ADAPTIVE CACHE REPLACEMENT TECHNIQUE

Dinesh Dasarathan and Santhosh Kulandaiyan
School of Computer Science and Engineering
College of Engineering, Guindy
Anna University, Chennai-25.

Abstract:

In this poster we describe an adaptive cache replacement technique that aims at adapting itself to the program's cache access behavior by keeping track of the recentness of cache access and the frequency of cache access. The key to our technique is a parameter which is adjusted dynamically as the program proceeds in execution, so as to give frequency of access importance over the recentness of access or vice versa.

1. Introduction:

Memory access is one of the most time consuming jobs in computer systems. In the recent past, while CPU performance has increased at a rapid rate, memory access has not quite kept pace with it. Since in memory systems, cache plays a very important part, a lot of techniques have been proposed to improve the efficiency of the cache accesses. We propose a block replacement technique which adapts itself to the access pattern of the application.

Basically in caches, the most important problem lies in identifying which block to replace when we get a cache miss after all the blocks in the cache are filled up. Now in caches, instruction caches have a higher temporal locality than data caches, which means that if a block is accesed now, there is a high probability that it will be accessed in the near future also. Instruction caches also exhibit spatial locality according to which items whose addresses are near to one another tend to be referenced close together in time. This locality of reference also applies to data caches, but not as strongly as it does to instruction caches. Therefore we aim at achieving higher locality of reference in data caches.

2. Concept Involved:

Our poster tries to exploit two dimensions which are very important when we determine which block in the cache has to be replaced.

1. Frequency of access: The number of hits for the cache block

2. Recentness of access: The time since the last hit has occurred; this denotes the number of cache references that have taken place to other blocks since this cache block was last accessed.

Until the program begins to execute, we have little knowledge about the spatial and temporal locality that the program has got to offer. At a given instance of time, the program's data requirements may be such that it frequently accesses the block which has been accessed the most number of times or it frequently accesses the block that has been accessed the latest in time. So we can have an efficient block replacement technique if we have a parameter that adjusts itself to the requirements of the program - i.e.) by assigning more priority to one of the two dimensions, namely number of hits or time elapsed. This parameter is called α.

Using the parameter α we determine γ which is represented as

$$\gamma = (f1\ (\alpha) *_1 f2\ (\text{hits})) *_2 (g1(\alpha) *_3 g2\ (\text{time}))$$

where $*_1$, $*_2$ and $*_3$ are operations that vary from one implementation to the other. In this poster we have tried to project one such implementation. The functions f1, f2, g1, g2 are again implementation dependent. The purpose of α is to give weightage to either the hits or the time elapsed while we determine which block to eliminate. γ is the actual value based upon which we choose a block for replacement.

Later in this poster we have described a pair of simple algorithms to modify α dynamically, thereby adapting to the program's data access patterns.

3. Implementation:

For this implementation we maintain a data structure called ARU (adaptive replacement unit) that consists of many records. Each record consists of the following fields namely; block number (BN), number of hits (NOH) and time since last access (TLA). We shall discuss these parameters along with α in the following sections.

3.1 α:

α is stored in memory (or a register) and it occupies one byte of space. In binary representation if α is given the value of 0000 0001 then it gives a higher priority to Time elapsed than what it gives to the Number of hits. If α is given a value of 0000 0011 then it still gives a higher priority to time elapsed than what it does to the number of hits but the difference is somewhat lesser than in the previous case. A value of 0000 1111 indicates equal priority to both hits and time elapsed. This goes on till α reaches a value of 1111 1111 where time elapsed is given least priority and the number of hits is given the most priority.

3.2 Block Number:

Block Number denotes the tag number of the block if the cache is fully associative and denotes the set number and tag number if the cache is set associative.

3.3 Number of hits:

Number of hits of a block specifies the total number of times the block has been accessed from its most recent arrival into the cache. When a block is brought from main memory to cache, its corresponding NOH is set to the value 0000 0001. Whenever this block is accessed we left shift once (with 1 entering at the LSB) so that after 4 accesses the value of NOH becomes 0000 1111. NOH acts like a saturating counter in the sense that, once the value becomes 1111 1111(8 accesses) any further accesses will leave NOH unchanged.

3.4 Time since last access:

Time since last access denotes the number of cache accesses that have taken place to other blocks since this block was last accessed. Initially, when the block is brought from main memory to cache, the TLA field of this block is set to the value of all 1s, indicating that this is the most recently accessed block. Whenever an access to another block takes place we right shift once (with 0 entering at the MSB) so that 0001 1111 denotes that the 3 most recent accesses did not access this block. TLA too acts like a saturating counter, i.e., once TLA reaches the value 0000 0000(8 recent accesses did not access this block) even if the next access did not target this block, then too the value remains unchanged.

3.5 Determining the functions:

In this poster we have presented one implementation for the calculation of γ given by the formula

γ = (f1 (α)$*_1$ f2 (hits)) $*_2$ (g1 (α)$*_3$ g2 (time)).

