Journal of Circuits, Systems, and Computers
Vol. 18, No. 6 (2009) 1081–1092

World Scientific
www.worldscientific.com

AUGMENTED FIFO CACHE REPLACEMENT POLICIES FOR LOW-POWER EMBEDDED PROCESSORS

SANGYEUN CHO* and LORY AL MOAKAR†

Department of Computer Science, University of Pittsburgh,
5407 Sennott Square, 210 s. Bouquet St, Pittsburgh, PA 15260, USA
**cho@cs.pitt.edu*
†*lorym@cs.pitt.edu*

Revised 5 May 2009

This paper explores a family of augmented FIFO replacement policies for highly set-associative caches that are common in low-power embedded processors. In such processors, the implementation cost and complexity of the replacement policy is as important as the cache hit rate. By exploiting the cache hit way information between two replacements, the proposed replacement schemes reduce cache misses by 1% to 18% on average depending on the cache configuration, compared with the conventional FIFO policy. The proposed schemes come at a small implementation cost of additional state bits and control logic. The reduction in cache misses directly translates into data access energy savings of 1% to 15% on average, depending on the cache configuration. Our work suggests that there is room for improving the popular FIFO policy at a small cost.

Keywords: Cache memory; energy consumption.

1. Introduction

Processor performance is heavily dependent on the timely delivery of data from memory. To effectively fill the widening speed gap between a processor and main memory, cache memory has been used in virtually all high-performance processors.[19] Due to the benefit of the cache memory, an increasing number of low-power embedded processors are employing the cache memory.[4, 18, 20]

In recent embedded processors, highly associative cache memories based on the CAM-tag organization have become popular.[23] For example, the StrongARM processor[16] and Intel's XScale[11] have 32-way set-associative caches for instructions and data. ARM's 920T core has 64-way set-associative caches.[4] In a highly set-associative cache, the FIFO replacement policy is typically used. Although the LRU policy is in general, superior to the FIFO policy in terms of the average hit rate,[19] the cost and complexity of the LRU policy is much larger than that of the FIFO policy, thus discouraging its use in highly set-associative caches.

In this paper, we explore three augmented FIFO replacement schemes for highly set-associative caches (for higher hit rates), by incrementally adding small storage

and logic overheads to the basic FIFO scheme. The basic idea behind the proposed schemes is that considering cache block usage patterns as well as the FIFO order will reduce the chance of replacing a hot cache block. Our experimental result shows that the proposed simple extensions to the FIFO policy can lead to sizable miss reductions; the reduction of cache misses was at least 11% on average when the cache size is 8 kB. Our results demonstrate that there is room for improving the basic FIFO policy, subject to the hardware budget and the performance goal.

In the remainder of this paper, we will first give a brief background about the cache replacement schemes and summarize the existing body of research on cache replacement and cache energy reduction (Sec. 2). Section 3 then describes the proposed augmented FIFO schemes and their main trade-offs. Section 4 presents a quantitative evaluation, followed by conclusions in Sec. 5.

2. Background

There is a large body of research and development work related with the cache replacement policy.[19] Today, the most popular policies found in real products are *random*, *FIFO*, *LRU* and *PLRU* (Pseudo LRU). They differ both in the complexity of the design, state update time, and average hit rate.

2.1. *Replacement policy in low-power caches*

There are two aspects of the replacement policy choice for low-power embedded processor cache design. The first is the miss rate. A cache miss causes performance penalty and large energy consumption, as the missed access can make its way to off-chip memory such as DRAM or flash memory. Our estimation (in Sec. 4.2.2) indicates that the energy consumption due to a cache miss can be three orders of magnitude larger than that of a cache hit in current and future deep submicron CMOS technology. Energy is also wasted during the processor idle time. Hence, choosing a replacement policy that leads to fewer misses not only improves the average program execution time but also the overall energy efficiency of the processor.

