H-NMvu: A Low Area, High Performance Cache Replacement Policy for Embedded Processors

Sourav Roy
Freescale Semiconductors, India Design Center
sourav.roy@freescale.com

Abstract

We propose a low area, high performance cache replacement policy for embedded processors called Hierarchical Non-Most-Recently-Used (H-NMvu). The H-NMvu is a parameterizable policy where we can tradeoff performance with area. We extended the Dinero cache simulator \[1\] with the H-NMvu policy and performed architectural exploration with a set of cellular and multimedia benchmarks. On a 16 way cache, a two level H-NMvu policy where the first and second levels have 8 and 2 branches respectively, performs as good as the Pseudo-LRU (PLRU) policy with storage area saving of 27%. Compared to true LRU, H-NMvu on a 16 way cache saves huge amount of area (82%) with marginal increase of cache misses (3%). Similar result was also noticed on other cache like structures like branch target buffers. Therefore the two level H-NMvu cache replacement policy (with associativity/2 and 2 branches on the two levels) is a very attractive option for caches on embedded processors with associativities greater than 4.

1 Introduction

Older generation embedded processors would consist of a large internal SRAM and an external memory interface connecting to a DRAM outside the chip. Most applications would fit in the internal SRAM comfortably. Caches were not preferred due to real-time variability in memory access latencies. As embedded systems became more complex, applications became very large and caches became inevitable. In the last decade, cache architectures have penetrated the world of embedded processors. At present embedded processors have cache architectures as complex as general purpose processors. The architecture of embedded processor caches, especially on mobile devices, is complicated by the fact that all three metrics of performance, power and area have to be satisfied simultaneously within the given constraints. This is different from general purpose processors, where performance is the clear priority, followed by power (in recent times). Today in the multi-core era, there is a trend to move to embedded systems with multiple processors, each of low area and frequency connected with high throughput interconnect fabric. Hence there is a need to reduce the complexity, area and power of individual processor components like caches, with minimal performance reduction.

Cache replacement policies determine which line to replace when there is a miss. In a set-associative cache, a miss occurs when the accessed set is full. There are three replacement policies that are widely used: LRU (least recently used), FIFO (first in first out) and RAND (random). Among these LRU is widely accepted to be the most preferred cache replacement policy. LRU most closely corresponds to the concept of temporal locality. Typically, on general purpose benchmarks FIFO and RAND amplifies the LRU miss rate by 12% and 20% respectively \[2\] \[3\]. More recently there has been some work on cache performance evaluation on SPEC2K benchmarks \[4\] \[5\]. The LRU policy is implemented as a circular stack as shown in fig 1. During a hit the most recently accessed line is placed at the top of stack. During a miss, the line at the bottom of the stack is replaced and the newly inserted line is placed at the stack top. Though LRU is the preferred policy from a performance angle, it is also the most complicated to implement. Each set comes with its own LRU history bits and they are updated with each memory reference to that set. For a \(m\)-way set-associative memory \((m - 1) * \log_2 m\) bits are required per directory entry of a set \[2\]. This is a big overhead in embedded processors where area is extremely important. To overcome this problem, approximations to LRU which occupy much lesser area have been suggested. The most prominent and widely used among them is Pseudo LRU (PLRU). The PLRU is extensively used in the PowerPC series.
of microprocessors [6]. The PLRU is implemented a binary tree, whose leaf nodes are the lines in the set, as shown in fig. 1. During a hit, the path to the referenced leaf node is traversed and on the way the node values are flipped to point to the other branch. During a miss the tree is traversed according to the node values to find the line for replacement. For a \( m \)-way set-associative cache, PLRU requires \((m - 1)\) bits per directory entry of a set. The PLRU while occupying much lesser area has been reported to perform as well as the LRU policy on general purpose benchmarks [4]. Most embedded processors use LRU or PLRU as their cache replacement policies. Processors using LRU replacement policy typically avoid associativities higher than 4 due to extra area overhead of LRU history bits. PLRU can be also used for higher associativities like 8 and 16, and in other highly or fully associative cache like structures like branch-target-buffers (BTB).

