University of Jordan
Computer Engineering Department

A Study on Cache Replacement Policies

CPE 731: Advanced Computer Architecture
Dr. Gheith Abandah

Asma Abdelkarim

December 2, 2009

Abstract

A miss in the last level caches causes hundreds of stall cycles due to the need for a memory access. Therefore, last level caches are designed to reduce the possibility for cache misses. Since last level caches lack temporal locality, the Least Recently Used policy produces bad performance for them. Hence, many replacement policies were proposed to improve the miss rate for last level caches while maintaining low hardware overhead and minimum design changes. This report represents the recent proposed replacement policies and the details of the required cache design changes. Then, the report discusses recent proposed extensions to the replacement policies in order to manage shared caches in Chip Multiprocessors. However, the lack of unified simulation for all proposed policies prevents accurate comparison of their performance.

1. Introduction

Caches play a vital role in reducing the delay of memory accesses by providing a temporary fast-access storage unit for the data accessed by the processor. On a cache miss, data is fetched from the memory and placed in its corresponding location in the cache. For set-associative caches, a replacement policy is required to decide where in the set the fetched cache line should be placed. An efficient cache replacement policy can significantly reduce the cache miss rate, thus, reducing the possibility of a penalty of hundreds of cycles due to memory accesses.

The Least Recently Used (LRU) Policy has been the standard replacement policy used for caches (L1, L2 and L3 caches) [5] [6]. The LRU policy works sufficiently with L1 caches since they benefit the most from temporal locality because of their direct interface with the processor. In addition the LRU policy is perfect for the simplicity required in the

design of L1 caches [8]. However, references to L2 caches (and last level caches in general) lack temporal locality, since lines that are referenced recently are kept in the L1 cache and they will not be requested from the L2 cache again [5]. LRU will cause these lines (called dead lines by [10]) to be kept in the cache until they travel all their way from the MRU (Most Recently Used) position to the LRU used position, and only then they will be evicted [6] [10]. In the worst case these lines may never be reused during their residence in the cache, these lines are called zero-reused lines by [5 and 6].

Moreover, for workloads that have working sets that are larger than the L2 cache, LRU causes thrashing. These workloads are called memory intensive workloads. When the LRU policy is used for such workloads, lines that are inserted in the cache will be referenced in the future but due to the capacity misses, they will be replaced by new lines before being re-referenced. Thus, LRU causes a 0 hit rate for these workloads since all of the cache lines will be zero-reused lines. [5] [6]

In addition to what stated so far, the deign goal in L2 and last level caches is to minimize the possibility of long accesses to the memory that would be caused by a cache miss. Besides, last level caches are of larger sizes and higher associativity [8]. All these factors made recent studies attempt to find optimized replacement policies that can work better than the LRU poor performance with last level caches [3, 5, 6, 7, 8, 9, and 11]. The fact that replacement policies can be optimized to gain more control over the cache, motivated the researchers to extend their work to consider the problem of shared cache management in chip multiprocessors (CMPs) [2, 4, 10].

For all the proposed replacement policies, there are multiple design issues that must be taken in consideration. First, the additional required hardware should be as minimal as possible. Second, the policy should work efficiently with different types of workloads. It must not perform worse than LRU for a wide range of workloads. Finally, it should produce minimum changes to the existing cache design. [6] [9]

2. Literature Survey

Belady in [1] proposed the optimal replacement policy (OPT) which provided an upper limit on the hit rate that can be ever achieved. OPT states that "the optimal candidate for replacement is the one that is accessed farthest in the future". This policy cannot be implemented since it requires knowing which lines will be accessed in the future and when they are going to be accessed.

The traditional replacement policies well known in the literature includes: LRU, LFU, random replacement and FIFO, with the LRU policy being the standard for today's on-chip caches. Improvements on replacement policies have been static for a long time for two main reasons: the suggested solutions required significant additional hardware, and the assumption that LRU works sufficiently, which is the case for L1 caches only. [9]

The fact that the LRU policy has bad performance when used with last level caches, led to the emergence of a lot of studies that attempt other optimized solutions that may achieve a performance as close as possible to OPT while maintaining low hardware overhead. Qureshi et al. [5] [6] proposed anti-thrashing policies that performs well for memory intensive workloads and proposed a dynamic policy that adaptively chooses either LRU or the proposed anti-thrashing policy depending on the workload.