Second, the hardware implementing the replacement policy should be simple and efficient. Especially, the required state bit maintenance actions should be minimized, as well as the memory capacity needed to store the state bits. In embedded processors with a shallow pipeline, handling frequent events such as cache hits need to be done within a single clock cycle. The hardware complexity issues in cache replacement policy implementation have resulted in the popularity of relatively simple random and FIFO policies, even in high-performance processors. Recent highly set-associative caches we considered in this paper exclusively employ the FIFO replacement policy.[4, 11, 16] Another important consideration in the embedded processor cache design is the support for lock-down.[4] "Locked" cache blocks are not replaced on a cache miss; the memory contents in those cache blocks are guaranteed to be found in the cache, resulting in constant and fast memory access latency for

them. The FIFO policy naturally yields an efficient cache lock-down implementation by limiting the value of the victim pointer within two bounding values.[4] The LRU and its variant policies would require more complex control logic.

2.2. *Related work*

Due to the complexity of implementing the LRU policy, various simpler replacement policies have been proposed. The PLRU policy is one of the most popular policies used in the place of LRU.[2,6,14] PLRU typically uses a tree structure ("PLRUt") or per-block MRU bits ("PLRUm") to determine the victim block on a cache miss. When a cache miss occurs, the victim cache way is determined by decoding the bits associated with the tree nodes, beginning from the root of the tree, or by locating a cache block with its MRU bit not set. It still requires relatively complex state updates on a hit, however, because the tree bits or the MRU bits must be properly updated based on the hit way.

Deville[8] presents a replacement policy which uses a counter per cache set to partition "old" and less critical cache blocks and "busy" and more important ones. On a cache miss, the counter points to the victim block to replace. This scheme performs reasonably well in caches with low associativity, but may suffer in highly associative caches, even worse than FIFO. Our policies build on and performs better than FIFO robustly. Their later work[9] develops a more complex phase-based scheme to remedy the weakness in their previous scheme. It keeps information about the access patterns in previous phases. A global counter is used to keep track of the phase changes, and MRU bits are used to record access patterns.

There are "intelligent" cache replacement strategies that combine different replacement policies by taking into account previous cache access behaviors and dynamically configuring the cache replacement policy. Altman *et al.*[3] proposed a genetic algorithm-based strategy to tune the cache to a given workload by using an appropriate combination of LRU, FIFO, or other information. Their strategy leads to hit rate improvement over the LRU policy. Khalid[13] presented a neural network-based algorithm that uses back-propagation neural network (BPNN) to guide block replacement decisions. The key to their algorithm is to identify and subsequently discard the dead blocks in cache sets such that other cache blocks that may be used in the future can be kept in the cache for longer time. Manzoul[15] proposed two fuzzy inference systems for cache replacement that considers information on cache block age and cache block access frequency to select a victim. They employ 18 or 34 rules for inference in their proposed systems. While these intelligent cache replacement strategies were shown to be effective under the workloads and cache configurations examined, their applicability is limited in low-cost embedded processors with a highly set-associative cache because they require more state bits to maintain and their effectiveness tends to narrow when the associativity is high.

Finally, highly set-associative caches are usually designed with a circuit for fast associative search in the tag memory — *Content Addressable Memory (CAM).*[21–23]

Studies show that smart CAM circuit designs enable highly set-associative caches to be power efficient. Our work in this paper uses this already energy-optimized, highly set-associative cache design as our target of optimizing further. Because such designs are and will be popular in low-power embedded processors,[11, 16] our findings in this paper carry practical importance.

3. Augmented FIFO Policies

In this section, we present three new FIFO-based replacement schemes: *Move-on-Hit FIFO* (MH-FIFO), *Set-on-Hit FIFO* (SH-FIFO) and *Counter-Based FIFO* (CB-FIFO). These schemes differ in how they capture cache access patterns (i.e., hit way information) and how they exploit the information in making a replacement decision. They incrementally add more state bits to record the information.

