Dynamic Tag-Check Omission: A Low-Power Instruction Cache Architecture Exploiting Execution Footprints

Inoue, Koji
Department of Computer Science and Computer Science, Fukuoka University

Moshnyaga, Vasily G.
Department of Computer Science and Computer Science, Fukuoka University

Murakami, Kazuaki
Department of Informatics, Kyushu University

出版情報：Proc. of the Workshop on Power Aware Computer Systems, pp.15-22, 2002-02. the Workshop on Power Aware Computer Systems
バージョン：
権利関係：

Dynamic Tag-Check Omission: A Low Power Instruction Cache Architecture Exploiting Execution Footprints

Koji Inoue†, Vasily Moshnyaga†, and Kazuaki Murakami‡

†Department of Computer Science and Computer Science, Fukuoka University
8-19-1 Nanakuma, Jonan-ku, Fukuoka 814-0180 JAPAN
{inoue, vasily}@tl.fukuoka-u.ac.jp

‡Department of Informatics, Kyushu University
6-1 Kasuga-Koen, Kasuga, Fukuoka 816-8580 JAPAN
murakami@c.csce.kyushu-u.ac.jp

Abstract

This paper proposes an architecture for low-power direct-mapped instruction caches, called "*history-based tag-comparison (HBTC) cache*". The HBTC cache attempts to detect and omit unnecessary tag checks at run time. Execution footprints are recorded in an extended BTB (Branch Target Buffer), and are used to know the cache residence of target instructions before starting cache access. In our simulation, it is observed that our approach can reduce the total count of tag checks by 90 %, resulting in 15 % of cache-energy reduction, with less than 0.5 % performance degradation.

key words: cache, low power, tag check, dynamic optimization, BTB

1 Introduction

On-chip caches have been playing an important role in achieving high performance. In particular, instruction caches give a great impact on processor performance because one or more instructions have to be issued on every clock cycle. In other words, from energy point of view, instruction caches consume a lot of energy. Therefore, it is strongly required to reduce the energy consumption for instruction-cache accesses.

On a conventional cache access, tag checks and data read are performed in parallel. Thus, the total energy consumed for a cache access consists of two factors: the energy for tag checks and that for data read. In conventional caches, the height (or the total number of word-lines) of tag memory and that of data memory are equal, but not for the width (or the total number of bit-lines). The tag-memory width depends on the tag size, while the data-memory width depends on the cache-line size. Usually, the tag size is much smaller than the cache-line size. For example, in the case of a 16 KB direct-mapped cache having 32-byte lines, the cache-line size is 256 bits (32×8), while the tag size is 18 bits (32 - 9bit index - 5bit offset). Thus, the total cache energy is dominated by data-memory accesses.

Cache subbanking is one of the approaches to reducing the data-memory-access energy. The data-memory array is partitioned into several subbanks, and only one subbank including the target data is activated [6]. Figure 1 depicts the breakdown of cache-access energy of a 16 KB direct-mapped cache with the varied number of subbanks. We have calculated the energy based on the Kamble's model [6]. All the results are normalized to a conventional configuration denoted as "1(8)". It is clear from the figure that increasing the number of subbanks makes significant reduction for data-memory energy. Since the tag-memory energy is maintained, however, it becomes a significant factor. If the number of subbanks is 8, about 30 % and 50 % of total energy are dissipated by the tag memory where the word size is 32 bits and 64 bits, respectively.

In this paper, we focus on the energy consumed for tag checks, and propose an architecture for low-power direct-mapped instruction caches, called "*history-based tag-comparison (HBTC) cache*". The basic idea of the HBTC cache has been introduced in [4]. The HBTC cache attempts to detect and omit unnecessary tag checks at run time. When an instruction block is referenced without causing any cache miss, a corresponding execution footprint is recorded in an extended BTB (Branch Target Buffer). All execution footprints are erased whenever a cache miss takes place, because the instruction block (or a part of the instruction block) might be evicted from the cache. The execution footprint indicates whether the instruction block currently resides in the cache. At and after the next execution

