. The conditions for footprint invalidation are:

- I-cache miss: Because the cache-line replacement evicts an instruction block (or a part of it), the execution footprints have to be invalidated according to the victim line.
- BTB replacement: When a BTB entry is replaced, we might lose the end-address information of an instruction block, which has a valid execution footprint.

To avoid complex hardware control, we assume that the cache operates in the Nmode whenever the branch-target address is provided by the RAS (Return Address Stack), or a branch mis-prediction is detected (footprint invalidation is not performed).

2.4 Operation Example

Figure 4 (A) shows the execution example of a loop. The solid and broken arrows represent the control flow of the execution. Figure 4 (B) depicts the behavior of the extended BTB. A number and a capital letter pair denotes the BTB look-up/update time. For instance, $1-C$ indicates the time when the branch-C (the branch addressed by C) is executed in iteration 1. It is assumed that the branch prediction is perfect, and that the initial operation mode is the Nmode

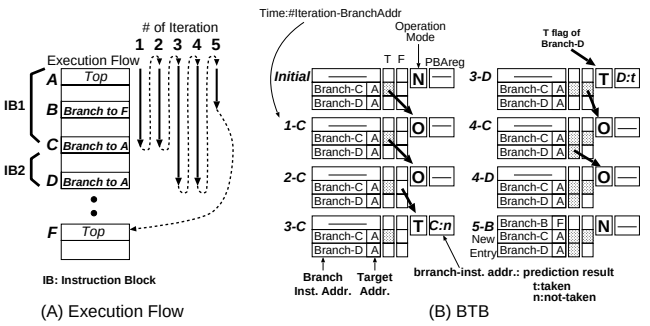

Figure 4: Operation Example

(Normal-Mode). In addition, we assume the branch-C in the BTB has the valid flag **T** at the initial state. In the first iteration, tag checks must be performed, as we operate in the Nmode. After that, the HBTC cache works as follows:

1-C : The branch-C is predicted as a taken branch, and the associated flag **T** is read from the BTB. This flag indicates whether the associated target-instruction block (i.e., **IB1** in the figure) resides in the cache. Since the flag is '1', the cache enters the Omode (Omitting-Mode). Therefore, tag checks for the **IB1** in the second iteration are omitted.

2-C : The branch-C is predicted as a taken branch again. Since the flag **T** read from the BTB is '1', the cache works in the Omode. Thus, during the third iteration, all instructions in the **IB1** can be fetched without tag checks.

3-C : The prediction result for the branch-C is "not-taken", and the associated flag **F** is read from the BTB. That flag indicates whether the fall-through-instruction block (i.e., **IB2** in the figure) resides in the cache. Since the read flag is '0', the operation mode is changed to the Tmode (Tracing-Mode). Thus, tag checks for the **IB2** in the third iteration are performed. In addition, the branch-instruction address (or PC) C and the prediction result "$not-taken$" are stored in the PBAreg.

3-D : The flag **T** associated with branch-D is 0, so that the operation mode is not changed (Tmode). In addition, this is a BTB hit in the Tmode. Therefore, the flag **F** pointed by the PBAreg ($AddressC : not-taken$) is set to '1' in order to leave the execution footprint of the **IB2**. The BTB access information ($AddressD : taken$) is then stored in the PBAreg.

4-C : The flag **F** of branch-C is '1', so that the operation mode transits to the Omode. Thus, the tag checks for the fall-through-instruction block denoted as **IB2** are omitted. Moreover, since this is a BTB hit in the Tmode, the flag **T** addressed by the PBAreg ($AddressD : taken$) is set to '1'.

4-D : The flag **T** of branch-D is '1', so that the cache works in the Omode and tag checks are not performed during the fifth iteration.

5-B : This is the first taken operation for the branch-B, hence branch-B is registered in the BTB. In this case, a branch-miss prediction should be detected because the microprocessor can not obtain the branch-target address from the BTB. The operation mode is thus changed to the Nmode, and tag checks are resumed.

2.5 Performance/Energy Overhead

The proposed approach allows us to eliminate all tag checks in the Omode. On the other hand, accessing the execution footprints produces an energy overhead for the BTB. In addition, if the operation enters the Tmode, saving the address in the PBAreg produces a small energy overhead.