3.1. *Move-on-Hit FIFO (MH-FIFO)*

Our first scheme, MH-FIFO, is the simplest of all with no additional state bits added to the baseline FIFO scheme. In the baseline FIFO scheme, each set has a counter of $\log W$ bits pointing to the next victim block in the set (i.e., "victim pointer"[4]), where W is the number of cache blocks per set or simply the associativity of the cache. This counter is incremented whenever there is a cache miss in the FIFO scheme. On the other hand, in MH-FIFO, this counter is also incremented if the cache block that is hit by the current access coincides with the counter value. This is illustrated in Fig. 1(a). When a cache access hits in the block pointed by the FIFO pointer (case 1), the FIFO pointer moves to the next cache block in the set. When a cache access hits in the block not pointed by the FIFO pointer (case 2), the FIFO pointer remains at its current position.

MH-FIFO salvages the "oldest" cache block in the target cache set from being replaced if it is still in use. By comparison, FIFO simply discards the oldest cache block on a replacement. Intuitively, MH-FIFO will reduce the probability of replacing a cache block that will be used again soon, thereby decreasing the miss rate. Implementing MH-FIFO does not incur additional state bits. However, it introduces a match operation between the current FIFO counter value and the hit way number on a cache hit so that the FIFO counter value can be updated if they actually match. This match operation can be efficiently implemented with a $\log W$-bit comparator or an array of a single-bit comparator (an AND gate) if the counter value is pre-decoded.

3.2. *Set-on-Hit FIFO (SH-FIFO)*

Our second scheme, SH-FIFO, attaches a *use bit* to each cache block. The use bit is set when a cache access hits in the associated cache block. The use bits in a set are reset all together when a cache access misses in the set. Therefore, the use bits in aggregation record the recent history of access patterns (since the last cache miss).

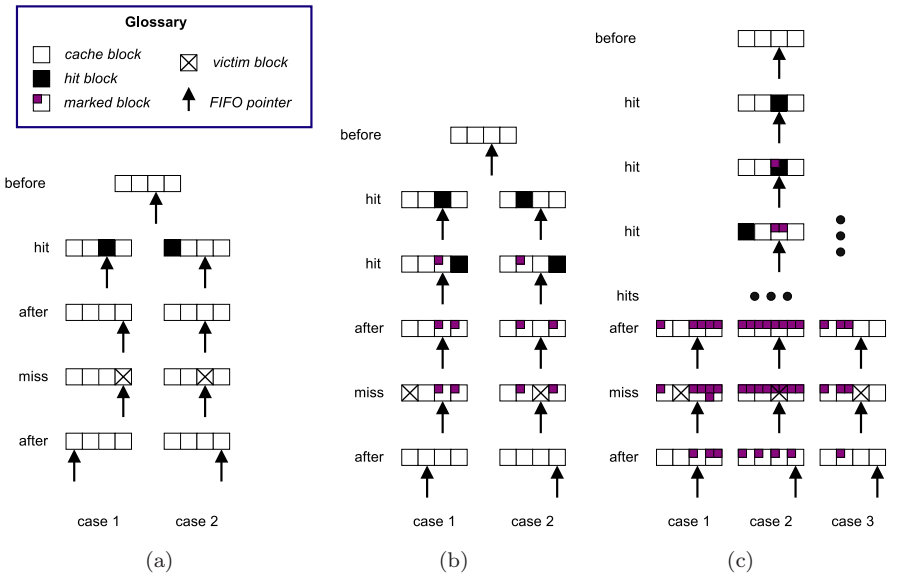


Fig. 1. (a) Operation of MH-FIFO, (b) Operation of SH-FIFO, (c) Operation of CB-FIFO. From an initial state ("before"), a series of cache hit and miss events occur. Multiple cases are shown to expose different operational behaviors of each policy.

Given the use bits, SH-FIFO determines the victim block by selecting a cache block that has not been accessed since the last replacement in its set. Starting from the cache block pointed by the FIFO victim pointer, SH-FIFO searches for the first block whose use bit is not set. This is illustrated in Fig. 1(b). If no cache block with its use bit unset can be found (i.e., all the cache blocks have been accessed since the last miss), the victim block is simply the cache block pointed by the FIFO pointer at first. SH-FIFO does not victimize a cache block that was accessed at least once if there is a cache block that was not. Like MH-FIFO, SH-FIFO reduces the probability of replacing a cache block that will be used again soon.

