Proceedings of the 2010 IEEE
International Conference on Information and Automation
June 20 - 23, Harbin, China

A Cache Scheme Based on LRU-Like Algorithm

Dongxing Bao
College of Electronic Engineering
Heilongjiang University
Harbin, Heilongjiang, China
bdx2008@yahoo.com.cn

Xiaoming Li
Microelectronic Center
Harbin Institute of Technology
Harbin, Heilongjiang, China
lixiaoming@hit.edu.cn

***Abstract* - The LRU-like algorithm was proposed for the block management to enhance the direct-mapped cache scheme. Base on the LRU-like algorithm, the Least-Recently-Used Block Filtering cache (LBF cache), which can filter LRU blocks, is designed. From the point of cache usage rate, the LBF cache uses LRU-like algorithm to allocate and replace blocks, and evicts the blocks whose usage shows the least efficiency. After the simulation, the performance of LBF cache shows better performance than typical cache schemes with similar architecture (such as victim cache and assist cache). Comparison also shows, with the same block size of 32 bytes, 9KB LBF cache reduces 26% in average miss rate over the traditional 16KB direct-mapped cache and 53% over the 8KB 2-way set-associative cache.**

Index Terms – Cache, miss rate, LRU algorithm

I. INTRODUCTION

Memory access latencies have been the bottleneck of high performance microprocessors. While the microprocessor industry tries to design higher performance superscalar and VLIW processors, the problem becomes more prominent. The growing disparity between processor and memory performance will take cache misses increasingly expensive. Therefore, cache memory is of increasingly great importance in modern computer systems to reduce memory access penalty and power consumption, and eventually improve the overall system performance. At the same time, as the available chip area grows, it makes sense to spend more resources to allow intelligent control over the on-chip cache management, in order to adapt the caching decisions to the dynamic accessing behavior.

Historically, hit time has been a concern on many caches. This is because the cache read path is of critical importance and often is the most critical speed-path on many microprocessors. Direct-mapped caches (DM caches) are sometimes preferred due to the less complicated organization enables reduced hit time. Furthermore, a direct-mapped cache organization is more conducive to reducing design cycle times, as well as enables a variety of cache size configurations in a relatively easy fashion. However, direct-mapped caches suffer a fundamental drawback in that conflict misses generally become a significant contributor to the total misses, when compared to an n-way (n>1) set-associative cache, especially for small sizes such as 0.5k.

For the performance improvement of microprocessor, access time and hit rate of cache are of the same importance to designers. Therefore, two paths to access data in many cache schemes do help to enhance the performance: one fast path to satisfy most CPU accesses, at the same time a slower path to be utilized to decrease the miss cost. Caches designed based on the method could be classified into four types: decoupled cache, multiple-access cache, augmented cache and multilevel cache [1]. Augmented cache architecture includes two parts, one direct-mapped cache and a much smaller fully-associative cache. Both caches are accessed in parallel with almost the same access latency because of the small capacity of the fully-associative cache.

Gary Tyson et al. [2] showed through application code profiling that a very small number of load/store instructions are responsible for a large percentage of cache misses. Therefore, selective caching could be an important direction for cache management research, especially for data cache management. Most selective caching schemes for optimization were based on the reuse of data in order to hold the reusable data recently in the on-chip cache as long as possible. However, the reusability of the data blocks is difficult to be distinguished for the randomness and uncertainty of application programs.

When cache miss occurs and if the associativity of cache is more than one, the replacement algorithm may be utilized to help decide which block should be replaced to assure the efficiency of cache. Various replacement algorithms have been proposed, in which the Least Recently Used (LRU) algorithm was the most approximate to OPT algorithm. But exact LRU algorithm consumes much hardware resource on chip, and increases the hardware complexity.

In this paper, the LRU-like algorithm is proposed for the augmented cache structure. A new cache structure based on the LRU-like algorithm is presented and simulated also.

II. RELATED WORK

Victim cache [3] improves the performance of direct mapping caches through the addition of a small, full associative cache between the L1 cache and the next level in the hierarchy. When access misses occur, the replaced blocks are transferred into the fully-associative cache. When one hit occurs in the fully-associative cache, the block will be swapped with the block of the same set in the DM cache.

