A Low Energy Set-Associative I-Cashe with Extended BTB

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 Conference on Computer Design, pp.187-192, 2002-09. IEEE Computer Society
バージョン：
権利関係：

A Low Energy Set-Associative I-Cache with Extended BTB

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

Kazuaki 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 low-energy instruction-cache architecture, called history-based tag-comparison (HBTC) cache. The HBTC cache attempts to re-use tag-comparison results for avoiding unnecessary way activation in set-associative caches. The cache records tag-comparison results in an extended BTB, and re-uses them for directly selecting only the hit-way which includes the target instruction. In our simulation, it is observed that the HBTC cache can achieve 62% of energy reduction, with less than 1% performance degradation, compared with a conventional cache.

1. Introduction

On-chip caches have been playing an important role in bridging the performance gap between low-speed main memory and high-speed microprocessors. As on-chip cache size has increased, however, the energy dissipated by on-chip caches has become significant. Instruction caches (or I-caches) particularly affects total energy consumption due to their high access frequency. For instance, ARM920T microprocessor dissipates 25% of its total power in the I-cache [10]. For recent mobile devices, such as notebook and hand-held computers, one uncompromising requirement is low-energy consumption, because that directly affects the battery life. Therefore, it is important to reduce the energy consumption of the I-cache.

Modern microprocessors employ *set-associative scheme* as L1 or L2 caches to achieve high cache-hit rates. An *n*-way set-associative (SA) cache has *n* locations where a cache line (or a block) can be placed. However, from the energy point of view, SA caches tend to dissipate larger amount of energy than direct-mapped caches [1]. The energy consumed per I-cache access, E_{Cache}, can roughly be approximated by the following equation [3]:

$$E_{Cache} = Tnum \times E_{tag} + Lnum \times E_{line}, \quad (1)$$

where E_{tag} and E_{line} are the energy dissipated for reading a tag and a cache-line from SRAM array, respectively; $Tnum$ and $Lnum$ are the number of tags and cache-lines accessed, respectively.

On an SA-cache access, all ways are activated in parallel to compensate for longer access time. Namely, both $Tnum$ and $Lnum$ are equal to the cache associativity n. However, the parallel search strategy dissipates large amount of energy unnecessarily, because at most only one way can include the target instruction. In order to solve the energy issue of the SA cache, we propose an I-cache architecture called *history-based tag-comparison cache (HBTC cache)*. The HBTC cache attempts to co-operate with an extended branch target buffer (BTB) to avoid unnecessary way activation. Tag-comparison results stored in the BTB are re-used in order to directly select the hit-way (i.e., $Tnum = 0$and $Lnum = 1$)[1]. Since our approach does not affect the cache-access time and the hit rate, the cache performance can be maintained.

The rest of this paper is organized as follows: Section 2 shows related work. Section 3 presents the organization of the HBTC cache, and explains its operation in detail. In Section 4, we evaluate the energy efficiency of the HBTC cache, and Section 5 gives some concluding remarks.

2. Related Work

In order to alleviate the negative effect of SA caches, researchers have proposed many low-energy cache architec-

[1] In [4], a cache architecture exploiting an extended BTB has been proposed to reduce tag-check frequency of direct-mapped instruction caches. On the other hand, the architecture presented in this paper can be applied to set-associative caches, and eliminates not only the energy for tag checks but also that for data read.

tures. A well known approach is to employ a small Level-0 cache (or L0-cache) between the microprocessor and the Level-1 main cache, e.g., Block-Buffer[5], S-cache[9], filter-cache[7], loop-cache[2], etc. By concentrating memory accesses to the L0-cache, the number of L1-cache accesses can be reduced (i.e., $Tnum = 0$ and $Lnum = 0$). Only when an L0-cache miss takes place, the L1-cache is accessed. The HBTC cache proposed in this paper does not affect the memory hierarchy, therefore it can be used in conjunction with the L0-caches.

If two instructions *i* and *j* reside in the same cache line, and instruction *j* is executed immediately after *i*, then it is guaranteed that the hit-way of *j* is the same as that of *i*. Therefore, without performing tag checks, we can directly select the hit way for *j* (i.e., $Tnum = 0$ and $Lnum = 1$)[9][10]. We call this technique *interline tag-comparison cache (ITC cache)* in this paper. On the other hand, the HBTC cache can eliminate unnecessary way activation even if consecutive instructions reside in different cache lines.