An SH-FIFO implementation incurs more state bits than FIFO, namely, a use bit per cache block. Unlike LRU (or PLRU), however, updating a use bit in SH-FIFO does not require reading and modifying other state bits. When a hit way is known, the corresponding use bit is simply set. Implementing the update logic for SH-FIFO is, therefore, more straightforward. Moreover, because the victim choice in SH-FIFO is still based on the FIFO pointer, the cache lock-down support can be easily implemented by assigning the lower-bound value to the FIFO pointer (to skip the locked-down ways) whenever it wraps around.[4]

3.3. *Counter-Based FIFO (CB-FIFO)*

Our third and last scheme is CB-FIFO, which uses a two-bit counter to register the number of accesses to a cache block. CB-FIFO is similar to SH-FIFO in that it records the access history of individual cache blocks. CB-FIFO adds more

information than SH-FIFO; it counts and records the number of accesses that hit in each cache block. The recorded information places different "weights" among the cache blocks in a set when a victim block is to be selected. In fact, SH-FIFO can be thought of as a special case of CB-FIFO where each counter is a single bit.

Figure 1(c) depicts how CB-FIFO works in three different cases. On a cache hit, the counter associated with the hit cache block is incremented. Each counter is a saturating counter; if a counter has already reached the maximum value, it does not change its value on the next increment. On a cache replacement following a miss, a victim block whose access count is the smallest is selected as the victim. All the counter values in the set are then decremented, or alternatively, reset. While this action is more complex than the replacement action in the baseline FIFO policy, it can be done easily by grouping W cache blocks into four value groups (0, 1, 2, and 3) with parallel comparators. We will discuss the impact of using more than two bits per cache block in CB-FIFO and the impact of resetting the counter values in Sec. 4.

Like SH-FIFO, CB-FIFO guarantees that it never replaces a cache block that was accessed recently if there is a cache block that has not been used. Because CB-FIFO attaches two state bits to each cache block, it incurs more storage overhead than SH-FIFO. However, the number of bits to be inspected and updated is limited to two bits on a cache hit, and the update logic implementation is straightforward as in the case of SH-FIFO.

Table 1. Different replacement schemes' number of state bits, action on a hit and action on a miss. The numbers in ( ) are the necessary state bits for a 32-way 8 kB cache with 32 B blocks.

Scheme	Description
LRU	$N_{sets} \times W \times \log W$ (1280 bits) Update LRU list Update LRU list
FIFO	$N_{sets} \times \log W$ (40 bits) Do nothing Advance FIFO counter
MH-FIFO	$N_{sets} \times \log W$ (40 bits) Advance FIFO counter if it matches hit way Advance FIFO counter
SH-FIFO	$N_{sets} \times (\log W + W)$ (296 bits) Set use bit of hit block Search for victim and set FIFO counter
CB-FIFO	$N_{sets} \times (\log W + 2 \cdot W)$ (552 bits) Increment counter of hit block Search for victim and set FIFO counter
PLRU (PLRUt)	$N_{sets} \times (W - 1)$ (248 bits) Update tree bits Update tree bits

3.4. *Discussion*

The three augmented FIFO schemes we presented add a small overhead to the baseline FIFO scheme. Table 1 summarizes each scheme's state storage requirement and the necessary actions on a hit or a miss. For comparison, we included a tree-based PLRU in the table.

It is shown that the required state bit storage of MH-FIFO, SH-FIFO or CB-FIFO is much smaller than that of LRU. It is also clear that their state update actions on frequent cache hits are significantly simpler than the LRU list update action. To assess the complexity of the state update logic, we implemented a cache controller using a product-grade 130 nm standard cell library. Because new state bits (e.g., $(counter + 1)$ in MH-FIFO) can be computed *a priori* even before the hit information is provided, the state bit update latency is found to never extend the existing critical path. Our cache controller has about 500 bits of storage (flip-flops) to implement controller state bits and various buffers (e.g., a 4-entry write-back buffer) to interface the processor and the memory bus. In the case of FIFO, MH-FIFO, SH-FIFO and PLRU, the number of replacement-related state bits is roughly equivalent to or smaller than the number of bits needed to implement the cache controller and the cache datapath. For an 8 kB 32-way cache design, the cache area difference relative to FIFO is, 0%, 3.7%, and 3.0% for MH-FIFO, SH-FIFO and PLRU, respectively. The area of full LRU and SH-FIFO is 17.9% and 7.2% larger than that of FIFO.