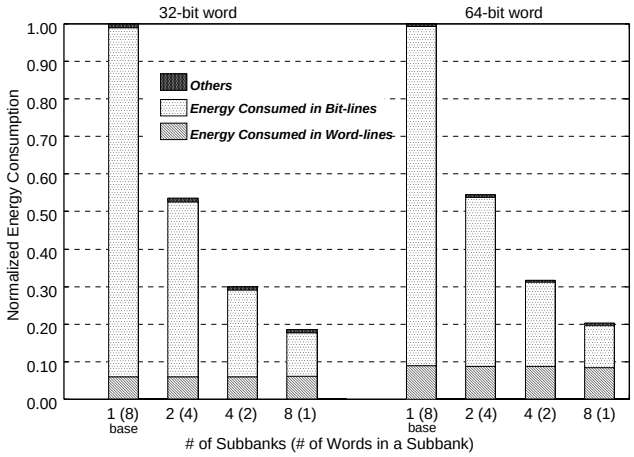

Figure 1: Effect of tag-check energy

of that instruction block, if the execution footprint is detected, all tag checks are omitted. In our simulation, it has been observed that our approach can reduce the total count of tag checks by 90 %, resulting in 15 % of cache-energy reduction, with less than 0.5 % performance degradation.

The rest of this paper is organized as follows. Section 2 shows related work, and explains the detail of another technique proposed in [11] to omit tag checks as a comparative approach. Section 3 presents the concept and mechanism of the HBTC cache. Section 4 reports evaluation results for performance/energy efficiency of our approach, and Section 5 concludes this paper.

2 Related Work

A technique to reduce the frequency of tag checks has been proposed [11]. If successively executed instructions i and j reside in the same cache line, then we can omit the tag check for instruction j. Namely, the cache proposed in [11] performs tag checks only when i and j reside in different cache lines. We call the cache *interline tag-comparison cache (ITC cache)*. This kind of traditional technique has been employed at commercial microprocessors, e.g., ARMs. The ITC cache detects unnecessary tag checks by monitoring program counter (PC). Against to the ITC cache, our approach exploits an extended BTB in order to record instruction-access history, and can omit unnecessary tag checks even if successive instructions reside in different cache lines. In Section 4.2.1, we compare our approach with the ITC cache.

Direct Addressing (DA) is another scheme to omit tag checks [13]. In DA, previous tag-check results are recorded in the DA register, and are reused for future cache accesses. The DA register is controlled by compiler, whereas our HBTC cache does not need any software support. Note that the ITC cache and the DA scheme can be used for both instruction caches and data caches, while our HBTC cache can be used only for direct-mapped instruction caches. The extension to set-associative caches is discussed in Section 5. Ma et al. [9] have proposed a dynamic approach to omitting tag checks. In their approach, cache line structure is extended for recording valid links, and a branch-link is implemented per two instructions. Their approach can be applied regardless of cache associativity. The HBTC cache is another alternative to implement their approach on direct-mapped instruction caches, and can be organized with smaller hardware overhead. This is because the HBTC cache records 1-bit cache-residence information for each instruction block, which could be larger than the cache line.

The S-cache has also been proposed in [11]. The S-cache is a small added memory to the L1 cache, and has statically allocated address space. No cache replacements occur in the S-cache. Therefore, S-cache accesses can be done without tag checks. The scratchpad-memory [10], the loop-cache [3], and the decompressor-memory [5] also employ this kind of a small memory, and have the same effect as the S-cache. In the scratchpad-memory and the loop-cache, application programs are analyzed statically, and compiler allocates well executed instructions to the small memory. For the S-cache and the decompressor-memory, prior simulations using input-data set are required to optimize the code allocation. They are differ from ours in two aspects. First, these caches require static analysis. Second, the cache has to be separated to a dynamically allocated memory space (i.e., main cache) and a statically allocated memory space (i.e., the small cache). The HBTC cache does not require these arrangements.