In a way-predicting cache[3], before starting normal cache access, the hit-way is predicted. If the way prediction is correct, only the hit-way is activated and the cache access can be completed in one cycle (i.e., $Tnum = 1$ and $Lnum = 1$). Witchel et al.[12] presented a *Direct Addressed* scheme that allows software to access cache data without hardware tag checks. The key idea is to store the tag-check results to the *DA-register file*, and then re-use them based on the compiler supports (i.e., $Tnum = 0$ and $Lnum = 1$). Unlike the way-predicting cache, our scheme does not perform speculative operation. In addition, no software supports are required, therefore the code compatibility can be completely maintained.

Ma et al.[8] suggested a dynamic way-memoization for eliminating the cache-search operation. The idea is to record within I-cache both the tag check results (*links*) and the *valid bits*. If the link is valid, it is followed to fetch the next instruction without tag checks (i.e.,$Tnum = 0$ and $Lnum = 1$). Although our purpose is similar to that of the way memoization, implementation scheme is completely different. In our approach, the hardware for tag-comparison re-use is separated from the cache core, so that cache-core access time, energy consumption, and cost are not affected. In addition, since we exploit the BTB already employed for branch prediction, our scheme can be implemented with smaller hardware overhead.

3. History-Based Tag-Comparison

The HBTC cache exploits a BTB, which is conventionally used for branch prediction, to achieve I-cache energy reduction. In this section, we explain the details of the HBTC cache architecture.

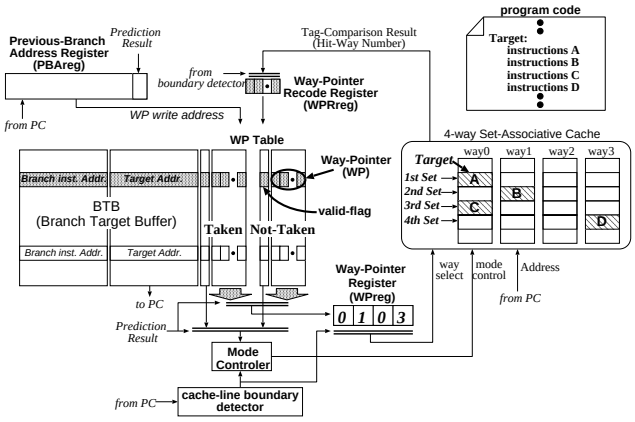


Figure 1. Block diagram of a 4-way SA HBTC cache

3.1. Tag-Comparison Re-use

For almost all programs, I-caches can achieve high hit rates. In other words, the state (or contents) of the I-cache is rarely changed. Only when a cache miss takes place, the state is changed by evicting some instructions from the cache and filling the missed instructions. Therefore, once an instruction is loaded into the cache, it stays there at least until the next cache miss occurs.

Now, consider where an instruction is executed repeatedly. At the first reference of the instruction, tag checks have to be performed to identify the hit way (i.e., $Tnum = n$ and $Lnum = n$: n is the cache associativity). However, at and after the second reference, if no cache miss has occurred, it is guaranteed that the target instruction still stays at the same location. Therefore, by re-using the tag-comparison result of the first reference, we can directly select the hit-way (i.e., $Tnum = 0$ and $Lnum = 1$).

High performance microprocessors employ a branch target buffer (BTB) to obtain a branch-target address in an early pipeline stage. Executed taken-branches are registered in the BTB at run time. Therefore, the BTB inherently includes the information for execution footprints. The main idea of the HBTC cache is to exploit the BTB information for the tag-comparison re-use.

3.2. Organization

As shown in Figure 1, the HBTC cache requires six additional components: Way-Pointer table (WP table), Way-Pointer Register (WPreg), Way-Pointer Record Register (WPRreg), a mode controller, Previous Branch-Address Register (PBAreg), and Cache-Line Boundary Detector.

