A BLOCK PRIORITY BASED INSTRUCTION CACHING SCHEME FOR MULTIMEDIA PROCESSORS

Jiyang Kang and Wonyong Sung

School of Electrical Engineering
Seoul National University
Shinlim-dong, Gwanak-gu, Seoul 151-742 KOREA
E-mail: {jiyang, wysung}@dsp.snu.ac.kr

ABSTRACT

In this paper, a new instruction caching scheme that utilizes the block priority information is proposed mainly targeted for embedded multimedia processors. The block priority information is obtained by profiling application programs. The goal of this caching scheme is to keep more important code blocks longer using the block priority information, which programmers provide by analyzing the profiling results of multimedia applications. In addition to a new caching scheme, the methods for determining the priority of each code block are also developed and their performances are evaluated using real multimedia applications. The experimental results show that the cache miss ratio can be reduced up to nearly a half of that of the normal LRU replacement scheme although the improvement depends on the cache size.

1. INTRODUCTION

Many DSP or multimedia processors have employed internal memories, such as on-chip RAM and ROM, instead of cache memories [1] [2] [3]. On-chip memory spaces are not only linear, contiguous and addressable, but most importantly ensure the access time, which is critical for real-time applications. However, as the size of multimedia applications grows, the limited spaces of on-chip memories become hard to manage, especially in the case of instruction memory due to the need of sophisticated address conversion.

In contrast, cache memories are very convenient in that case, because caches usually do not require user management and, moreover, are able to contain multiple hot-spot codes from different parts of more than one application. For that reason, some of recently developed multimedia processors are employing flexible internal memory structures which can also be configured as instruction cache memories [4]. Cache memories need a replacement strategy to determine which cache line should be discarded when a cache miss occurs. Note that conventional caching scheme, as far as the authors know, do not assign different weights to each code segment.

To compensate for the dynamic behavior of a cache, various hardware or software instruction prefetching techniques can be employed [5]. However, if the program execution path is different from the instruction prefetch path due to branches, jumps, and function calls, the prefetched block may not be used. This wasted prefetch can cause increase in memory traffic, cache pollution and unnecessary power consumption, which makes it unacceptable for low-power portable systems.

This work was supported by the Brain Korea 21 Project and the National Research Laboratory program.

In this paper, we propose a new caching scheme focused on the applications of multimedia processors to combine the advantages of both internal memory and cache. The proposed caching scheme assigns priorities to each code block, and tries to keep more important code blocks longer in the case of set associative caches. The priority information is specified by a programmer by analyzing the regular program behavior of multimedia applications.

This paper is organized as follows. In Section 2, the proposed cache architecture is presented. A few proposed block prioritization methods that can be categorized into static and dynamic are shown in Section 3. In Section 4, the experimental environment and results are presented. Finally, Section 5 concludes this paper.

2. THE PROPOSED CACHE ARCHITECTURE

We assume that the candidate cache organizations for an embedded multimedia processor would be either a *direct mapped* or a *set associative* cache considering the cost-sensitive nature of embedded multimedia applications. With an *n*-way set associative cache, there are *n* different sets of cache lines to choose from when a cache miss occurs. A mostly used replacement policy to determine which cache line should be discarded is the LRU (Least Recently Used) algorithm [6]. This algorithm keeps track of when each cache line has been accessed by ordering cache entries in a stack structure. When a cache line is needed to load new program memory locations, the algorithm selects the cache line that has not been read from for the longest time.

In contrast, our proposed caching scheme utilizes the priority information of each code block given by a programmer as well as the LRU stack. Actually, the priority information can override the LRU stack state, as is shown in Fig. 1. If there are any cache lines with the priority 0, which is the lower priority, the victim for replacement is chosen among only the priority-0 lines according to the LRU policy. However, in the case when all the candidate lines are of the same priority, the cache behaves in the same way with the normal LRU replacement policy. Note that the proposed block prioritizing scheme can also be applied to other replacement policies, e.g., random and FIFO, with minor modifications.