The filter cache [8] achieves low power consumption by adding a very small L0-cache between the processor and the L1-cache. The advantage of the L0-cache largely depends on how many memory references hit the L0-cache. Block buffering can achieve the same effect of the filter cache [6]. Bellas et al. [2] proposed a run-time cache-management technique to allocate the most frequently executed instruction-blocks to the L0-cache. On L0-cache hits, accessing both the tag-memory and data-memory of L1-cache is avoided, so that of cause tag checks at L1-cache do not performed. However, on L0-cache misses, the L1-cache is accessed with conventional behavior (tag checks are re-

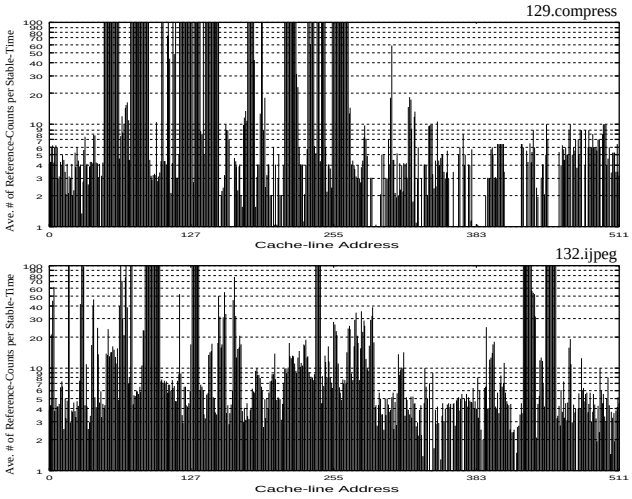

Figure 2: Opportunity of tag-check omission

quired). Our approach can be used in conjunction with the L0-caches in order to avoid L1-cache tag checks.

3 History-Based Tag-Comparison Cache

3.1 Concept

On an access to a direct-mapped cache, a tag check is performed to determine whether the memory reference hits the cache. For almost all programs, instruction caches can achieve higher hit rates. In other words, the state (or contents) of the instruction cache is rarely changed. Only when a cache miss takes place, the state of instruction cache is changed by filling the missed instruction (and some instructions residing in the same cache line with the missed instruction). Therefore, if an instruction is referenced once, it stays in the cache at least until the next cache miss occurs. We refer the period between a cache miss to the next cache miss as a *stable-time*.

Here, we consider where an instruction is executed repeatedly. At the first reference of the instruction, the tag check has to be performed. However, at and after the second reference, if no cache miss has occurred since the first reference, it is guaranteed that the target instruction currently resides in the cache. Therefore, for accesses to the same instruction in a stable-time, performing tag checks is absolutely required at the first reference, but not for the following references. We can omit tag checks if the following conditions are satisfied.

- The target instruction has been executed at least once.
- No cache miss has occurred since the previous execution of the target instruction.

Figure 2 shows how many unnecessary tag checks are performed in a conventional 16 KB direct-mapped cache for two SPEC benchmark programs. Simulation environment is explained in Section 4.1. The y-axis is the average reference-count (up to one hundred times) for each cache line per stable-time. We ignored where each cache line has never referenced in a stable-time. The x-axis is the cache-line address. It can be understood from the figure that the conventional cache wastes a lot of energy for unnecessary tag checks. Almost all cache lines are referenced more than four times in a stable-time, and some cache lines are referenced more than one hundred times.

In order to detect the conditions for omitting unnecessary tag checks, the HBTC cache records execution footprints in an extended BTB (Branch Target Buffer). An execution footprint indicates whether the target-instruction block or fall-though-instruction block associated with a branch resides in the cache. An execution footprint is recorded after all instructions in the corresponding instruction block are referenced. All execution footprints are erased, or invalidated, whenever a cache miss takes place. At the execution of an instruction block, if the corresponding execution footprint is detected, we can fetch instructions without performing tag checks.

3.2 Organization

Figure 3 depicts the organization of the extended BTB. The following two 1-bit flags are added to each BTB entry.

- EFT (Execution Footprint of Target instructions): This is an execution footprint of the branch-target-instruction block whose beginning address is indicated by the target address of current branch.
- EFF (Execution Footprint of Fall-through instructions): This is an execution footprint of the fall-through-instruction block whose beginning address is indicated by the fall-through address of current branch.

The end address of the branch-target- and fall-though-instruction block is indicated by another branch-instruction address which is already registered in the BTB, as shown in Figure 3.