A conventional BTB is extended by adding the WP table. Each entry of the table corresponds to that of the BTB, and consists of two of M way-pointers. A tag-comparison result (i.e., hit-way number) is stored in the extended BTB as a way pointer (WP). Therefore, the WP can be implemented as a $\log n$-bit flag where n is the cache associativity, and specifies the hit-way of the corresponding instructions. The 1-bit valid-flag is used for determining whether the M of WPs are valid, or not. The taken WPs are used for the target instructions, and the not-taken WPs are used for the fall-through instructions, of the corresponding branch in the BTB. In Figure 1, for example, cache line **A**, **B**, **C**, and **D** are referenced sequentially after a taken branch is executed. In this case, the tag-comparison results (or the hit-way numbers) for their references are **0**, **1**, **0**, and **3**. This information is stored in the WP table, and is re-used when the target instructions are referenced in the future. Note that the HBTC cache attempts to re-use tag-comparison results at cache-line granularity.

At the first reference of instructions, we have to perform tag checks. In order to record the generated tag-comparison results in the WP table, the WPRreg is used as a temporal register. 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 to store the value of the WPRreg to the WP table. At every BTB hit, the WPs read from the BTB is stored in the WPreg, and are provided to the I-cache for tag-comparison re-use. The mode controller manages the HBTC behavior. The details of the HBTC operation are explained in Section 3.3.

In order to re-use the tag-comparison results at cache-line granularity, we need to detect cache-line boundary for instruction references. This can be done by monitoring PC control signals and a few bits of the PC[9].

3.3. 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 (i.e., $Tnum = n$ and $Lnum = n$ where n is the cache associativity).
- Omitting mode (Omode): The cache re-uses tag-comparison results, so that only the hit-way is activated without performing tag checks (i.e., $Tnum = 0$ and $Lnum = 1$).
- Tracing mode (Tmode): The cache works as the same as the Nmode (i.e., $Tnum = n$ and $Lnum = n$), and also attempts to record the tag-comparison results generated by the I-cache (this operation is not performed in the Nmode).

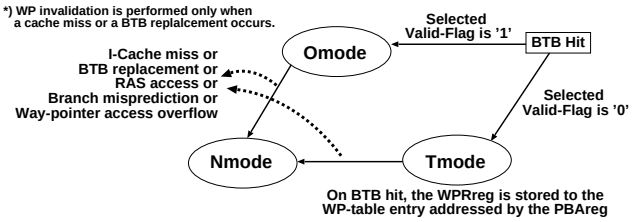


Figure 2. Operation-Mode Transition

Figure 2 displays the operation transitions. On every BTB hit, the HBTC cache reads in parallel both the taken and the not-taken WPs associated with the BTB-hit entry, and selects one of them based on the branch prediction result. If the selected valid-flag is '1 ', the operation enters the Omode and the selected WPs are stored to the WPreg. Otherwise the Tmode is activated, and both the branch-instruction address (PC) and the branch-prediction result (taken or not-taken) are stored to the PBAreg.

In the Omode, whenever a cache-line boundary is detected, the next WP in the WPreg is selected. On the other hand, in the Tmode, the tag-comparison results generated by the I-cache are stored to the WPRreg at cache-line granularity. When the next BTB hit occurs in the Tmode, the value of the WPRreg is written into the WP-table entry pointed by the PBAreg and the corresponding valid-flag is set to '1'.

The WPreg and the WPRreg can hold WPs up to M, where M is the total number of WPs implemented in a WP-table taken (or not-taken) entry. In the Omode or the Tmode, if the cache attempts to access the $M + 1$th WP in the WPreg or the WPRreg, *WP-access overflow* occurs and the operation switches to the Nmode.

Whenever a cache miss takes place, all WPs recorded in the WP table are invalidated by resetting all the valid-flags to '0', and operation transits to the Nmode. This is because instructions corresponding to valid WPs may be evicted from the cache.