Assist cache [4] seeks to separate the data stream into dynamic temporal and dynamic nontemporal data. Temporal blocks, possessing high reuse probability during a cache lifetime, are given the highest priority for residency in the cache. In order to reduce the effect of nontemporal conflicts, nontemporal data is excluded from the cache as far as possible.

The NTS (non-temporal streaming) cache [5] and the PCS (program counter selective) cache [6] are both proposed by J. A. Rivers et al to improve the performance of the direct-mapped cache with the help of some hardware applications. The former is a location-sensitive cache management scheme and the latter adopts its replacement based on the PC of memory instruction causing the current miss. The NTS cache scheme separates the reference stream into temporal (T) and non-temporal (NT) block references. Blocks are treated as non-temporal until they become temporal in the main cache. Cache blocks that are identified as non-temporal during their last residence in the main cache are allocated to the small fully-associative cache on subsequent requests. The locality information is detected and held by a hardware named NTDU. A block is considered NT if no word in that block was reused during a tour in the cache. A block is considered T if at least one word in the block was reused during a tour. A detection unit (DU) that contains locality information about blocks recently evicted from the cache is used to look ahead the locality characteristics of the fetched block. The block entering cache is checked to see if it has an entry in DU. the block is placed in the main cache if its locality flag bit indicates temporal, or else in the small fully-associative cache. The reuse information is kept via a single bit, the NT bit. If no DU match is found, a new entry is created in the DU and the block is assumed to be temporal and placed into the main cache.

The MAT (memory address table) cache [7] is one cache scheme based on the use of effective addresses like NTS cache. It dynamically partitions cache data blocks into two groups based on their frequency of reuse. Blocks are tagged as either Frequently or Infrequently Accessed. A memory address table is used to track reuse information. The granularity for grouping is a macroblock, defined as a contiguous group of memory blocks that are judged as having the same usage pattern features. Those blocks that are thought of Infrequently Accessed are allocated in a separate small cache.

C/NA(Cacheable/ Non-Allocable) cache [8] tries to examine the cache behavior of each load instruction and identifies the ones with the lowest cache hit rate. These are marked C/NA that means the data references generated by these load instructions will not invoke the allocation policy of the hardware cache management algorithm. It doesn’t mean that the data reference will not be in the cache – the data item might be in the cache if different instructions will allocate on miss references that address.

ABC (Allocation By Conflict) allocation scheme [9] decides where to allocate blocks (into the “main” cache with larger data store or the “buffer” with smaller data store) based on the current tour usage of the block it might replace in “main” cache, rather than on the past tour usage of the incoming block. ABC allocates a block to the “main” cache if the LRU element of its set in “main” cache has not been reaccessed since the last miss reference to that set that did not replace a block in “main” cache; otherwise, the block is allocated to the “buffer”.

III. LRU-LIKE ALGORITHM

When CPU accesses certain data, one word (or less) is accessed from the aspect of CPU. But the operation means one block access for the cache. Thus the locality of data stream may be exploited from various aspects:

1) The CPU aspect: when certain word is accessed, it might be accessed soon, and the words in adjacent region might be accessed soon.

2) The cache aspect: when certain block is accessed, it might be accessed soon, and the adjacent blocks might be accessed soon.

We could further analyze the operative property of the LRU replacement algorithm from the cache aspect. The LRU algorithm makes use of the deduction of the locality principle, i.e. a block most recently used (MRU block) could be accessed again soon, so the block determined to be evicted should be the least-recently-used block (LRU block).

Suppose there are n blocks (B_{n-1}, B_{n-2}……B_m……B_1, B_0) competing for the same set in a cache. Within an executing interval of program, the n blocks were accessed and placed into the cache in the block number sequence from 0 to n-1. As a result, many conflict misses occur if the ways in a set are not enough. According to the LRU algorithm, each way in a set should be attached to a counter to record the accesses recently, and thus the MRU and LRU block could be distinguished. When the set is full, the LRU block should be replaced.