From the performance point of view, controlling the execution footprints may lead to some performance degradation. In conventional microprocessors, the BTB is accessed at every clock cycle for branch prediction. Reading the execution footprints can be done in parallel with the BTB access. However, writing the execution footprints requires one processor-stall cycle because it is performed on another BTB entry for branch prediction. This performance drawback appears at every BTB hit in the Tmode. Also, the execution-footprint invalidation caused by cache misses or BTB replacements produces one stall cycle due to the BTB access conflict.

3. EVALUATION

3.1 Energy Modeling

The total energy dissipated by the HBTC cache can be expressed by the following equations:

$$E_{TOTAL} = E_{CACHE} + E_{BTBadd}, \quad (1)$$

where, E_{CACHE} is the I-cache energy, and E_{BTBadd} is the additional energy dissipated by the BTB extension (the energy of the conventional BTB is not included).

$$E_{CACHE} = E_{tag} + E_{data} + E_{output} + E_{dec}, \quad (2)$$

where E_{tag} and E_{data} are the energy dissipated by the tag array and the data array, respectively; E_{output} the energy required for driving output buses; E_{dec} the energy of the address decoder. In this paper, we do not consider E_{dec}, because it is usually three orders of magnitude smaller than the other components [8].

$$E_{BTBadd} = E_{BTBef} + E_{BTBlogic}, \quad (3)$$

where E_{BTBef} is the energy dissipated on reading or writing the execution footprints; $E_{BTBlogic}$ the energy consumed by the mode controller and the PBAreg. Since the PBAreg is small and the mode controller is simple, $E_{BTBlogic}$ can be ignored. We also ignore the energy for execution-footprint invalidations, since they occur very rarely, i.e. when an instruction-cache miss or a BTB replacement occurs.

3.2 Simulation Environment

We used the SimpleScalar simulator from [2] to evaluate the proposed architecture. The cache energy was computed based on 0.8 μm CMOS using models presented in [6]. The load capacitance of each node has been taken from [7] [12]. We experimented with the 16 KB direct-mapped I-cache, 32 B cache-line size, assuming that the data array of the cache is partitioned into 4 subbanks. For the other processor parameters, we used the default values of the SimpleScalar simulator.

The cache was simulated on two SPEC benchmark programs: *129.compress* (text file compression) and *132.ijpeg* (jpeg image compression), and four Mediabench programs[1]: *adpcm* (adaptive differential pulse code modulation), *mpeg2*

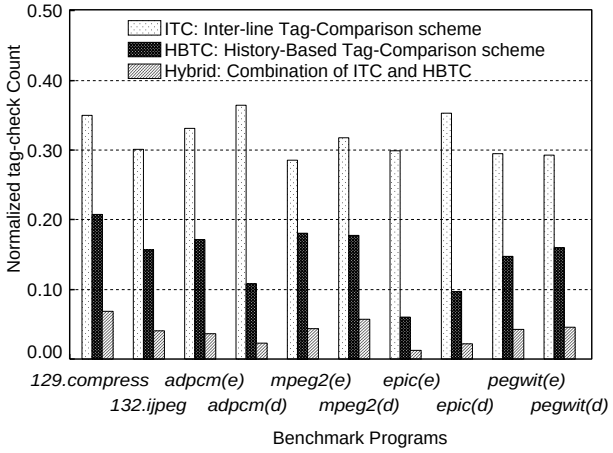

Figure 5: Tag-check counts

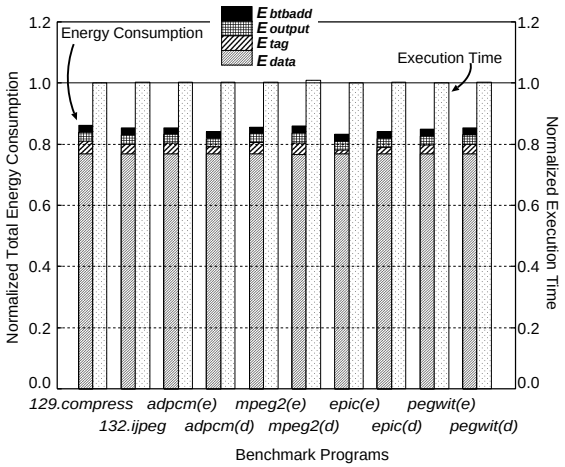

Figure 6: Energy and Performance

(MPEG-2 video bitstream compression), *epic* (efficient Pyramid image coder), and *pegwit* (public key encryption and authentication). For each of the Mediabench programs, an encoder and a decoder were evaluated separately. Thus, ten benchmark programs were used in total.