In the Tmode, when a BTB hit occurs before the number of recorded WPs reaches to M (i.e., before the WPRreg is completely filled by valid WPs), some of invalid WPs are stored to the WP table. This issue can be solved by following BTB access behavior. Consider the BTB-entry *i* makes a BTB hit in the Tmode, which occurs just after $L(L < M)$ of tag-comparison results are written in the WPRreg. In this scenario, L of valid WPs and $M - L$ of invalid WPs are stored to the WP table, and the corresponding valid-flag is set to '1'. After that, when the stored WPs are re-used, the HBTC cache switches to the Omode because the corresponding valid-flag is '1'. However, we can re-use only L of valid WPs, because the remaining $M - L$ WPs are invalid. Here, we assume that no BTB replacement has oc-

curred since the previous Tmode. Under this assumption, it is guaranteed that the BTB-entry *i* makes the next BTB hit just after L of valid WPs are accessed. Since the WPreg is overwritten by the next BTB hit, there is no chance to be used for the $M - L$ of invalid WPs. In order to guarantee this assumption, the cache performs WP invalidation and changes the operation mode to the Nmode whenever not only a cache miss takes place but also a BTB replacement occurs.

To simplify the mode control, in addition, we assume that the cache operates in the Nmode whenever a branch-target address is provided by a return address stack (RAS), or a branch mis-prediction is detected (WP invalidation is not performed).

3.4. Performance/Energy Overhead

The proposed scheme eliminates unnecessary way activation in the Omode. Thus, the energy reduction for cache accesses is proportional to the number of accesses in the Omode. On the other hand, accessing the WP table produces an energy overhead at every BTB access.

From the performance point of view, controlling the WPs may lead to some performance degradation. Writing the WPRreg into the WP table may be performed to a different entry with that accessed for branch-target prediction. Therefore, the branch-prediction unit has to wait until the WP write operation is completed. Also, during WP invalidation, accessing to the BTB is prohibited. These BTB conflicts cause one processor-stall cycle. Since reading the WPs can be performed in parallel with the BTB access for obtaining a branch-target address, no performance overhead appears for WP-table read accesses.

4. Evaluation

In this section, we evaluate the performance/energy efficiency of the HBTC cache by using some benchmark programs.

4.1. Simulation Environment

We used the SimpleScalar simulator from [14] to evaluate the HBTC cache. The cache energy was computed based on 0.8μm CMOS using the cache-energy model presented in [5]. The load capacitance of each node was taken from [6][11]. The energy overhead caused by the WP table is also included. However, we did not take into account the energy consumed by logic portion, the WPreg, the WPRreg, the PBAreg, the mode controller, and the cache-line boundary detector. In addition, we do not consider the energy consumed by address decoder, because it is usually three orders of magnitude smaller than the other cache components[1][7].

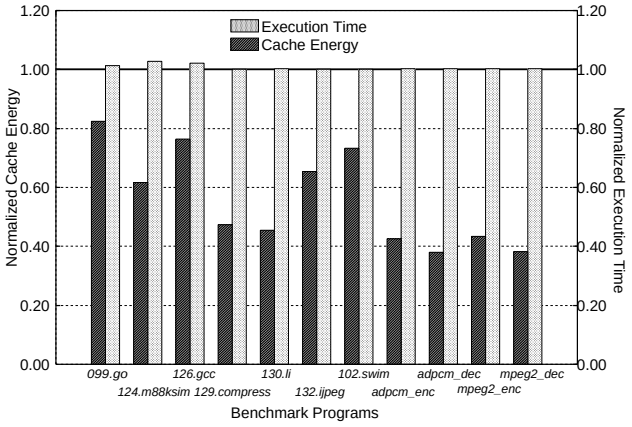


Figure 3. Energy consumption and program-execution time

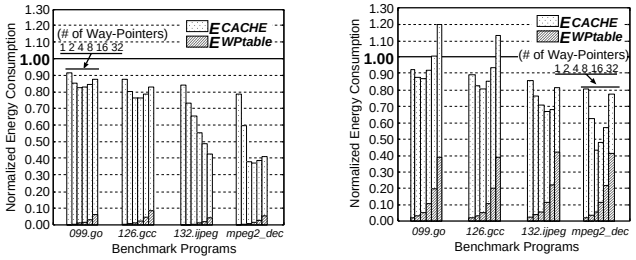
We experimented with a 16 KB 4-way SA I-cache, 32 B cache-line size. The number of WPs implemented in each WP table entry (M) is 4. In addition, it is assumed that the BTB access occurs only when jump or branch instructions are executed by employing pre-decoding scheme. The energy efficiency without pre-decoding scheme is also evaluated in Section 4.3. Moreover, the following configuration is assumed unless stated otherwise: 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.