Illuminated by LRU algorithm, we consider adding a buffer to filter the LRU blocks, at the same time holding all the MRU blocks in the DM cache in order to increase the hit rate of this new architecture of augmented cache.

Let’s suppose B_m be in the DM cache, and the other blocks competing for the same set be in the buffer. For all the blocks, the ideal method to utilize LRU algorithm is to attach a counter to each block. But that will increase the hardware cost. Therefore, only one bit for each block in the DM cache is designed to indicate the MRU block. The method is similar to that of LRU counter of 2-way associative cache. When cache access hit in DM cache, the B_m is marked as MRU block. Otherwise, when cache access hit in the buffer, the block of the same set as the needed block in DM cache is marked as LRU block. When the miss block is fetched, the location to refill in is determined by the indicating bit of the corresponding set, i.e. when the bit indicated MRU the block will be filled in the buffer; otherwise it will be filled in the DM cache. Since the LRU blocks on DM cache might not be the true LRU blocks relative to the blocks in the buffer, this replacement strategy is named LRU-like algorithm.

IV. LRU BLOCK FILTERING CACHE SCHEME

According to the LRU-like algorithm, a new augmented cache scheme named Least-Recently-Used Block Filtering Cache (LBF cache) is proposed. Fig. 1 shows the LBF cache structure, which includes a DM cache and a fully associative buffer.

For the DM cache the tag of each block is added one MRU block judge bit (M bit), which is used to detect if the block was accessed recently. The block in DM cache generally obtains relatively more reusability, so the evicted block from DM cache should be kept in L1 cache as long as possible.

The steps of LBF cache for every condition are shown below.

1) Cache hit: CPU accesses the DM cache and the buffer in parallel. If cache access hits in the DM cache, the M bit for the hit block is set to 1. Otherwise, when cache access hit in the buffer, the M bit for corresponding set in DM cache is cleared.

2) Cache miss: The fetched block from next level of memory hierarchy is filled into DM cache if the M bit for the same set of the miss block is 0; otherwise, the block is filled into the buffer.

3) Block transfer: When CPU waits for the required block, the evicted block from DM cache is filled into the buffer.

4) Cache refilling: When a block is refilled into the buffer for the first time, the M bit for the same set in DM cache is cleared to 0; when a block is refilled into the DM cache for the first time, the M bit is set to 1.

Let's examine how effective this scheme will be, for some simple reference patterns also used by McFarling [8] to illustrate his dynamic cache exclusion scheme.

A. Conflict between loops

In this case, there is a conflict between references inside two different loops. If there is such a reference and there is a memory access pattern such as $(A^{10}B^{10})^{10}$, where the superscript denotes the frequency of usage of the particular data word address access.

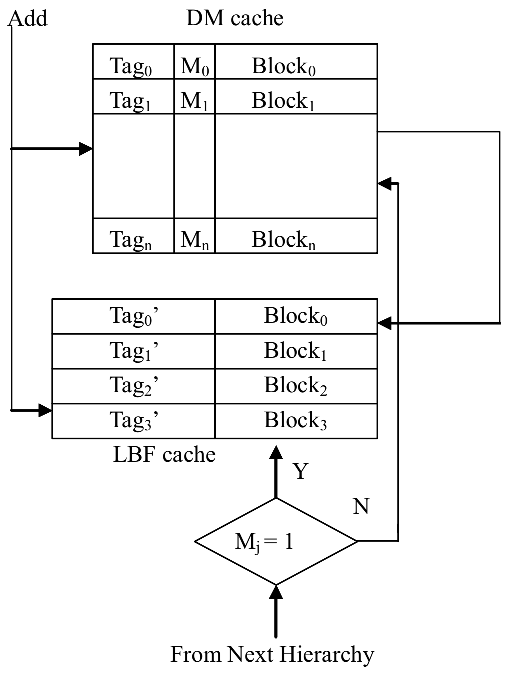

Fig.1 The LBF cache.