![LRU and PLRU cache replacement policies](image)

**Figure 1. LRU and PLRU cache replacement policies**

Cache replacement policies have been explored thoroughly over the last two decades. For embedded processors, the known cache replacement policies are efficient enough to provide high performance. However in today’s multi-core era there is an acute requirement to reduce the complexity and area of caches, with minimal performance impact. In this work we have proposed a new cache policy H-NMRU to reduce area without sacrificing non-negligible performance over the LRU policy. The rest of this paper is organized as follows. Section 2 introduces the H-NMRU cache replacement policy and its computational and space complexity. Section 3 describes the results of using H-NMRU cache replacement policy on a set of cellular and multimedia benchmarks, vis-a-vis other popular cache replacement policies. Section 4 gives further results of applying H-NMRU replacement policy in BTBs on a UMTS voice call trace. Section 5 concludes with the advantage of using H-NMRU over LRU and PLRU policies.

2 The H-NMRU Cache Replacement Policy

The H-NMRU cache replacement policy can be explained using a multi-way tree, where the leaf nodes represent the lines in the set. Each intermediate node stores the value of its most-recently-used (MRU) child. Let us take an example of a 16-way set-associative cache as shown in fig. 2. In this example, the root has 4 branches. Each child in turn has 2 branches, followed by another 2 branches in the next level. This multi-way tree has 3 levels. It is represented using H-NMRU(4,2,2) where the numbers represent the number of branches at each level of the tree. The nodes of the tree store the value of the most recently traversed branch.

During a cache hit, the H-NMRU tree is traversed to reach the accessed line at the leaf node. On the way, the value of the nodes are updated to point to the path of traversal. In this way, the most recently used branches are stored at each node of the tree. On a cache miss, the tree is traversed selecting a random value different from the MRU value stored in the node. From each level a non-MRU path is selected. In this way, this algorithm points to a leaf node which has not been accessed in recent times. For example in fig. 2, the most recently used way is 11, which can be determined by a simple tree traversal. The dotted lines represent the current set of branches that can be traversed randomly for way replacement. Hence the H-NMRU policy selects one from way 3, way 5, or way 14 for replacement. The random replacement is done in hardware with simple linear feedback shift registers (LFSR). The random selection is done only for the accessed set in the cache. Hence it occupies negligible hardware area.

Since this is a parameterizable policy, the key is to find the correct number of levels and the number of branches for each level. Let us define a generic H-NMRU policy for a cache with associativity \( N \):

\[
\text{Cache Policy} : H\text{-NMRU}(i_0, i_1, \ldots, i_{n-1})
\]

\[
\text{with } \prod_{j=0}^{n-1} i_j = N
\]

The PLRU can then be defined as a special case of H-NMRU where each of the \( i_j = 2 \) and \( n = \log_2 N \).
Similarly NMU is also a special case of H-NMRU.

\[
\text{PLRU} = \text{H-NMRU}(2, 2, \ldots, 2)
\]

\[
\text{NMU} = \text{H-NMRU}(N)
\]

The computational complexity of H-NMRU is directly related to the number of levels in the multi-way tree i.e., \( n \). Let us now find the space complexity of the H-NMRU policy. A node that has \( m \) branches can be represented using \( \log_2 m \) bits. Hence the number of history bits per set required for the H-NMRU policy is given by,

\[
\text{No. of bits per set} = \sum_{j=0}^{n-1} \left( \prod_{k=0}^{j-1} i_k \right) \times \log_2(i_j)
\]

The randomizer hardware needs to select the non-MRU branch at every level. Hence it can select one out of \( (i_j - 1) \) branches. For any level with greater than 2 branches, the LFSR needs to have \( \log_2(i_j) \) bits. In a simple implementation of the randomizer hardware we create a pseudo-random sequence of \( (i_j - 1) \) states excluding the all-zero state. When this state is exclusive-OR ed with the MRU state value, we get the random non-MRU state.