In addition, the following hardware components are required.

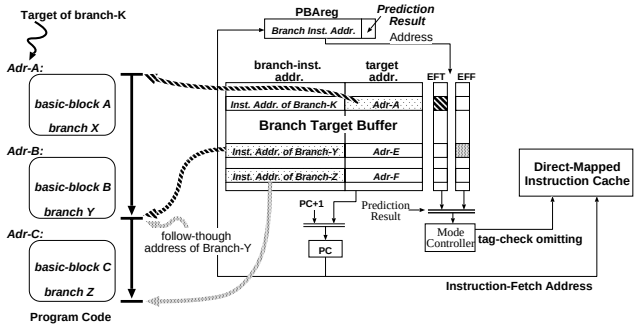

Figure 3: The Organization of a Direct-Mapped HBL Cache

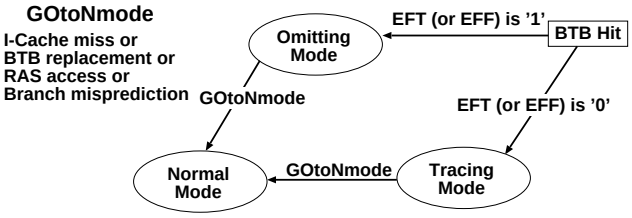

Figure 4: Operation-Mode Transition

- Mode Controller: This component selects one of the following operation modes based on the execution footprints read from the extended BTB. The detail of operation is explained in Section 3.3.
  - Normal-Mode (Nmode): The HBL cache behaves as a conventional cache (tag checks are performed).
  - Omitting-Mode (Omode): Tag checks for instruction-cache accesses are omitted.
  - Tracing-Mode (Tmode): The HBL cache behaves as a conventional cache (tag checks are performed). When a BTB hit is detected in this mode, the execution footprint indexed by the PBAreg is set to '1'.
- PBAreg (Previous Branch-instruction Address REGister): This is a register to keep the previous-branch-instruction address. The prediction result (taken or not-taken) is also kept.

3.3 Operation

Execution footprints (i.e., EFT and EFF flags) are left or erased at run time. Figure 4 shows operation-mode transition. On every BTB hit, the HBTC cache works as follows:

1. Regardless of current operation mode, both EFT and EFF flags associated with the BTB-hit entry are read in parallel.
2. Based on the branch-prediction result, EFT for taken or EFF for not-taken is selected.
3. If the selected execution footprint is '1', operation mode is transited to Omode.
4. Otherwise, operation mode is transited to Tmode. At that time, current PC (branch-instruction address) and the branch-prediction result are stored into the PBAreg.

Whenever a cache miss takes place, operation mode is transited to Nmode, as explained in the next paragraph. Therefore, occurring a BTB hit on Tmode means that there has never been any cache miss since the previous BTB hit. In other words, the instruction block, whose beginning address is indicated by the PBAreg and end address is indicated by the current branch-instruction address, has been referenced without causing any cache miss. Thus, when a BTB hit occurs on Tmode, the execution footprint indexed by the PBAreg is validated (set to 1).

If one of the followings takes place, execution footprints have to be invalidated. In addition, the operation mode is transited to Nmode.

- instruction-cache miss: The state of instruction cache is changed by filling the missed instruction. The cache-line replacement might evict the instruction block (or a part of the instruction block) corresponding to valid execution footprints from the cache. Therefore, the execution footprints of the victim line have to be invalidated.
- BTB replacement: As explained in Section 3.3, the end address of an instruction block is indicated by another branch-instruction address already registered in the BTB. We lose the end-address information when the BTB-entry is evicted. Thus, the execution footprints of the instruction block, whose end address is indicated by the victim BTB-entry, have to be invalidated.

Although it is possible to invalidate only the execution footprints affected by the cache miss or the BTB replacement, we have employed a conservative scheme, i.e., all execution footprints in the extended BTB are invalidated. In addition, when an indirect jump is executed, or a branch mis-prediction is detected, the

HBTC cache works on Nmode (tag checks are performed as conventional organization). These decisions make it possible to avoid area overhead and complex control logic.