A lower value of γ denotes that the corresponding block is more suitable to be replaced. We took into consideration 2 factors in determining the functions:

1) The minimality of hardware required to implement the functions.

2) The accurateness of γ in determining the block to be replaced.

Based on the above we deduced the following functions:

f1 (α) = α

f2 (hits) = NOH (for e.g. if hits=4 then f2 (hits) =0000 1111).This can actually be implemented in hardware using a Left Shift Unit with 1s entering from the right side.

g1 (α) = The one's complement of α normalized to the right. (For e.g. if α =0000 0011 then 1s comp α =1111 1100 and normalized value=0011 1111). This can be implemented in hardware using a Not Unit and a Shift Unit.

g2 (time) = TLA (for e.g. if time=3 then g2 (time) =0001 1111).This can also be I implemented using a Right Shift Unit with 0s entering from the left side.

3.6 Determining the Operators:

The operators $*_1$, $*_2$ and $*_3$ were deduced as follows:

$*_1$, $*_3$ = Bitwise And

$*_2$ = Taking bitwise 1's complement of both the operands from the LSB until the first 0 is encountered for which a 1 is substituted. Leave the other zeroes intact. After this the Bitwise OR is performed for both operands.

4 Example:

We consider 2 blocks and decide on which block to replace.

Block1 has the following values .
NOH = 0011 1111 (6 hits)
TLA = 0000 0111(5 slots)

Block2 has the following values.
NOH = 0000 0011(2 hits)
TLA = 0111 1111(1 slot)

Slots here denote the number of cache references since this block was last accessed. Now let us consider 2 values of α to determine which block to replace.

Case1: α = 0001 1111

Block1

f1 (α) = 0001 1111	f2 (hits) = 0011 1111
g1 (α) = 0000 0111	g2 (time) = 0000 0111

Performing the operations as described above, we get the following result for γ.

f1 (α)$*_1$ f2 (hits) = 0001 1111

g1 (α)$*_3$ g2 (time) = 0000 0111

The intermediate step in $*_2$ is shown:

0001 1111 is converted to 0010 0000
0000 0111 is converted to 0000 1000

The final result is obtained by bitwise ORing the two values and we obtain

γ = 0010 1000

Block2

f1 (α) = 0001 1111	f2 (hits) = 0000 0011
g1 (α) = 0000 0111	g2 (time) = 0111 1111

Performing the operations as described above, we get the following result for γ.

f1 (α)$*_1$ f2 (hits) = 0000 0011
g1 (α)$*_3$ g2 (time) = 0000 0111

The intermediate step in $*_2$ is shown:

0000 0011 is converted to 0000 0100
0000 0111 is converted to 0000 1000

The final result is obtained by bitwise ORing the two values and we obtain

γ = 0000 1100

Since the value of γ is lower for the 2nd block, it is selected for replacement. This was because, we assigned the value of α so that it gave priority to the number of hits rather than the time slots. This resulted in block1 to have a greater value of γ and thereby preventing itself from being replaced.

Case2: α = 0000 0011

Block1

f1 (α) = 0000 0011	f2 (hits) = 0011 1111
g1 (α) = 0011 1111	g2 (time) = 0000 0111

Performing the operations as described above, we get the following result for γ.

f1 (α)$*_1$ f2 (hits) = 0000 0011
g1 (α)$*_3$ g2 (time) = 0000 0111

The intermediate step in $*_2$ is shown:

0000 0011 is converted to 0000 0100
0000 0111 is converted to 0000 1000

The final result is obtained by bitwise ORing the two values and we obtain

γ = 0000 1100

Block2

f1 (α) = 0000 0011	f2 (hits) = 0000 0011
g1 (α) = 0011 1111	g2 (time) = 0111 1111

Performing the operations as described above, we get the following result for γ.

f1 (α)$*_1$ f2 (hits) = 0000 0011
g1 (α)$*_3$ g2 (time) = 0011 1111

The intermediate step in $*_2$ are shown:

0000 0011 is converted to 0000 0100
0011 1111 is converted to 0100 0000

The final result is obtained by bitwise ORing the two values and we obtain

γ = 0100 0100

Since the value of γ is lower for the 1st block, it is selected for replacement. This was because, we assigned the value of α so that it gave priority to the time since last accessed rather than the number of hits. This resulted in block2 to have a greater value of γ and thereby preventing itself from being replaced.

5 Adaptive Algorithms:

The value of α determines which has a higher priority, the number of hits or the time since last access, in choosing a block for replacement. For a particular program, if the value of α that exists doesn't work well (i.e. blocks in the cache still keep missing) then it means that the value of α has to shift so that it gives priority the other way around (i.e. if initially α gives priority to number of hits and cache misses still occur then the value of α has to be shifted so that it now gives priority to the time elapsed).