Next we explore the design space of the H-NMRU policy by evaluating the performance for a 16 way, 16 KB data cache with different parameters. Fig. 3 plots the average misses for different H-NMRU configurations. The misses are normalized with respect to that of PLRU or H-NMRU(2,2,2,2). The misses are measured by simulating a large number of cellular and multimedia benchmarks on the Dinero cache simulator which has been extended to support the configurable H-NMRU replacement policy. As seen from the plot, three other H-NMRU configurations have very similar performance (within 2% degradation) to that of PLRU but with much lower area requirement. The most attractive is the H-NMRU(8,2) configuration which uses only 11 bits per set entry instead of 15 as in PLRU, yet deliver a performance with only 2% degradation. As we reduce the number of bits per set, the miss rate of the H-NMRU policy increases. For the same number of bits, H-NMRU(8,2) performs better than H-NMRU(2,2,4).

3 Performance and Area of H-NMRU policy

In this section we will discuss the performance of the H-NMRU cache replacement policy. We extended the Dinero cache simulator to support the H-NMRU cache replacement policy. Then we applied several real-life traces on it to measure the cache performance. The benchmarks included in the paper are cellular and multimedia applications - Adaptive MultiRate (AMR) encoder, G.729 speech codec, MPEG4 decoder, Session Initiation Protocol (SIP), protocol stack of 3G voice call with HSDPA, and MP3 decoder. All the traces consist of more than 100 million instruction and data accesses. This is to ensure that we do not get misleading results from small traces. Level 1 instruction and data caches with 16 ways and 32B line size have been simulated using Dinero. Among the entire set of H-NMRU configurations, we have chosen H-NMRU(8,2) and H-NMRU(4,2,2). The former is the most attractive option from an area vs. performance trade off, whereas the latter is the best performance obtained from a non-PLRU H-NMRU policy. The results of individual applications are shown in fig. 4 and the av-
Figure 3. Design Space Exploration of H-NMRU policy on 16KB Data Cache

Figure 4. Comparison of various cache replacement policies
erage over all the applications is shown in fig. 5. We conclude that the miss rate for H-NMRU(8,2) is only 3% higher than LRU and 1.5-2.0% higher than PLRU. Other cache metrics like traffic depend on the miss rate, and so we have only analyzed the cache misses. Though we have considered only level 1 caches here, similar relative miss rates can be observed on level 2 caches as well. This directly follows from the hypothesis that if the level 1 cache is small compared to the level 2 cache, the performance of the latter can be computed by assuming that no level 1 cache is present [7].

![Figure 5. Average Performance](image)

**Figure 5. Average Performance**

The H-NMRU(N/2,2) option is the best option in terms of area savings with very marginal performance loss. For an associativity N, PLRU bits per set is given by \( (N - 1) \) whereas H-NMRU(N/2,2) bits per set is given by \( \log_2(N/2) + N/2 \). As N increases, the area savings of H-NMRU is higher. For large N, \( N/2 \gg \log_2(N/2) \), hence H-NMRU bits \( \approx N/2 \) and PLRU bits \( \approx N \). Therefore theoretically H-NMRU(N/2,2) takes half the storage bits of PLRU for large associativities. For typical associativities of 16 and 32, H-NMRU reduces storage requirement by 27% and 35% over PLRU. Compared to LRU, the H-NMRU savings are 82% and 87% respectively. Fig. 6 compares PLRU and H-NMRU(N/2,2) storage bits across cache sizes and associativities. It is important to note that it is not only the storage bits that are reduced but also the associated logic around the storage elements.