3.3 Results

3.3.1 Tag-Check Count

Figure 5 shows the total amount of tag-checks performed on the benchmarks. The label (e) denotes encoders, and (d) denotes decoders. All the results are normalized to the values produced by the 16 KB conventional cache. The 'ITC' bars show the results of the ITC approach explained in Section 1.2, and the 'Hybrid' bars show the results of the combined approach which includes both the ITC and the HBTC.

We observe that the ITC approach has almost a constant reduction factor along the simulated programs (around 0.33) because it focuses mainly on sequential accesses inherent in programs. In contrast, the results obtained by the proposed approach vary over the programs. For all the programs, the HBTC cache produces better results than the ITC approach, allowing us to reduce the amount of tag checks up to 95% (*epic(e)*). The reason is that since the HBTC cache effectively treats the branch operation, the cache can detect and eliminate large amount of unnecessary tag checks due to many repetitive executions of instructions.

The hybrid model of the ITC and the HBTC approaches shows the best results, removing 85% to 98% of total tag checks. This is because it combines the benefits of the ITC for sequential accesses and that of the HBTC for dealing with branches.

3.3.2 Performance/Energy Efficiency

Figure 6 reports energy consumption of the HBTC cache and its break down. All the results are normalized to the results of the conventional organization. Although the energy model used in this paper does not take into account the energy consumed in sense amplifiers, we believe that their consideration would not change the picture. Since the HBTC scheme completely avoids the tag-array accesses in the Omode, the energy dissipated by sense amplifiers in this mode is also eliminated. As the figure shows, the HBTC cache reduces the total energy by 8% – 17% even though there is a small energy overhead for the BTB accesses.

Now consider the bars, which depict the normalized execution time in Figure 6. As we see, the performance degradation is less than 1% for all benchmarks. This small performance overhead is caused by the processor stalls due to the simultaneous BTB accesses by the processor for branch prediction and the footprint up-date in the HBTC approach. However, these BTB conflicts can be easily alleviated by adding a special port for treating the execution footprints.

3.3.3 Footprint-Invalidation Penalty

So far, we have assumed that the footprint invalidation can be performed in one processor-clock cycle. In practice, the invalidation penalty largely depends on the BTB implementation. Figure 7 shows the performance overhead caused by the HBTC approach when the invalidation penalty changes from 1 to 32 cycles. We observe that the performance degradation is very small even when the invalidation penalty extends to 4 clock cycles. We analyzed the invalidations and found that over 98% of them are caused by cache misses, and less than 2% by BTB replacements. When a cache miss occurs, the microprocessor waits to fetch new instructions until the missed instruction is loaded from the next-level memory into the I-cache. The footprint invalidation caused by the cache miss is performed in parallel with the cache-line replacement. Therefore, the footprint invalidation penalty can be hidden if it is smaller than the cache-miss penalty. In the experiment, we assumed a cache-miss penalty of 6 clock cycles.

3.3.4 Effects of Cache Size and BTB Associativity

To evaluate the impact of the cache size and the BTB associativity, we simulated caches varying in size from 4 KB to 64 KB (assuming a BTB associativity of 4), and the BTB associativity from 1 to 32 (assuming a cache size of 16 KB). Figure 8 and 9 show the results, respectively. The results are normalized to the value of a conventional cache with the same cache size and BTB associativity.

As we see, the energy reduction of the proposed approach increases as the cache grows in size, because the frequency of footprint invalidations is reduced. On the other hand, although increasing the BTB associativity has the same effect