Lastly, PLRU has a lower state storage requirement than SH-FIFO and CB-FIFO. However, incorporating support for cache locking was more complex in PLRU because state update actions on hits must also consider locked-down cache blocks.

4. Quantitative Evaluation

4.1. *Experimental setup*

To evaluate the performance of different cache replacement policies, we use a detailed execution-driven simulator based on sim-outorder in the SimpleScalar tool set (v4.0)[5] to model a processor configured to resemble a real design.[16] The modeled processor has a single-issue in-order pipeline and uses a 266 MHz clock. The off-chip SDRAM is clocked at half the processor clock (133 MHz). Table 2 summarizes the key processor parameters.

Table 2. Summary of the simulated processor.

Pipeline	Single-issue in-order
Branch prediction	2 k-entry combined branch predictor
I-cache	4–16 kB, 8/16/32-way SA, 32 B block, 1-cycle latency
D-cache	4–16 kB, 8/16/32-way SA, 32 B block, 1-cycle latency
Main memory	SDRAM, 32-bit bus, 24 processor cycles for cache block transfer

Table 3. Summary of the benchmark programs.

Name	Description
cjpeg	jpeg encoder; ppm to jpeg
djpeg	jpeg decoder; jpeg to ppm
ispell	spell checker
mp3decode	mp3 audio decoder; mp3 to wave
mp3encode	mp3 audio encoder; wave to mp3
rijndaeld	aes decryption
rijndaele	aes encryption
rsynth	text to speech synthesis
tiff2rgba	image format conversion

For workload, we use nine programs from the MiBench suite,[10] summarized in Table 3. We picked programs based on their different miss rates, ranging from $\sim$0.4% (*rsynth*) to 12.6% (*rijndaele*) when a 4 kB 8-way cache is used. Programs were compiled to target the ARM ISA using gcc 2.95.2 with the `-O2` optimization flag. Programs were run in their entirety.

4.2. *Evaluation results*

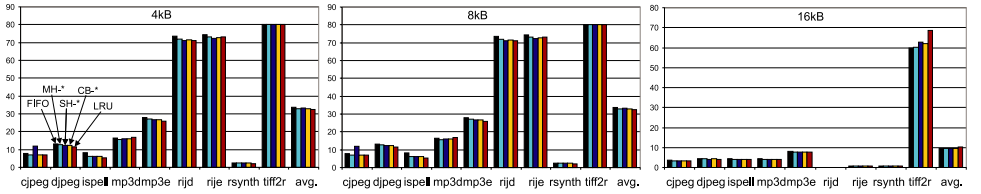
4.2.1. *Cache misses*

Figure 2 shows the number of cache misses per thousand instructions (MPKI) using different replacement policies. Table 4 shows the maximum and the average reduction in cache misses under each different cache configuration, relative to the conventional FIFO policy.

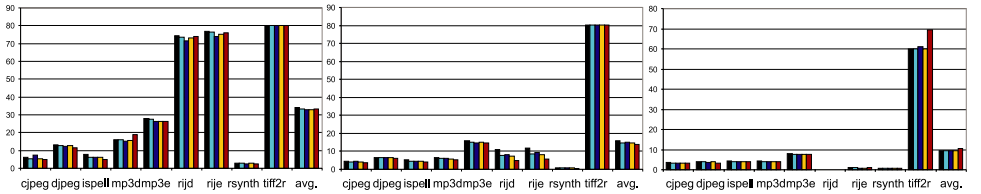
Although the absolute reductions vary depending on the application and the particular cache configuration used, the proposed augmented FIFO schemes are consistently able to reduce cache misses, compared with the conventional FIFO scheme. It is also demonstrated that adding more state bits leads to more reductions in general. Almost always, CB-FIFO outperforms SH-FIFO and SH-FIFO reduces more misses than MH-FIFO. Most of the time, LRU achieves the best results. We have examined both PLRUt and PLRUm; however, they performed worse than the presented schemes (including FIFO) and especially poorly when running media-processing benchmarks such as *cjpeg*, *djpeg*, and *mp3**. Hence, we do not further consider PLRU.