The behavior of a conventional direct-mapped cache would be

$$(A_mA_h^9B_mB_h^9)^{10}$$

and the miss rate is

$$M_{DM} = 20/200 = 10\%$$

Now let's consider the behavior of our scheme. When A is refilled into the DM cache, the M bit is set. Then B is requested after nine hits of A, it will be refilled into the buffer. For the next nine cycles, no conflict occurs. On the other hand, if A is refilled into the buffer for the first time, the M bit for corresponding set in DM cache is cleared, so B will be refilled into the DM cache. For both the refilling conditions, the access behavior would be

$$A_mA_h^9B_mB_h^9\ (A_h^{10}B_h^{10})^9$$

and the miss rate is

$$M_{LBF} = 2/200 = 1.0\%$$

Thus LBF cache may gain more hits in this case.

B. Conflict between inner and outer loops

In this case, a conflict occurs between a reference inside a loop with another reference outside the inner loop, and the memory access pattern can be represented as $(A^{10}B)^{10}$. The behavior of a direct-mapped cache would be

$$(A_mA_h^9B_m)^{10}$$

and the miss rate is

$$M_{DM} = 2/11 = 18\%$$

Consider the behavior of our cache scheme in this case, the same refilling process guarantees both block A and B will be reserved in L1 cache. So the behavior of LBF cache would be

$$A_mA_h^9B_m\ (A_h^{10}B_h)^9$$

and the miss rate is

$$M_{LBF} = 2/110 = 1.82\%$$

C. Conflict within loops

In this case, there are two references A and B within a single loop mapping to the same location in the cache. This type of memory access pattern may be $(AB)^{10}$. we can see that the behavior of direct-mapped cache would be

$$(A_mB_m)^{10}$$

and the miss rate is

$$M_{DM} = 20/20 = 100\%$$

The LBF cache have the same behavior, i.e. after initial A miss and B miss in the L1 cache, the next time A and B will be put in the DM cache and the buffer, therefore the behaviors are

$$A_mB_m\ (A_hB_h)^9$$

and the miss rates are

$$M_{LBF} = 2/20 = 10\%$$

From the examples above we can see that LBF cache may improve the hit rate of direct-mapped cache.

V. SIMULATION

A. Simulation Environment

sim-outorder in SimpleScalar toolset [10] was selected as our simulator, and mlcache [11] was used to replace the corresponding cache subprogram in sim-outorder.c. The simulation just considered to evaluate the performance of L1 Dcache, so the Icache, L2 cache and bus were supposed ideal. The parameters of simulator and memory system are listed in table I.

All the benchmarks for simulation were from SPEC95, in which only 129.compress were inputted with train data set, and others with test data set. All the benchmarks were run to the end.

B. Metrics

The performance measure best suited for the evaluation of the proposed caches is effective memory access time or total memory access time. If h is the hit rate in L1 cache, t is the access time of L1 cache, and t_{eff} is the effective access time, then

$$t_{eff} = h*t + (1- h) * misspenalty \qquad (1)$$

Here we assumed that the access time of L1 cache be one cycle, and the miss penalty be 18 cycles. As to our simulation, the miss rate is defined as

$$miss\ rate = L1\ Dcache\ miss\ number/\ L1Dcache\ reference\ number \qquad (2)$$

The speedup is

$$t_{eff} \text{ of the } target\ structure\ /\ t_{eff} \text{ of the } base\ structure \qquad (3)$$

Bus traffic that indicates the words L1 Dcache swaps with next level of memory hierarchy is another important metrics for processor. In this paper, Relative Bus Traffic is defined as below:

$$Relative\ Bus\ Traffic= Bus\ Traffic\ of\ base\ structure\ -\ Bus\ Traffic\ of\ target\ structure \qquad (4)$$

The Relative Bus Traffic is more, the speed of the processor is faster.

VI. EXPERIMENTAL RESULTS

The DM cache, 2-way associative cache, victim cache and assist cache are simulated to be compared to the LBF cache. All the augmented caches for compare are (8+1)KB, i.e. 8KB DM cache and 1KB buffer. The block size is 32B for all the caches.

A. Miss Rate

Table II shows miss rates for various L1 Dcache schemes after running 8 SPEC95 benchmarks. As a result, for all the benchmarks, the miss rates of the LBF cache are basically lower than those of assist cache except that of su2cor; except miss rates of gcc and su2cor, the miss rates of LBF cache are lower than those of victim cache. Among the three cache schemes, the average miss rates of LBF cache, victim cache and assist cache are respectively 5.34%, 5.38% and 5.48%.