3.4 Advantages and Disadvantages

Total energy dissipated in the HBTC cache (E_{TOTAL}) can be expressed as follow:

$$E_{TOTAL} = E_{CACHE} + E_{BTBadd},$$

where E_{CACHE} is the energy consumed in the instruction cache and E_{BTBadd} is the additional energy for BTB extension. The energy consumed in conventional BTB organization is not included. E_{CACHE} can be approximated by the following equation:

$$E_{CACHE} = E_{tag} + E_{data} + E_{output} + E_{ainput},$$

where E_{tag} and E_{data} are the energy consumed in tag memory and data memory, respectively. E_{output} is the energy for driving output buses, and E_{ainput} is that for address decoding. In this paper, we do not consider E_{ainput}, because some papers reported that it is about three orders of magnitude smaller than other components [1] [8]. E_{BTBadd} can be expressed as follows:

$$E_{BTBadd} = E_{BTBef} + E_{BTBlogic},$$

where E_{BTBef} is the energy consumed for reading and writing execution footprints, and $E_{BTBlogic}$ is that for the control logic (i.e., mode controller and PBAreg). The logic portion can be implemented by simple and small hardware, so that we do not take account for $E_{BTBlogic}$.

On Omitting-Mode (Omode), the energy consumed for tag checks (E_{tag}) is completely eliminated. However, that for accessing execution footprints (E_{BTBef}) appears as energy overhead on every BTB access. On the other hand, from performance point of view, the HBTC cache causes performance degradation. Reading execution footprints can be performed in parallel with normal BTB access from the microprocessor. However, for writing, the HBTC cache causes one processor-stall cycle. This is because the BTB entry accessed for execution-footprint writing and that for branch-target reading are different. Whenever a cache miss or BTB replacement takes place, execution-footprint invalidation is required. This operation also causes processor-stall cycles, because BTB access from the microprocessor has to wait until the invalidation is completed. The invalidation penalty largely depends on the implementation of the BTB. In Section 4.2.4, we discuss the effects of the invalidation penalty on processor performance.

4 Evaluation

4.1 Simulation Environment

In order to evaluate the performance-energy efficiency of the HBTC cache, we have measured the total energy consumption (E_{TOTAL}) explained in Section 3.4 and total clock cycles as performance. We modified the SimpleScalar source code for this simulation [15]. To calculate energy consumption, the cache energy model assuming 0.8 um CMOS technology explained in [6] was used. We referred the load capacitance for each node from [7] [12].

In this simulation, the following configuration was assumed: instruction-cache size is 16 KB, cache-line size is 32 B, the number of direct-mapped branch-prediction-table entry is 2048, predictor type is bimod, the number of BTB set is 512, BTB associativity is 4, and RAS size is 8. For other parameters, the default value of the SimpleScalar out-of-order simulator was used. In addition, we assumed that all caches evaluated in this paper employ subbanking approach, 16 KB data memory is partitioned into 4 subbanks. The following benchmark programs were used in this evaluation.

- SPECint95 [16]: *099.go*, *124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, *132.ijpeg* (using training input).
- Mediabench [14]: *adpcm_encode*, *adpcm_decode*, *mpeg2_encode*, *mpeg2_decode*

4.2 Results

4.2.1 Tag-Check Count

Figure 5 shows tag-check counts required for whole program executions. All results are normalized to a 16 KB conventional cache. The figure includes the simulation results for the ITC cache explained in Section 2 and the combination of the ITC cache and the HBTC cache.