There are several programs and configurations exhibiting a different result than the above general description. SH-FIFO performs poorer than the FIFO or the other augmented FIFO schemes in *cjpeg*. *rijndaeld* and *rijndaele* perform better under the SH-FIFO and the CB-FIFO schemes than LRU when the cache size is 4 kB and the associativity is at least 16. Interestingly, LRU performed not as well as all the other schemes for *tiff2rgba* when the cache size is 16 kB, regardless of the associativity.

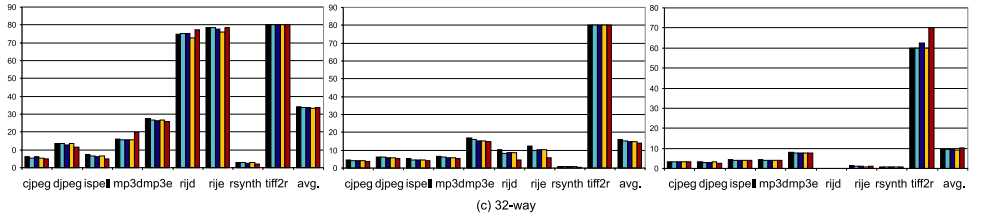
In CB-FIFO, employing counters with more than two bits does not lead to significantly better results and is not justified. In certain cases, miss rates can be



(a) 8-way



(b) 16-way



(c) 32-way

Fig. 2. Misses per thousand (kilo) instructions (MPKI). Three cache sizes (4 kB, 8 kB and 16 kB from left) are shown for three associativities (8-way, 16-way and 32-way from above).

negatively affected because it takes more time to replace a cache block that is no longer in use. Resetting counter values instead of decrementing them also leads to mixed results, with a slightly worse result on average.

It is interesting to observe that the media-type benchmarks (*cjpeg*, *djpeg*, *mp3decode*, *mp3encode*, and *tiff2rgba*) show somewhat different results compared with the rest of the benchmarks. With a small cache size (4 kB), CB-FIFO performs better than SH-FIFO or MH-FIFO for the media-type benchmarks. However, all the augmented FIFO schemes perform very similarly when the cache size is increased to 16 kB. On the other hand, CB-FIFO and SH-FIFO perform better than MH-FIFO for the other, non-media-type benchmarks especially when the cache size is large (16 kB).

4.2.2. *Energy consumption*

Today, the energy consumption related with a cache hit is much different from the energy consumption due to a cache miss. As new, low-power technologies are

Table 4. Average (maximum) miss reduction (%), relative to FIFO.

Configuration	MH-FIFO	SH-FIFO	CB-FIFO	LRU
4 kB, 8-way	6 (22)	1 (22)	7 (26)	10 (33)
4 kB, 16-way	5 (20)	5 (22)	7 (22)	10 (35)
4 kB, 32-way	4 (15)	6 (21)	4 (12)	10 (36)
8 kB, 8-way	12 (26)	12 (22)	14 (29)	20 (47)
8 kB, 16-way	13 (29)	11 (25)	15 (32)	25 (54)
8 kB, 32-way	11 (22)	12 (18)	12 (18)	26 (56)
16 kB, 8-way	10 (50)	18 (74)	14 (59)	15 (62)
16 kB, 16-way	8 (32)	14 (49)	17 (59)	12 (48)
16 kB, 32-way	4 (8)	6 (19)	8 (34)	6 (24)