In order to specify the priority information for a given code, instead of directly specifying a priority bit in the instruction word, a two level scheme is employed, as is illustrated in Fig. 2. First, on top of the most significant bit of the instruction address space, a

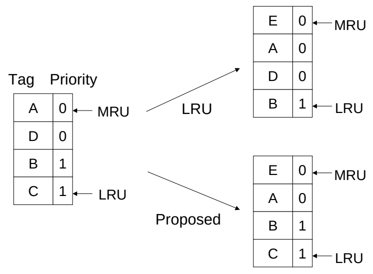


Fig. 1. Proposed cache replacement scheme

two-bit address header field is concatenated. This field represents the number of the code block where this instruction belongs to, and has no effect on the program control flow. Using a branch instruction referencing a full-length address rather than an offset value, e.g., branch-by-register-value, the programmer can specify the number of the code block starting at the branch target address. Second, a dedicated field (CPF: Cache Priority Flag) in the PSR (Processor Status Register) indicates the number of the code block whose priority is currently set to one, as is shown in Fig. 2-(b). The priority of the current code block is determined to be one if the CPF bit corresponding to the number of the current code block is set. This scheme is advantageous because it can change the priority dynamically during the program execution.

3. BLOCK PRIORITIZATION METHODS

We developed a total of five methods, including static and dynamic, to determine the priority of a given code block. The methods are categorized into *static* or *dynamic* priority methods according to whether the priority of a code block can be changed during the execution time or not.

3.1. Static priority methods

We propose two static block prioritization methods that need a user specified parameter γ.

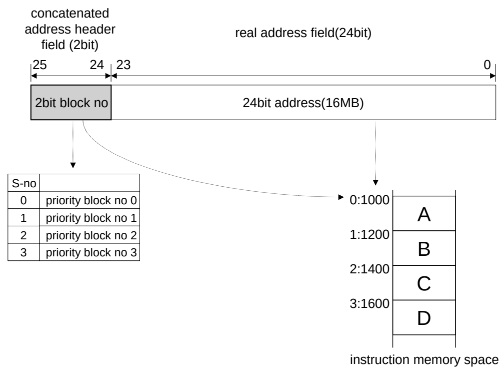
- **AC** (Access Count) method: This method utilizes the profiling information for which all the basic blocks in a program are sorted by the number of times each basic block is executed. Basic blocks with the rank within the top γ percent are given the priority of 1.
- **RR** (Reuse Ratio) method: We first classify the memory reference trace by its cache index. Then, for each cache line, we count the number of the cases when the line is accessed again within a window of fixed size since its last access, as illustrated in Fig. 3. This cache line ‘reuse’ count is summed up on a per-basic-block basis. Finally, all the basic blocks are sorted by the reuse ratio, i.e., reuse count divided by the total access count, and the top γ percent basic blocks are given the priority of 1.

3.2. Dynamic priority methods

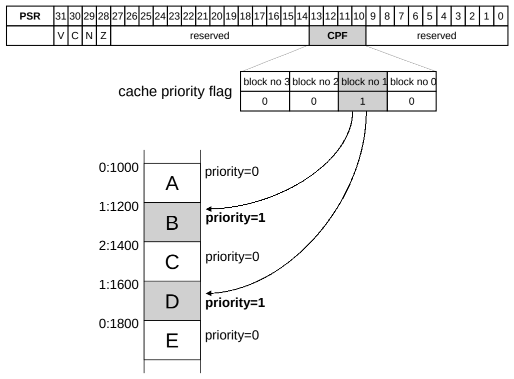
With the static priority approaches, it is possible that some code blocks of priority-1 can remain on the cache even when they are no more needed. To alleviate this problem, we introduce dynamic priority methods so that the priorities of unnecessary cache lines can be turned off and eventually have them replaced sooner.

- **NDP** (N De-Prioritization) method: In this method, a code block with priority-1 is deprioritized when it is cached but not accessed for the duration of N cycles. For our experiments, 10000 is chosen for N.
- **LDP** (LRU De-Prioritization) method: If a cache block is the LRU but not victimized because it is prioritized, then its priority is reset. This corresponds to giving a second chance to a prioritized LRU victim, and has the benefit of being simple enough to be implemented even in hardware.
- **Hand** (Hand tuning) method: Since that most multimedia applications show regular behavior at run time, application programmers can conduct an optimal tuning of when to prioritize or deprioritze a code block. We believe that this approach, though it may be time-consuming and tedious, will give the best performance in the end.