Other policies that combine both LRU and LFU were proposed by Subramanian et al. in [9] and Megiddo and Modha in [3]. In [7] Qureshi et al. proposed a policy that takes its replacement decision in order to exploit Memory Level Parallelism (MLP), such that lines that may cause misses with high MLP cost are least probably evicted. Rajan and Govindarajan [8] proposed a policy that mimics Belady's optimal policy [1] by producing the shepherd cache. Zebchuk et al. [11] modified the shepherd cache so that it has lower hardware overhead.

Recent studies also include using replacement policies to manage shared caches in CMPs. Most of the proposed policies are optimizations to the proposed policies for uniprocessor. In Qureshi et al. modified their dynamic policy so that it chooses the replacement policy for each core individually. Kron et al. [2] proposed the notion of promotion policies and combines it with the dynamic policy proposed in [6] to form a policy that combines both dynamic replacement and dynamic promotion. In [10] Xie and Loh proposed a policy that implicitly partition the shared cache among cores using the insertion and promotion policies.

3. Recent replacement policies for uniprocessors

3.1 MLP-Aware replacement policies [7]

Memory Level Parallelism (MLP) refers to the process of serving multiple memory operations due to cache misses in parallel. MLP reduces the number of stall cycles due to memory accesses. Multiple cache misses may occur concurrently because modern processors are speculative superscalar processors with pre-fetching capabilities. [7]

In [7] Qureshi et al. proposed the idea of optimizing replacement policies so that they exploit memory level parallelism, these are called MLP-aware replacement policies.

Instead of reducing the number of misses, MLP-aware policies aim to reduce the number of memory stalls by avoiding misses that occur in isolation. Such misses do not exploit MLP since they will be served individually. When a replacement decision is to be made, lines that may cause isolated misses will be replaced with the least probability.

MLP-aware policies assign a cost for each line that reflects the number of parallel misses that are concurrent with that line's miss, called MLP cost. Since this value cannot be known for any line in advance it is predicted based on the current MLP cost of the line. For any line, the difference between the MLP costs of two successive misses is computed, called a delta. Delta is computed for multiple successive misses, in order to be used to predict the next MLP cost for the line. Small delta values indicate less varying MLP costs and more accurate prediction. MLP cost = 1/N, where N is the number of concurrent misses. Hence, the line with the lower MLP cost is the one that will be replaced.

In order to compute the current MLP cost for each line, each time a miss occurs on a cache line it will be assigned an entry in a Miss Status Holding Register (MSHR). The MLP cost will be computed depending on the number of concurrent misses in the MSHR. Figure 1 shows the architecture of the MLP-aware policy.

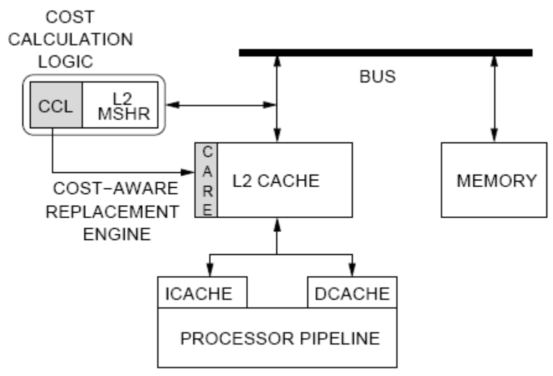


Figure 1: The architecture of the MLP-aware policy [7]

Qureshi et al. [7] extended the MLP-aware policy to include recency by producing the linear (LIN) policy. Blocks are given recency values, with the MRU having the highest value and the least recently used having the lowest. The block to be replaced is the one with the minimum {recency_value + MLP_cost}. They found that LIN improved the performance of some of the workloads. However, for workloads with high delta values, LIN produced worse performance compared with the LRU policy. For that, they proposed a hybrid replacement policy that can choose either LRU or LIN depending on which performs better. This required a mechanism to observe the performance of each policy according to the cache accesses.

Qureshi et al. [7] called this mechanism the Tournament Selection (TSEL). In this mechanism a counter called the saturating counter (SCTR) is updated on each miss of the two policies. For a miss in LIN, SCTR is decremented. For a miss in LRU, SCTR is incremented. The most significant bit in SCTR is then used to select the policy, if 1 LIN will be used, if 0 LRU will be used.