We propose 2 algorithms by which we can vary α so that the cache replacement policy can adapt to the program.

5.1 Constant Hopping Algorithm: (Hop Step=1)

1. Start from a value of α, say equal priority to both time and number of hits and note the number of misses/unit time (MT). (Note that MT is obtained after every 'n' time slots).

2. Increase the value of α so that it moves towards giving the number of hits a higher priority. We also maintain a single bit called direction bit (DB) which states the direction of motion of α (1 indicates that α was previously incremented, 0 indicates that α was previously decremented).

3. Compare the new value of MT with the old value. If it is lesser, hop along the same direction. i.e. if DB=1,

increment α and if DB=0, decrement α.(Incrementing α value is to move alpha towards giving the number of hits a higher priority and decrementing α value is to move α towards giving the time since last access higher priority).

4. If the new value of MT is higher, then the direction in which α hopped previously was not efficient. Therefore, depending on the DB value, hop in the reverse direction. i.e. if DB=1, then decrement α and if DB=0, increment α. Reverse the value of DB (i.e. if DB were 1 it becomes 0 and if it were 0 it becomes 1).

5. After n time slots calculate the value of MT and Go to step3.

5.2 Variable Hopping Algorithm: (Hop Step is variable)

1. Start from a value of α, say equal priority to both time and number of hits and note down the number of misses/unit time (MT).

2. Increase the value of α so that it moves towards giving the number of hits a higher priority. We also maintain a single bit called direction bit (DB) which states the direction of motion of α (1 indicates that α was previously incremented, 0 indicates that α was previously decremented).

3. Compare the new value of MT with the old value. If it is lesser, hop along the same direction by 2 steps. i.e. if DB=1, increment α by 2 and if DB=0, decrement α by 2.

4. If the new value of MT is higher, then the direction in which α hopped previously was not efficient. Therefore, depending on the DB value, hop in the reverse direction by 1 step. i.e. if DB=1, then decrement α by 1and if DB=0, increment α by 1 . Reverse the value of DB (i.e. if DB were 1 it becomes 0 and if it were 0 it becomes 1).

5. After n time slots calculate the value of MT and Go to step3.

5.3 Example for Adaptive Hopping Algorithms:

α	4	5	6	5	4	3	4	5
DB	1	1	0	0	0	1	1	--
MT	20	18	22	21	20	23	21	

Table 1: Constant Hopping Algorithm
DB = Direction Bit MT= misses/unit time

In Table 1, we start with an α value of 4 .Initially we increase α to 5. Since it shows increasing performance (since MT decreases from 20 to 18) α further hops along the same direction to 6. At this point MT increases from 18 to 22.So we decrease the value of α by one and reverse DB (i.e., make it 0).

α	4	5	7	6	4	2	3	5
DB	1	1	0	0	0	1	1	--
MT	20	19	23	22	20	24	23	--

Table 2: Variable Hopping Algorithm
DB = Direction Bit MT= misses/unit time

In Table 2, we start with a α value of 4. (The main difference from the previous algorithm is that we use variable hops for α - 1, 2 instead of a constant hop, namely 1).Initially we increase α to 5. Since it shows increase in performance we now increase α by a variable hop, namely 2, to 7.This results in decrease of performance and hence we invert DB and decrease α by 1.The process repeats as shown.

6 Future Work:

We can enhance the logic put forth by this poster by 2 means:

During each computation, maintain a lower cutoff or threshold for the value of γ using the previously obtained value for it. At the current computation if a block crosses this cutoff then that block can be selected for replacement without attempting to calculate the γ values of other blocks. This reduces computational time involved in selecting the block for replacement. Further a Buffer is maintained to store the present block contents. Even as data is given from this buffer we can simultaneously determine which block has to be replaced. The idea here is that the program shouldn't stall while we determine the block for replacement.

7 Conclusion:

In this poster we have proposed a technique to dynamically alter the replacement technique allowing it to range from the Least Recently Used replacement policy to the Least Frequently Used replacement policy depending on the program's access behavior. We hope that this technique could provide a good combination of both the above policies in trying to exploit cache behavior, thereby reducing the number of overall misses that can occur.

References:

1) David A Patterson and John L Hennessy "Computer Architecture A Quantitative Approach – 2/e" Morgan Kaufmann Publishers 1996.

2) Hill, Mark D. "Aspects of Cache Memory and Instruction Buffer Performance." Ph.D. Th., University of California, Berkeley, 1987.

3) Jouppi, Norman P. "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers"

4) Agarwal A. "Analysis of Cache Performances for Operating Systems and MultiProgramming" Ph.D. Th., Stanford University, 1987 ,Tech Report No. CSL-TR-87-332(May)

5) Vijayalakshmi Srinivasan "Improving Performance of an L1 Cache with an Associated Buffer" Electrical Engineering , University of Michigan , Feb 12 ,1998.