Note that these dynamic priority methods can utilize the static methods to determine the initial priority value of each code block.



(a) Current block number specified in an address field



(b) Active priority block number specified in the PSR

Fig. 2. Block priority specification

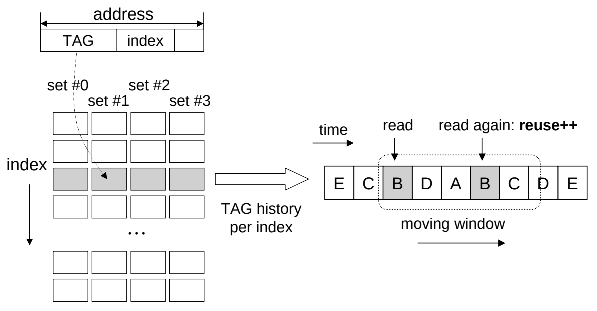


Fig. 3. Counting cache line reuse in the RR method

Table 1. Description of benchmarks and simulation parameters

Description	Program Size	# of Mem. References	Warm Up Period
H.263 Encoder	23KB	22M	100K
AC-3 Decoder	14KB	12M	100K
Mixed (H.263+AC-3)	37KB	50M	100K

4. EXPERIMENTAL RESULTS

We used an H.263 video encoder [7] and an AC-3 audio decoder [8] for performance evaluation. We also employed a mixed trace of H.263 and AC-3, in order to investigate the performance when multiple applications are running simultaneously. This trace was generated under the assumption that the context is switched every 10 million cycles, which corresponds to 33 ms on a 300 MHz processor. All traces used for these experiments are tens of millions of memory references long, as presented in Table 1. As for the cache organization, we will concentrate on 4-way associative caches with the line size of 32 bytes in this paper.

For the AC and the RR methods, the experiment results were obtained as the γ value was varied in 10 % increments from 10 % to 100 %. Because a small number of basic blocks tends to dominate the basic block access count, as is suggested by the 90/10 locality rule [6], the γ value was additionally varied in 1 % increments from 1 % to 20 % for the AC method.

Using the benchmarks described above, we evaluate the performance of the proposed caching scheme with various block prioritization methods and show the experimental results in Fig. 4. In all the graphs of Fig. 4, the solid lines represent the cache miss ratio and the dashed lines represent the cache miss ratio normalized by that of the LRU scheme. For the AC and the RR methods, only the cases with the lowest miss ratio are presented in the graphs. Their corresponding γ values are shown in Table 2.

The experimental results of static block prioritization methods with H.263 and AC-3 are presented in Fig. 4-(a) and (b). The results show that the AC method performs slightly better than the RR method in case of H.263, and much better in case of AC-3. The geometric means of the normalized cache miss ratio of the AC method are shown to be 83.0 % (H.263) and 81.6 % (AC-3), respectively. In comparison, those values of the RR method are 83.1 % (H.263) and 91.5 % (AC-3), respectively. Interestingly, the performance improvement is more pronounced when the cache size is around the 'knee' of the cache miss ratio curve, that is to

Table 2. γ values chosen for Fig. 4 (shown in percent)

Benchmark	512	1K	2K	4K	8K	16K	32K
H.263, AC	65	80	1	8	25	60	X
H.263, RR	40	80	5	10	30	60	X
AC-3, AC	4	7	8	19	25	X	X
AC-3, RR	10	20	25	30	40	X	X
Mixed, AC	65	7	60	12	30	60	40
Mixed, RR	40	20	95	15	30	60	75

(Note: 'X' means that all γ yield same results)

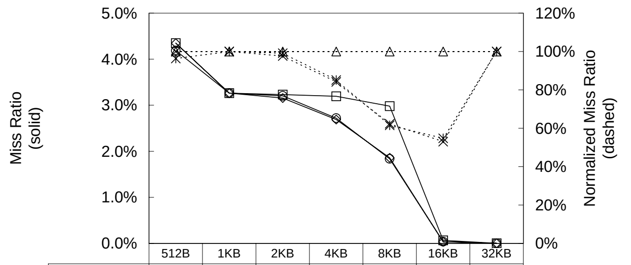
say, when the cache is neither too large nor too small. For example, the geometric mean of the normalized miss ratio when the cache size varies from 4 KB to 16 KB is shown to be 65.3 % in case of H.263 with the AC method. The lowest normalized miss ratios we can obtain are 53.0 % (H.263, 16 KB) and 64.1 % (AC-3, 8KB), both with the AC method.