Since sequential accesses are inherent in programs, the ITC cache works well for all benchmark programs. While the effectiveness of the HBTC cache is application dependent. The HBTC cache produces more tag-check count reduction than the ITC cache for two SPEC integer programs, *129.compress* and *132.ijpeg*, and all media programs. In the best case, *adpcm_dec*, the tag-check count is reduced by about 90 %. However, for the other benchmark programs, the ITC cache is superior to our approach. This result can be understood by considering the characteristics of benchmark programs. Media application programs have relatively well structured loops. The HBTC cache attempts to

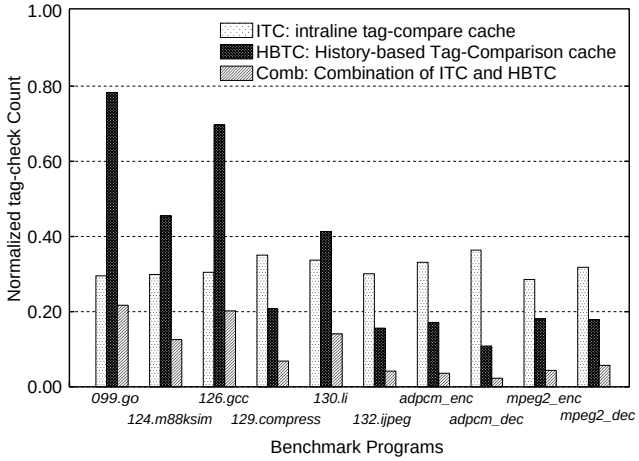

Figure 5: Tag-check count compared with other approaches

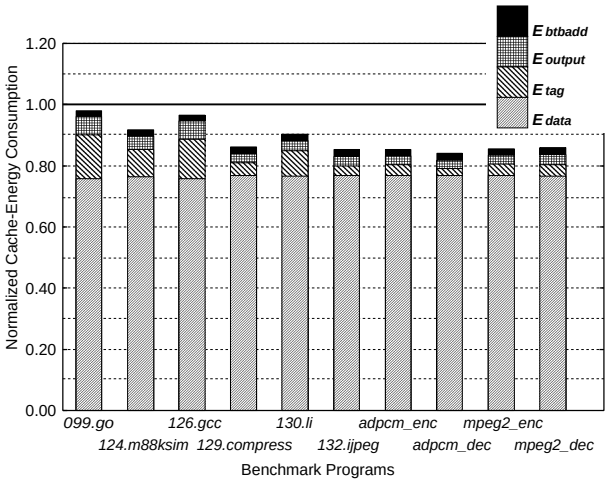

Figure 6: Cache-Energy Consumption

avoid performing unnecessary tag checks by exploiting iterative execution behavior. Thus, we can consider that if our main target is media applications, employing the HBTC cache makes energy advantages. Otherwise, we should employ the ITC cache.

The hybrid model of the ITC cache and the HBTC cache makes significant reductions. It eliminates more than 80 % and 95 % of unnecessary tag checks for all benchmark programs. Therefore, we conclude that combining the ITC and the HBTC caches is the best approach to avoiding energy dissipation caused by unnecessary tag checks.

4.2.2 Energy Consumption

Figure 6 reports energy consumption of the HBTC cache and its break down for each benchmark program. All results are normalized to the conventional cache. As explained in Section 4.1, a 0.8 um CMOS technology is assumed. The energy model used in this paper does not take account for the energy consumed in sense amplifiers. However, we believe that the energy reduction reported in this section can be achieved even if sense amplifiers are considered. This is because tag-memory accesses can be completely eliminated when the HBTC cache works on Omitting-Mode, so that the energy consumed in sense amplifiers is also eliminated.

As discussed in Section 4.2.1, the HBTC cache makes a significant tag-check count reduction for *129.compress*, *132.ijpeg*, and all media application programs. Since the extension of each BTB entry for execution footprints is only 2 bits, the energy overhead for BTB accesses (E_{BTBadd}) does not have a large impact on the total cache energy. As a result, the HBTC cache reduces the total cache energy by about 15 %. However, for *099.go* and *126.gcc*, the energy reduction is only from 2 % to 3 %. This is because the HBTC cache could not eliminate effectively unnecessary tag checks due to irregular behavior of the program execution.

4.2.3 Performance Overhead

As explained in Section 3.4, the HBTC cache causes processor stalls when the extended BTB is up-dated for recording or invalidating execution footprints. Figure 7 shows program-execution time in terms of the total number of clock cycles. All results are normalized to the conventional organization.

From the simulation results, it is observed that the performance degradation is less than 1 % for all but three benchmark programs. However, for *126.gcc*, the performance is degraded by about 2.5 %. This might not be acceptable if high performance is strictly required. The processor-stalls are caused by conflicting BTB accesses from the processor with up-date operations of execution footprints.