TABLE I
PROCESSOR AND MEMORY CHARACTERISTICS

Fetch Mechanism	Fetches up to 4 instructions in program order per cycle
Branch Predictor	Bimodal predictor with 2084 entries
Issue Mechanism	Out-of-order issue of up to 4 operations per cycle, 16 entry re-order buffer, 8 entry load/store queue
Functional Units	4 integer ALUs, 4 FP ALUs, 1 integer MULT/DIV, 1 FP MULT/DIV
Data cache	Write-back, write-allocate, 32-byte lines, 4 read/write ports, non-blocking

TABLE II
MISS RATES OF SIX L1 DCACHE SCHEMES

	compress	gcc	li	ijpeg	perl	hydro2d	su2cor	swim
DM:8K	0.0673	0.0462	0.0231	0.0564	0.0597	0.1195	0.0891	0.4068
DM:16K	0.0557	0.0288	0.0178	0.0221	0.0373	0.1063	0.0796	0.1424
8K2W	0.0563	0.0300	0.0148	0.0448	0.0288	0.1112	0.0768	0.3904
Victim	0.0553	0.0259	0.0141	0.0111	0.0246	0.1094	0.0709	0.0473
Assist	0.0562	0.0287	0.0148	0.0118	0.0281	0.1095	0.0722	0.0472
LBF	0.0543	0.0265	0.0137	0.0104	0.0235	0.1078	0.0723	0.0463

TABLE III
RELATIVE BUS TRAFFIC OF FOUR L1 DCACHE SCHEMES (MILLION WORD)

	compress	gcc	li	ijpeg	perl	hydro2d	su2cor	swim
LBF	-3.3	-6.4	-3.0	0.5	2.07	-129.8	-90.9	177.9
Victim	-11.1	-64.4	-122.9	-6.4	-14.3	-319.8	-309.1	-449.5
Assist	-8.2	-30.1	-57.9	-0.3	-3.4	-274.9	-212.9	124.8
8K2W	0.1	-0.4	16.4	-2.1	2.4	-8.8	11.4	-144.0

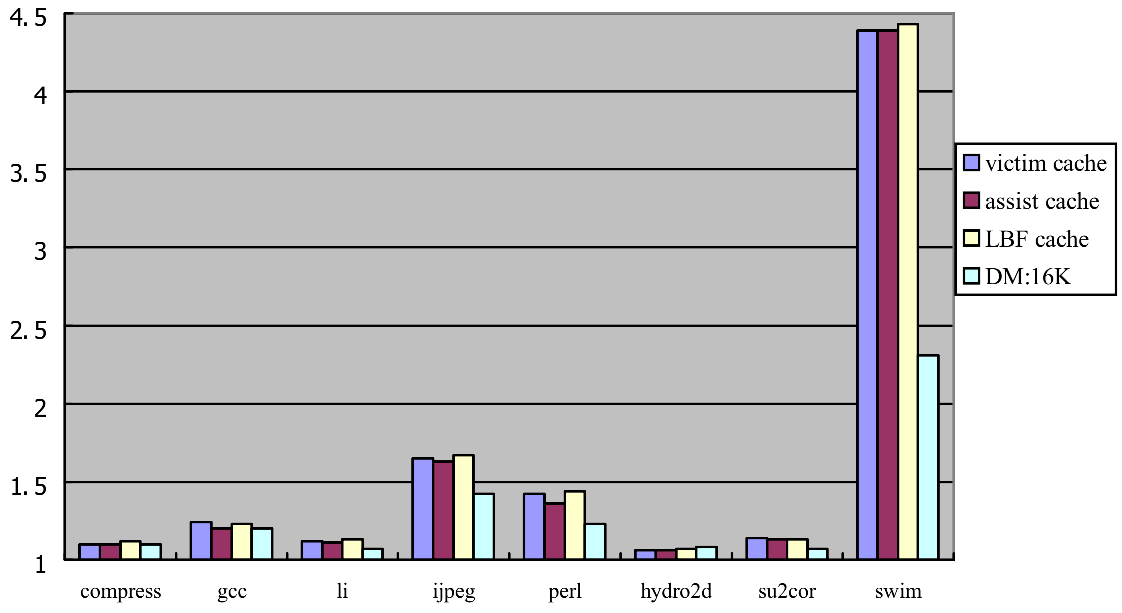

Fig. 2 Speedup of 4 cache schemes with 8KB DM cache as base structure.