The experimental results of static block prioritization methods with the mixed trace of H.263 and AC-3 are shown in Fig. 4-(c). In this figure, other than the AC and the RR methods, we show another case named as *Best*. Because the two applications can be independently prioritized, a wide range of experimental results is obtained by applying different γ values and methods to each program. The case that yields the best performance among them is chosen as the 'Best'. In general, it is expected that the cache miss ratio will be increased when multiple programs are in execution because of cache pollution. Our results show that the proposed caching scheme is more effective under such multiprogramming situations.

Figure 4-(d) shows the experimental results of dynamic priority methods. Only the case of H.263 is shown because of limited paper space. The NDP and LDP methods were implemented on top of one of the static methods with the γ value shown in Table 2. For the Hand methods, we determined the priority of each function of the H.263 application code by intuition. The results show that the NDP method obtained a slight gain over the static methods in terms of the normalized cache miss ratio that results in 82.4 % when compared to 83.0 % of the AC method. However, the LDP and Hand methods performed worse with the normalized cache miss ratio of 87.0 % and 94.3 %, respectively, although they are still better than the normal LRU algorithm. Looking at these results, we note that while dynamic deprioritization schemes can improve the performance of the static priority methods, they can actually perform worse if not implemented properly. Moreover, it is also suggested that the proposed static methods can achieve excellent performance with less efforts compared to the manual prioritization method which depends on the experience and know-how of a programmer.

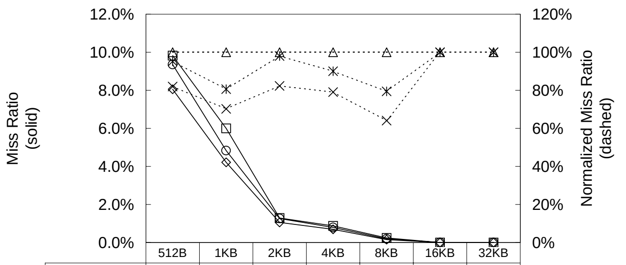
5. CONCLUDING REMARKS

In this paper, we have investigated the potential for improving cache performance by providing a new caching scheme for embedded multimedia processors, which allows the programmer to selectively prioritize parts of cache blocks. Several static and dynamic block prioritization methods for the proposed caching scheme are also developed and their performances are evaluated using a few real multimedia applications. The experimental results show that the cache miss ratio can be reduced up to nearly a half of that of



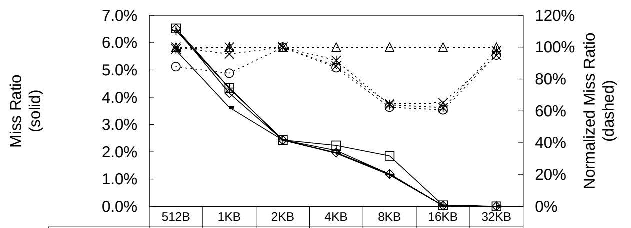
	512B	1KB	2KB	4KB	8KB	16KB	32KB
LRU	4.3483%	3.2622%	3.2313%	3.1941%	2.9799%	0.0650%	0.0018%
AC	4.3458%	3.2621%	3.1552%	2.6881%	1.8587%	0.0344%	0.0018%
RR	4.1882%	3.2620%	3.2027%	2.7190%	1.8342%	0.0356%	0.0018%
Norm. LRU	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Norm. AC	99.9%	100.0%	97.6%	84.2%	62.4%	53.0%	100.0%
Norm. RR	96.3%	100.0%	99.1%	85.1%	61.6%	54.7%	100.0%