In order to alleviate the negative effect of the HBTC cache, we can consider two approaches. First is to pre-decode fetched instructions. Since conventional BTB is accessed on every instruction fetch regardless of the instruction type, processor stalls occur whenever execution footprints are up-dated. By pre-decoding fetched instructions, we can determine whether, or not, it has to access the BTB before starting normal BTB access. In this case, processor stalls occur only when branch (or jump) instruction conflicts with up-dating execu-

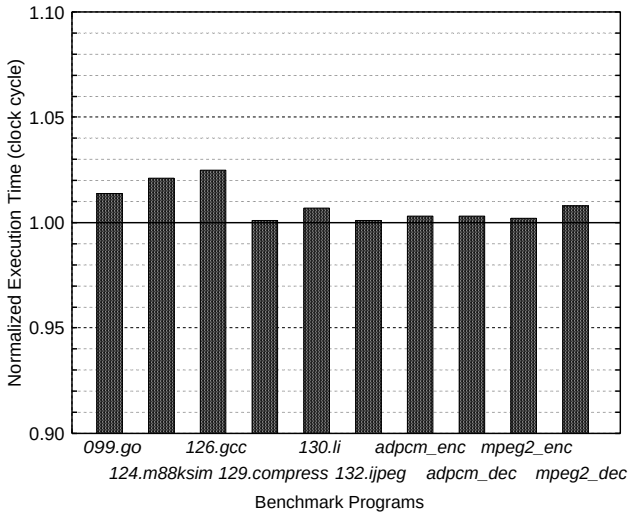

Figure 7: Program Execution Time

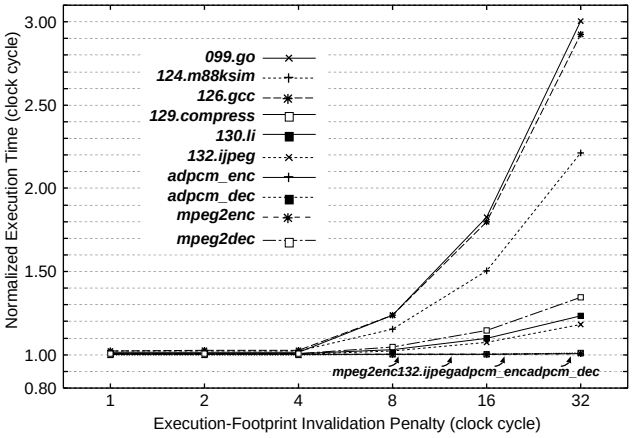

Figure 8: Effect of Execution-Footprint Invalidation Penalty

tion footprints. Another approach for compensating the processor stalls is to add a decoder logic for accessing execution footprints. This makes it possible to access BTB for obtaining branch-target address and up-dating execution footprints simultaneously.

4.2.4 Effects of Execution-Footprint-Invalidation Penalty

All execution footprints recorded in the extended BTB are invalidated whenever a cache miss, or a BTB replacement, takes place. So far, we have assumed that the invalidation can be completed in one processor-clock cycle. However, the invalidation penalty largely depends on the implementation of extended BTB.

Figure 8 depicts performance overhead caused by the HBTC approach where the invalidation penalty is varied from 1 to 32 cycles. The y-axis indicates program-execution time normalized to conventional organization for all benchmark programs, and the x-axis shows the invalidation penalty in terms of clock cycles. For all benchmark programs, it is observed that performance degradation is trivial if the invalidation penalty is equal to or less than 4 clock cycles. We have analyzed the break down of the invalidations, and have found that more than 98 % are caused by cache misses (less than 2 % are caused by BTB replacements). The invalidation penalty can be hidden if it is smaller than cache-miss penalty. Actually, in this evaluation, we have assumed that cache-miss penalty is 6 clock cycles.