From an absolute area perspective, L1 caches typically implement replacement storage bits using registers to avoid frequency reduction. Approximate savings of H-NMRU(N/2,2) over PLRU for 16 way, 16KB and 32KB L1 caches will be 20,000 and 50,000 \( um^2 \) in 65nm technology. For a typical 256KB L2 cache, the approximate saving of H-NMRU over PLRU is 2.5 \( mm^2 \) in 65nm technology. Sometimes L2 cache replacement bits are stored in SRAMs as they are not timing critical. Then the approximate area savings would reduce to 0.2 \( mm^2 \). Hence H-NMRU is practically more suited for higher associativities and also for large caches as found in L2 cache. It is to be noted here that real silicon area depends on the timing of the particular chip and also on the technology.

![Figure 6. Area Comparison of PLRU and H-NMRU(N/2,2)](image)

**Figure 6. Area Comparison of PLRU and H-NMRU(N/2,2)**

4 H-NMRU replacement for BTB

In this section we explore the performance of the H-NMRU replacement policy for another processor component called Branch Target Buffer (BTB). The BTB is a cache like structure that is used to reduce the performance penalty of branches by predicting the path of the branch. Each entry in the BTB typically contains the branch address, the target address and the prediction bits. When the processor fetches the instruction, its address is compared with the entries in the BTB. If there is a match with any entry, the prediction bits are used to predict if the branch will be taken or not. In case the prediction is branch taken, then the target address is used as the next instruction fetch address. This saves the branch penalty of the pipeline. There are two aspects of the BTB that are important for effective operation viz., BTB prediction accuracy and the BTB hit ratio. We will deal with the latter here in connection with replacement algorithm. If there is a BTB miss, it is advantageous to predict that program flow should go in line. Once there is a BTB miss and the program takes a branch, the BTB needs to be updated with the new branch entry. In this case an entry needs to be evicted to make space for the new entry. Typically, the LRU policy is used as the replacement policy. In this section we explore the performance of both PLRU and H-NMRU policies for BTB.

Several embedded processors with deeper pipelines
have started using BTB for branch performance improvement. Popular examples are ARM11 [8] and StarCore 3400 [9]. The challenge in embedded processors is to improve branch performance with limited area overhead. Typically modest size BTBs in embedded processors have high associativity, even fully associative or only very few (2-4) sets. This increases the area of LRU bits. To reduce the area overhead of history bits for cache replacement, the H-NMRU replacement policy can be used. In fig. 7 we plot the BTB misses for LRU, PLRU and H-NMRU(size/2,2) policy. The BTB explored here has full associativity and the application is a 3GPP UMTS voice call application. Few billion instructions have been simulated to study the BTB miss rates. As we see the H-NMRU policy performs very close (within 2-3%) to the LRU and PLRU policies while saving substantial area. For example the PLRU policy requires 31, 63, 127, 255 bits whereas the H-NMRU(size/2,2) would require 20, 37, 70, 135 bits respectively. Since the H-NMRU policy reduces storage bits and has only two levels in the tree, the timing also considerably improves.

Figure 7. BTB Misses for UMTS voice call

5 Conclusion

In this paper we presented a new cache replacement policy called H-NMRU that saves substantial area while providing similar performance to LRU and PLRU. The H-NMRU is a parameterizable policy and among them an attractive option is H-NMRU(way/2,2). On a 16 way cache, it saves 27% area over PLRU with minimal loss of performance(1.5%). The absolute area saving is encouraging for highly associative and level 2 caches. This policy can also be used for BTB entry replacement, where typically associativity is high. Timing also improves considerably due to lower number of bits and only two levels in the tree. Hence H-NMRU(N/2,2) policy can help to reduce complexity, area and standby power of cache structures in present-day multi-core mobile platforms.

Here we have explored H-NMRU cache replacement policy in relation with embedded processors as area is a high priority for them. However we would also like to encourage architects in the general purpose computing domain to explore H-NMRU.

6 Acknowledgments

We would like to thank Brace Randall and Itay Peled at Freescale Semiconductors for the application traces that we used for cache performance evaluation.

References