Compared to 16KB DM cache and 8KB 2-way associative cache, the average miss rate of (8+1)KB LBF cache decreases about 26% and 53% respectively.

B. *Bus Traffic*

Table III shows the relative bus traffic result of four cache schemes as the target structures and the 16KB DM cache as the base structure. According to the table, the accesses to bus for LBF cache are the least among the four cache structures.

C. *Speedup*

Fig. 2 shows the speedup of four cache schemes that the 8KB DM Dcache is as the base structure. For most of the benchmarks, the speedup of LBF cache is the highest.

D. *Hardware Overhead*

Compared with victim cache and assist cache, the LBF cache is easy to be realized. The LBF cache only utilizes a one-direction path to transfer evicted blocks, but victim cache needs two paths. On judgment of data transfer, assist cache needs the help of software interpreter, but the LBF cache doesn't need.

VII. CONCLUSIONS

In this paper, a method called LRU-Like algorithm to filter blocks that may not be used recently on chip was presented. According to this method, a new cache structure named LBF cache was proposed. The result of simulation showed that miss rate of (8+1)KB LBF cache are apparently lower than those of 16KB direct-mapped cache and 8KB 2-way associative cache. Compared with the typical augmented cache (victim cache and assist cache), the miss rate of LBF cache was lower.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful comments. This research was supported by Open Fund of Key Laboratory of Electronics Engineering, College of Heilongjiang Province, (Heilongjiang University), P. R. China (DZZD20100033).

REFERENCES

[1] J. K. Peir, W. W. Hsu, and A. J. Smith, "Functional Implementation Techniques for CPU Cache Memories," *IEEE Transaction on Computers,* 48(2): pp. 100-110, 1999.

[2] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," *Proc. of MICRO-28 (Dec 1995), Piscataway, USA*, 1995, pp. 93-103.

[3] N. P. Jouppi, "Improving Direct mapping Cache Performance by the Addition of a Small Full Associative Cache and Prefetch Buffers," *In Proceedings of the 17th International Symposium on Computer Architecture, Seattle, USA,* 1990, pp.364-373.

[4] G. Kurpanek, G. Chan, K. Zheng, and et al, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *In Proceedings of IEEE International Computer Conference, San Francisco, USA,* 1994, pp. 375-382.

[5] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson, "Active Management of Data Caches by Exploiting Reuse Information," *IEEE Trans on Computers,* 1999, 48(11): pp. 1244-1258.

[6] J. A. Rivers and E. S. Davidson. "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *Proc. of the ICPP, vol. I,* August 1996, pp. 151-160.

[7] T. Johnson and W. W. Hwu. "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. of ISCA-24, Denver, USA*, 1997, pp. 315-326.

[8] S. McFarling, "Cache Replacement with Dynamic Exclusion," *Proc. of Annual Symposium on Computer Architecture. Gold Coast, Australia,* 1992, pp. 191-200.

[9] E. S. Tam, S. A. Vlaovic, G. S. Tyson, and E. S. Davidson, "Allocation By Conflict: A Simple, Effective Multilateral Cache Management Scheme," *Proceedings of 2001 International Conference on Computer Design (ICCD'01), Austin, Texas*, 2001, pp. 133-140.

[10] D. Burger and T. M. Austin, "Evaluating Future Processors: The SimpleScalar Tool Set," *Technical Report #1342, University of Wisconsin, Madison,* June 1997.

[11] E. S. Tam, J. A. Rivers, G. S. Tyson, and et al, "*mlcache*: A Flexible Multilateral Cache Simulator," *in Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Montreal, Canada,* 1998, pp. 19-26.