However, the invalidation penalty clearly appears where it is grater than 6 clock cycles, so that we can see large performance degradation for *099.go* and *126.gcc*. On the other hand, for *132.ijpeg*, *adpcm_enc*, *adpcm_dec*, and *mpeg2_decode*, performance degradation is small even if the invalidation penalty is large. This is because cache-miss rates for these programs are high, resulting in the small number of invalidations. Actually, each cache-miss rate of *099.go*, *126.gcc*, *132.ijpeg*, and *mpeg2_decode* was 4.7%, 5.5%, 0.5%, and 0.5%, respectively.

5 Conclusions

In this paper, we have proposed the history-based tag-comparison (HBTC) cache for low-energy consumption. The HBTC cache exploits the following two facts. First, instruction-cache-hit rates are much higher. Second, almost all programs have many loops. The HBTC cache records the execution footprints, and determines whether the instructions to be fetched are currently cache resident without performing tag checks. An extended branch target buffer (BTB) is used to record the execution footprints. In our simulation, it has been observed that the HBTC cache can reduce the total count of tag checks by about 90 %, resulting in 15 % of cache-energy reduction.

In our evaluation, it has been assumed that the BTB size, or the total number of BTB entries, is fixed. Our future work is to evaluate the effects of the BTB size on the energy reduction achieved by the HBTC cache. In addition, the effects of branch-predictor type will be evaluated. Another future work is to establish a microarchitecture for set-associative caches. By memorizing way-access information as proposed in [9], we can

extend the HBTC approach for set-associative caches.

References

[1] R. I. Bahar, G. Albera, and S. Manne, "Power and Performance Tradeoffs using Various Caching Strategies," *Proc. of the 1998 International Symposium on Low Power Electronics and Design*, pp.64–69, Aug. 1998.

[2] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor," *Proc. of the 1999 International Symposium on Low Power Electronics and Design*, pp.64–69, Aug. 1999.

[3] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis,"Energy and Performance Improvements in Microprocessor Design using a Loop Cache," *Proc. of the 1999 International Conference on Computer Design: VLSI in Computers & Processors*, pp.378–383, Oct. 1999.

[4] K. Inoue and K. Murakami,"A Low-Power Instruction Cache Architecture Exploiting Program Execution Footprints," *International Symposium on High-Performance Computer Architecture, Work-in-progress session (included in the CD proceedings)*, Feb. 2001.

[5] T. Ishihara and H. Yasuura,"A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors," *Proc. of the Design, Automation and Test in Europe Conference*, pp617–623, Mar. 2000.

[6] M. B. Kamble and K. Ghose,"Analytical Energy Dissipation Models For Low Power Caches," *Proc. of the 1997 International Symposium on Low Power Electronics and Design*, pp.143–148, Aug. 1997.

[7] M. B. Kamble and K. Ghose,"Energy-Efficiency of VLSI Caches: A Comparative Study," *Proc. of the 10th International Conference on VLSI Design*, pp.261–267, Jan. 1997.

[8] J. Kin, M. Gupta, and W. H. Mngione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Proc. of the 30th Annual International Symposium on Microarchitecture*, pp.184–193, Dec. 1997.

[9] A. Ma, M. Zhang, and K. Asanović,"Way Memorization to Reduce Fetch Energy in Instruction Caches," *ISCA Workshop on Complexity Effective Design*, July 2001.

[10] R. P. Panda, D. N. Dutt, and A. Nicolau,"Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *Proc. of European Design & Test Conference*, Mar. 1997.

[11] R. Panwar and D. Rennels,"Reducing the frequency of tag compares for low power I-cache design," *Proc. of the 1995 International Symposium on Low Power Electronics and Design*, Aug. 1995.

[12] S. J. E. Wilton and N. P. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," *WRL Research Report 93/5*, July 1994.

[13] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović,"Direct Addressed Caches for Reduced Power Consumption," *Proc. of the 34th International Symposium on Microarchitecture*, Dec. 2001.

[14] MediaBench, *URL: http://www.cs.ucla.edu/l̃eec/mediabench/.*

[15] "SimpleScalar Simulation Tools for Microprocessor and System Evaluation," *URL:http://www.simplescalar.org/.*

[16] SPEC (Standard Performance Evaluation Corporation), *URL: http://www.specbench.org/osg/cpu95.*