The cache was simulated on six SPEC95 integer programs (*099.go*, *124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, and *132.ijpeg*) using the train input, one SPEC95 floating-point program (*102.swim*) using the test input [15], and two Mediabench programs [13] (*adpcm* and *mpeg2*). For each of the Mediabench programs, an encoder and a decoder were evaluated separately.

4.2. Performance/Energy Efficiency

Figure 3 shows energy consumption of the HBTC cache and program-execution time in terms of clock cycle. All results are normalized to the value of a conventional 16 KB 4-way SA cache. Note that the energy consumed by the WP table is included. As the figure shows, the HBTC cache reduces the total cache-energy by up to 62% (decoders of media programs) from the conventional organization.

Now consider the bars, which depict the normalized execution time in Figure 3. Wee can see that the perfor-



(A) with pre-decoding (B) without pre-decoding

Figure 4. Effect of the number of way-pointers

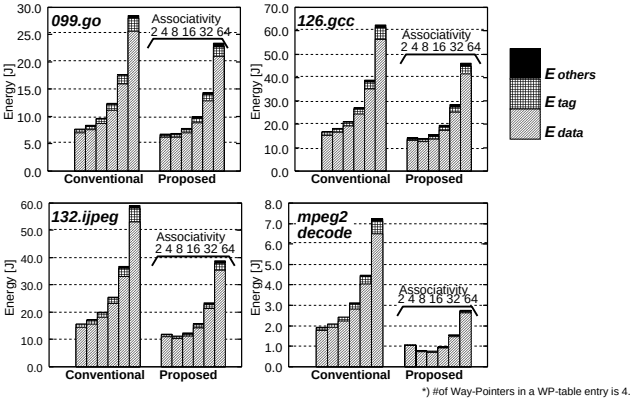


Figure 5. Effect of cache associativity

mance degradation is less than 1% for all but three (*099.go*, *124. m88ksim*, and *126.gcc*) benchmarks. The performance overhead is caused by the processor stalls due to simultaneous BTB accesses from the processor and the HBTC cache. However, this negative effect can be easily alleviated by adding a special write-port for up-dating the WP table.

4.3. Number of Way-Pointers

So far, we have assumed that the number of WPs implemented in each WP-table entry (M) is 4. Although increasing M would improve the opportunity for re-using tag-comparison results, it also increases the energy overhead caused by WP-table accesses.

Figure 4 depicts the total energy and its breakdown for four benchmark programs; *099.go* and *126.gcc* (worst two benchmarks in Figure 3), *mpeg2_dec* (the best benchmark), and *132.ijpeg* (middle of them). E_{CACHE} and $E_{WPtable}$ are the energy consumed by the I-cache and the WP table, respectively. In Figure 4 (A), pre-decoding is assumed (the WP table is accessed only when branch or jump instructions are executed), whereas the Figure 4 (B) is the results without pre-decoding scheme (the WP table is accessed at every instruction fetch).

From Figure 4 (A), it is observed that the total energy is reduced with the increase in the number of WPs until it reaches to 4. After that, the energy consumption increases in proportional to the number of WPs. This phenomenon is clearly shown in Figure 4 (B) due to the energy overhead caused by the WP table. From the simulation results, we conclude that the appropriate number of WPs is 4. Moreover, we can see that the HBTC cache produces significant energy reduction even if the pre-decoding scheme is not employed (e.g., the cache reduces the total energy by about 55% from the conventional cache for *mpeg2_dec*).

4.4. Cache Associativity

Figure 5 shows the total energy and its breakdown of the HBTC cache in case that the cache associativity is changed from 1 to 64. E_{data} and E_{tag} are the energy consumed by data-memory accesses and tag-memory accesses, respectively. E_{others} includes the energy consumed for driving output buses (E_{output}) and that for accessing the WP table.

In a conventional cache, the energy consumption is proportional to the cache associativity, because the energy dissipated by memory-array accesses ($E_{data} + E_{tag}$) is increased [1]. On the other hand, in the HBTC cache, we can see a different phenomenon. Since increasing the cache associativity reduces the total capacity of each way, this kind of selective activation effectively reduces cache-access energy. Therefore, if many cache accesses are performed in the Omode, as *132.ijpeg* and *mpeg2dec*, the total cache energy can be reduced as the associativity is increased. However, the energy starts to increase when the associativity is higher than 4 or 8. This is because the energy overhead caused by increasing the cache associativity is over the amount of energy reduced by the selective activation.