continuously deployed, the relative energy discrepancy between a cache hit and a cache miss will likely increase. Using CACTI 4.2,[7] we estimate that a read hit in a 8 kB 32-way CAM-tag cache consumes about 0.156 nJ at 0.887 V and about 0.199 nJ at 1 V assuming a 70 nm technology.[a] On the other hand, accessing off-chip memory composed of four 128 Mbit×8 SDRAM (similar to K4S280832B-TC1L[17]) to fetch a 32 B cache block can consume as much as 690.1 nJ. The estimation assumes a 16.35 pF load capacitance per data bus line, a 13.75 pF load per address bus line[12] and a 3.3 V interface voltage. Lowering the interface voltage to 1.8 V and halving the load capacitance result in much reduced total energy of 302.5 nJ, still much higher than the hit energy.

The bottom line of our estimation is that a cache miss can easily consume three orders of magnitude more energy than a cache hit, depending on the cache configuration (i.e., set-associativity in the CAM-tag structure[23]) and the voltage level used for the cache memory, the external DRAM memory, and the chip interface. Because of the large difference in the cache hit energy and the cache miss energy, the cache miss rate will determine the energy consumption in the memory hierarchy. In our experimental setup, this suggests that unless the cache miss rate is in the range of as low as 0.01% to 0.1%, the cache miss energy, not the cache hit energy, will dominate the total data access energy.

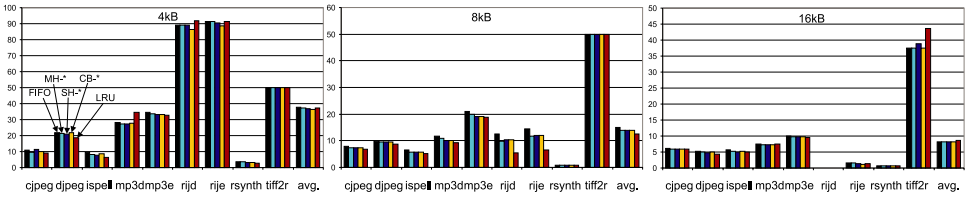


Fig. 3. Average energy consumption per data memory access (nJ/access). Results for three cache sizes (4 kB, 8 kB and 16 kB from left) and the associativity of 32 are shown.

[a]CACTI 3.2 gives a more conservative number: 0.556 nJ at 0.9 V.

Table 5. Average (maximum) energy savings (%), relative to FIFO.

Configuration	MH-FIFO	SH-FIFO	CB-FIFO	LRU
4 kB, 8-way	6 (22)	1 (22)	7 (25)	10 (33)
4 kB, 16-way	5 (20)	5 (22)	7 (22)	10 (35)
4 kB, 32-way	4 (15)	5 (21)	4 (12)	10 (36)
8 kB, 8-way	12 (26)	11 (22)	14 (29)	18 (39)
8 kB, 16-way	13 (29)	11 (25)	15 (32)	24 (54)
8 kB, 32-way	11 (22)	12 (18)	11 (18)	26 (55)
16 kB, 8-way	7 (28)	9 (34)	9 (33)	9 (35)
16 kB, 16-way	5 (15)	10 (31)	11 (34)	8 (23)
16 kB, 32-way	4 (8)	5 (18)	7 (33)	6 (23)

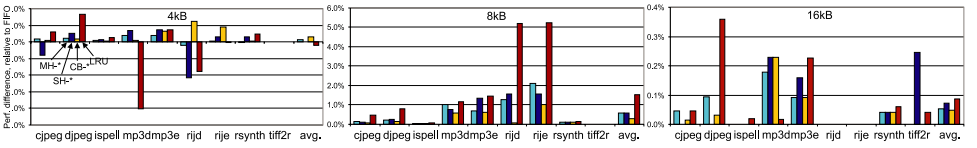


Fig. 4. Performance difference relative to FIFO. Results for three cache sizes (4 kB, 8 kB and 16 kB from left) and the associativity of 32 are shown.

Figure 3 shows the average energy per access under various replacement policies using a 32-way cache configuration. It is clearly shown that the reduced misses using an augmented FIFO scheme translate into reduced energy consumption. The maximum energy reduction was achieved for *rijndaeld* (34%) when a 16 kB 8-way cache was used and for *rijndaele* (34%) when a 16 kB 16-way cache was used. Table 5 shows the maximum and the average energy savings evidenced in each cache configuration.