(a) H.263, static



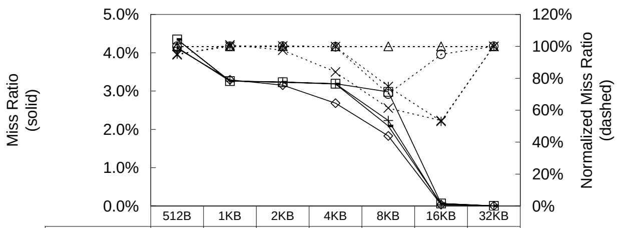
	512B	1KB	2KB	4KB	8KB	16KB	32KB
LRU	9.8226%	5.9987%	1.2871%	0.8787%	0.2401%	0.0004%	0.0004%
AC	8.0545%	4.2142%	1.0596%	0.6947%	0.1538%	0.0004%	0.0004%
RR	9.3534%	4.8336%	1.2617%	0.7912%	0.1908%	0.0004%	0.0004%
Norm. LRU	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Norm. AC	82.0%	70.3%	82.3%	79.1%	64.1%	100.0%	100.0%
Norm. RR	95.2%	80.6%	98.0%	90.0%	79.5%	100.0%	100.0%

(b) AC-3, static



	512B	1KB	2KB	4KB	8KB	16KB	32KB
LRU	6.5200%	4.3366%	2.4326%	2.2359%	1.8497%	0.0419%	0.0023%
AC	6.5188%	4.1486%	2.4320%	1.9714%	1.1910%	0.0273%	0.0022%
RR	6.4447%	4.3366%	2.4326%	2.0505%	1.1800%	0.0260%	0.0022%
Best	5.7190%	3.6258%	2.4315%	1.9470%	1.1528%	0.0254%	0.0022%
Norm. LRU	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Norm. AC	100.0%	95.7%	100.0%	88.2%	64.4%	65.1%	94.9%
Norm. RR	98.8%	100.0%	100.0%	91.7%	63.8%	62.1%	97.7%
Norm. Best	87.7%	83.6%	100.0%	87.1%	62.3%	60.6%	95.0%

(c) Mixed, static



	512B	1KB	2KB	4KB	8KB	16KB	32KB
LRU	4.3483%	3.2622%	3.2313%	3.1941%	2.9799%	0.0650%	0.0018%
NDP	4.1174%	3.2931%	3.1552%	2.6833%	1.8281%	0.0348%	0.0018%
LDP	4.1390%	3.2621%	3.2391%	3.1907%	2.2310%	0.0344%	0.0018%
Hand	4.3483%	3.2622%	3.2313%	3.1854%	2.0898%	0.0618%	0.0018%
Norm. LRU	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Norm. NDP	94.7%	100.9%	97.6%	84.0%	61.3%	53.5%	100.0%
Norm. LDP	95.2%	100.0%	100.2%	99.9%	74.9%	53.0%	100.0%
Norm. Hand	100.0%	100.0%	100.0%	99.7%	70.1%	95.0%	100.0%

(d) H.263, dynamic

Fig. 4. Experimental results and performance comparison

the normal LRU replacement scheme although the improvement depends on the cache size. This caching scheme can also be applied for the data cache with minor modifications.

There are several possible areas for future work. Additional experiments considering different cache organizations such as varying associativity and line size are currently in progress. The development of more effective dynamic prioritization methods leaves room for further research.

6. REFERENCES

[1] *Buyer's Guide to DSP Processors*, Berkeley, CA: Berkeley Design Technology, Inc., 1999.

[2] E. A. Lee, "Programmable DSP architectures: Part I," *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19, Oct. 1988.

[3] E. A. Lee, "Programmable DSP architectures: Part II," *IEEE ASSP Magazine*, vol. 6, no. 1, pp. 4–14, Jan. 1989.

[4] *TMS320C62xx CPU and Instruction Set Reference Guide*, Houston,Texas Instruments Inc., 1997.

[5] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1996, pp. 165–175.

[6] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Jan. 1990.

[7] ITU Telecom. Standardization Sector of ITU, "Video coding for low bitrate communication," *ITU-T Draft Recommendation H.263*, Mar. 1996.

[8] United States Advanced Television Systems Committee (ATSC), *Digital Audio Compression Standard (AC-3)*, Doc. A/52, Dec. 1995.