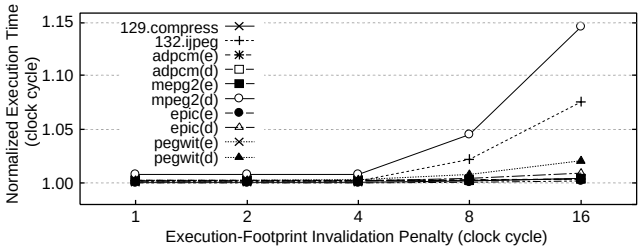

Figure 7: Effect of Footprint-Invalidation Penalty

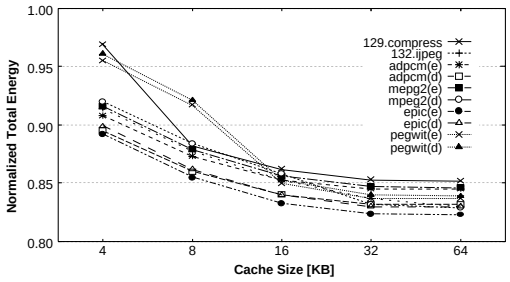

Figure 8: I-cache energy vs. cache size

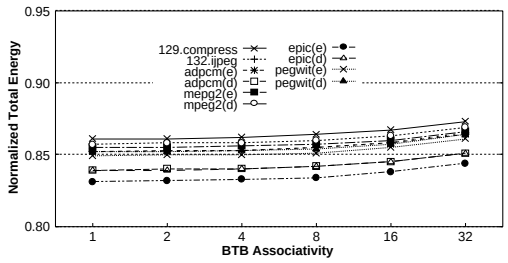

Figure 9: I-cache energy vs. BTB associativity

on the invalidation frequency (i.e., BTB replacements could be avoided), the energy efficiency obtained by the HBTC approach degradeds when the associativity is changed from 1 to 32. These results can be understood from the fact that almost all invalidations are caused by I-cache misses rather than BTB replacements. The total number of execution footprints to be read on each BTB access increases with an increase in the BTB associativity. Since the negative effect caused by the footprint reads exceeds the energy reduction obtained by reducing the invalidation frequency, increasing the BTB associativity worsens the HBTC energy efficiency.

4. CONCLUSIONS

In this paper, we have proposed a history-based tag-comparison (HBTC) cache for reducing the energy consumption of direct-mapped I-caches. The cache dynamically utilizes the BTB contents to determine whether the target instructions are within the I-cache without performing tag checks. The HBTC approach exploits the repeated execution of loop instructions, and hence is suitable for media programs.

Simulations showed that our cache can reduce the total amount of tag checks up to 95%, and save up to 17% of the energy consumed by the direct-mapped I-cache, with less than 0.2% performance degradation. In addition, it was observed that combining the ITC approach [10] and the proposed approach makes a significant tag-check reduction. Therefore, we conclude that this combination is very promising for reducing the frequency of the I-cache tag checks.

Acknowledgments

We thank Hiroto Yasuura who gave us advice. This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, for Scientific Research (A), 12358002, 13308015, and for Encouragement of Young Scientists (A), 14702064.

5. REFERENCES

[1] Mediabench. In *URL:http://www.cs.ucla.edu/l̃eec/mediabench/*.

[2] Simplescalar simulation tools for microprocessor and system evaluation. In *URL:http://www.simplescalar.org/*.

[3] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *Proc. of the 1999 Int. Conf. on Computer Design: VLSI in Computers & Processors*, pages 378–383, Oct. 1999.

[4] K. Ghose and M. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-segmentation. In *Proc. of the 1999 Int. Symp. on Low Power Electronics and Design*, pages 70–75, Oct. 1999.

[5] K. Inoue and K. Murakami. Instruction cache architecture exploiting program execution footprints. In *International Symposium on High-Performance Computer Architecture, Work-in-progress session (included in the CD proceedings)*, Feb. 2001.

[6] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proc. of the 1997 Int. Symp. on Low Power Electronics and Design*, pages 143–148, Aug. 1997.

[7] M. Kamble and K. Ghose. Energy-efficiency of vlsi caches: A comparative study. In *Proc. of the 10th Int. Conf. on VLSI Design*, pages 261–267, Jan. 1997.

[8] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proc. of the 30th Int. Symp. on Microarchitecture*, pages 184–193, Dec. 1997.

[9] A. Ma, M. Zhan, and K. Asanović. Way memorization to reduce fetch energy in instruction caches. In *ISCA Workshop on Complexity Effective Design*, July 2001.

[10] R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power i-cache design. In *Proc. of the 1995 Int. Symp. on Low Power Electronics and Design*, pages 57–62, Aug. 1995.

[11] S. Segars. Low power desin techniques for microprocessors. In *ISSCC Tutorial*, Feb. 2001.

[12] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. In *WRL Research Report 93/5*, July 1994.

[13] E. Witchel, S. Larsen, C. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proc. of the 34th Int. Symp. on Microarchitecture*, Dec. 2001.