4.5. HBTC Cache vs ITC Cache

As explained in Section 2, one of common techniques to avoid the unnecessary way activation is the interline tag-comparison (ITC) scheme. Figure 6 shows the comparison results of the HBTC cache with the ITC cache in terms of the total number of cache look-up performed (i.e., total number of parallel search executed). All results are normalized to the look-up count in the conventional cache.

Since programs inherently include sequential instruction-accesses, the ITC cache works well. For all but one, *adpcm_dec*, the cache makes from 60% to 70% of reduction. While the effectiveness of the HBTC

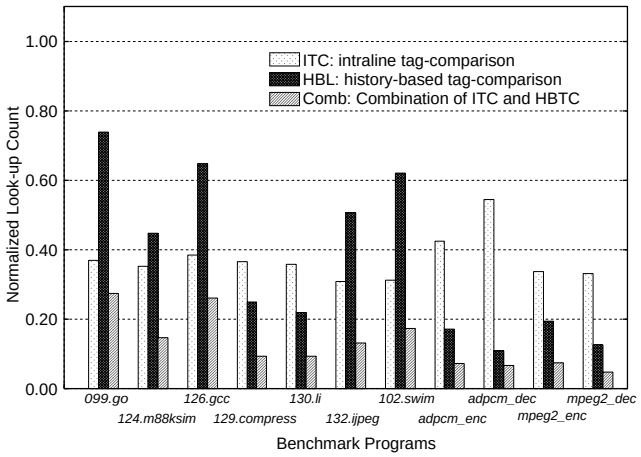


Figure 6. Comparison with other approaches

cache is application dependent. The HBTC cache produces better results than the ITC cache for more than half of programs. In particular, the cache works very well for media applications, *adpcm* and *mpeg2*, more than 80% of look-up count reduction is achieved. This is because the HBTC cache attempts to avoid unnecessary cache look-up by exploiting the iterative execution of instructions, which is the main feature of media programs. In addition, we see that combining the HBTC approach with the ITC scheme achieves outstanding reduction, about 95% in the best case (*mpeg2_dec*) and 70% in the worst case (*099.go*).

5. Conclusions

In this paper, we have proposed a low-energy SA I-cache architecture, called history-based tag-comparison cache (HBTC cache). The HBTC cache attempts to re-use tag-comparison results to eliminate unnecessary way activation. An extended BTB is used for the re-use strategy.

In our simulation, it has been observed that a 4-way set-associative HBTC I-cache can achieve 62% of energy reduction from a conventional cache with less than 1% performance degradation. In addition, it has been reported that the proposed cache can produce remarkable energy reduction by combining with the interline tag-comparison approach, which is commonly used in low-power microprocessors, 95% of cache look-up could be eliminated in the best case.

In this evaluation, we have assumed that no energy is consumed in the logic portion for controlling the HBTC cache. Our ongoing work is to design the HBTC cache and evaluate the energy overhead caused by the logic portion.

Acknowledgments

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.

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, 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.

[3] K. Inoue, T. Ishihara, and K. Murakami, "Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption," *Proc. of the 1999 International Symposium on Low Power Electronics and Design*, pp.273–275, Aug. 1999.

[4] K. Inoue, V. Moshnyaga, and K. Murakami, "A History-Based I-Cache for Low-Energy Multimedia Applications," *(under review) the 2002 International Symposium on Low Power Electronics and Design*, Aug. 2002.

[5] 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.

[6] 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.

[7] 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.

[8] A. Ma et al., "Way Memorization to Reduce Fetch Energy in Instruction Caches," *ISCA Workshop on Complexity Effective Design*, July 2001.

[9] 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.

[10] S. Segars, "Low Power Desin Techniques for Microprocessors," *ISSCC Tutorial*, Feb. 2001.

[11] 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.

[12] E. Witchel et al., "Direct Addressed Caches for Reduced Power Consumption," *Proc. of the 34th Int. Symp. on Microarchitecture*, Dec. 2001.

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

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

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