4.2.3. *Performance impact*

Because of the very low miss rates of the studied programs even with small cache sizes, the performance impact of different replacement policies, in terms of program execution time, is fairly limited. Figure 4 presents the result, showing that the augmented FIFO schemes are robust in improving the program performance.

5. Conclusions

This paper proposed and studied a family of augmented FIFO cache replacement schemes: MH-FIFO, SH-FIFO and CB-FIFO. The proposed schemes build on the conventional FIFO scheme that has been extensively used in recent low-power embedded processor caches having high associativity. Our study shows that the miss rate of the FIFO scheme can be improved at a slight hardware cost, which leads to sizable data memory access energy savings. As the relative energy consumption difference between a cache hit and a miss becomes larger as low-power technologies are deployed, tuning and augmenting the existing cache policies to cut

on cache misses will be highly beneficial. Finally, our experiments demonstrate that the studied schemes are amenable to implementation and can be easily integrated in a FIFO-based cache memory controller without impacting the cache's critical path. When the "golden" LRU scheme is not feasible, the proposed schemes can become an attractive replacement policy choice for a highly set-associative cache design.

References

1. D. H. Albonesi, Selective cache ways: On-demand cache resource allocation, *MICRO*, December 1999.
2. H. Al-Zoubi *et al.*, Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite, *ACM Southeast Regional Conf.*, April 2004.
3. E. R. Altman, V. K. Agarwal and G. R. Gao, A novel methodology using genetic algorithms for the design of caches and cache replacement policy, *ICGA* (1993).
4. ARM Ltd, *ARM920T Tech. Ref. Manual*, DDI 0151C, 2000, 2001.
5. T. Austin, E. Larson and D. Ernst, SimpleScalar: An infrastructure for computer system modeling, *IEEE Comput.* **35** (2002) 59–67.
6. C. Berg, PLRU cache domino effects, *WCET*, July 2006.
7. CACTI 4.2, http://quid.hpl.hp.com:9081/cacti/
8. Y. Deville, A low-cost usage-based replacement algorithm for cache memories, *SIGARCH Newsletter*, December 1990.
9. Y. Deville and J. Gobert, A class of replacement policies for medium and high-associativity structures, *SIGARCH Newsletter*, March 1992.
10. M. R. Guthaus *et al.*, MiBench: A free, commercially representative embedded benchmark suite, *WWC*, December 2001.
11. Intel Corp, Intel XScale Microarchitecture, technical summary (2000).
12. Y. Joo *et al.*, Energy exploration and reduction of SDRAM memory systems, *DAC*, June 2002.
13. H. Khalid, A new cache replacement scheme based on backpropagation neural networks, *SIGARCH Comput. Archit. News*, March 1997.
14. M. Kampe *et al.*, Self-correcting LRU replacement policies, *ACM Comput. Frontiers*, April 2004.
15. M. A. Manzoul, Fuzzy management of cache memories, *Fuzzy Control Systems*, CRC Press, 1994.
16. J. Montanaro *et al.*, A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor, *IEEE JSSC* **31** (1996) 1703–1714.
17. Samsung Semiconductor, K4S280832B 128Mbit SDRAM datasheet, http://www.samsungsemi.com.
18. NXP Semiconductor, Nexperia PNX1300 series, product brochure (2002).
19. A. J. Smith, Cache memories, *ACM Comput. Surveys* **14** (1982) 473–530.
20. Texas Instruments, TMS320C6211 fixed-point digital signal processors, SPRS073L. 1998, 2005.
21. A. Veidenbaum and D. Nicolaescu, Low energy, highly-associative cache design for embedded processors, *ICCD*, October 2004.
22. C. Zhang *et al.*, A highly configurable cache for low energy embedded systems, *ACM TECS* **4** (2005) 363–387.
23. M. Zhang and K. Asanović, Highly-associative caches for low-power processors, *Kool Chips Workshop*, December 2000.