In order to count the misses for each policy on the cache accesses, two extra tag directories called the Auxiliary Tag Directories (ATD) are added, one for each policy. These are called ATD-LIN and ATD-LRU. The misses from the two policies are used to update SCTR. The auxiliary directories must have the same associativity as the cache, which is a significant additional hardware overhead. Qureshi et al. [7] proposed a new mechanism which reduces the amount of tag entries required, called Dynamic Set Sampling. This mechanism proved that "the behavior of the cache can be approximated with high probability by sampling few sets in the cache" [7]. This mechanism reduces the additional overhead significantly. They proposed another mechanism, which further

reduces the additional hardware overhead, called Sampling Based Adaptive Replacement (SBAR). In this mechanism, one of the auxiliary directories is eliminated and implemented as part of the main directory itself. This is done by dedicating some of the sets in the main directory for one of the policies so that it counts the misses resulting from that policy. To specify the allocated dedicated sets, they divided the cache into constituencies. The number of the dedicated set in each constituency is the same as the number of the constituency, for example set 0 is the dedicated set in constituency 0 and set 1 is the dedicated set in constituency 1 and so on.

3.2 LIP, BIP and DIP [5] [6]

In [6] Qureshi et al. proposed three new replacement policies: LIP, BIP and DIP. They divided the replacement process into two parts: victim selection and insertion. The victim selection part decides which line should be evicted from the cache, while the insertion part decides where in the recency stack the new line should be placed. The recency stack is the way the cache keeps track of the recency of each line. For example, the LRU policy always selects the line in the LRU position as the victim while inserts the new line in the MRU position.

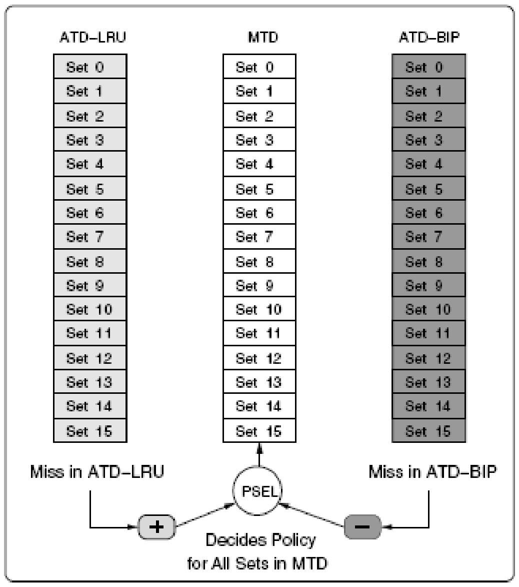
In their study, they tried to solve the problem of cache thrashing due to memory intensive workloads. These workloads, having working sets that are larger than the cache size, will cause 100% miss rate under the LRU policy. The first policy, LRU Insertion Policy (LIP), tries to keep a fraction of these memory intensive working sets in the cache, so that they would contribute to hits when they are re-referenced. This policy selects the victim from the LRU position as in LRU. However, it inserts the lines always in the LRU position. Lines are promoted from the LRU position to the MRU position only if they are

re-referenced. This policy achieves high hit rates for workloads with cyclic working sets. But it cannot adapt to changes in the working set since new lines will be inserted in the LRU position with older lines staying in the MRU position without being evicted.

The second policy, Bimodal Insertion Policy (BIP), modifies LIP to allow adapting with changes in the working set. This is done by allowing lines to be inserted in the MRU position with a very low probability Є. A 5-bit counter that is incremented on each miss is used to achieve that. Each time the counter reaches zero, the line fetched on the corresponding miss will be inserted in the MRU position.

BIP works well for memory intensive workloads. However, some workloads are LRU-friendly; which means they have higher hit rates when working under LRU while providing poor hit rates when working under BIP. For that, Qureshi et al. [6] proposed their third policy which is an adaptive policy, the Dynamic Insertion Policy (DIP). This policy dynamically chooses either LRU or BIP according to which one causes fewer misses. A counter similar to SCTR in [7] called PSEL (Policy SELector) is used, such that a miss in the LRU policy increments the counter while a miss in the BIP policy decrements the counter. If the most significant bit of PSEL is 1 then the selected policy is BIP, if it is 0 then the selected policy is LRU. However, a mechanism is needed to know whether each policy will yield a hit or a miss for each cache access. Qureshi et al. [6] discussed using two additional auxiliary tags, ATD-LRU and ATD-BIP, as in [7] which they called DIP-Global. The cache lines that are requested by the processor are fed to these auxiliary directories and the misses are counted for each policy. Figure 2 shows the DIP-Global tag directories. They also discussed using the Dynamic Set Sampling proposed in [7] to reduce the number of entries in the auxiliary directories to include only samples of the cache lines.

Finally they proposed a new mechanism called Set Dueling that modifies SBAR proposed in [7] so that no auxiliary tag directories are used at all. The two policies are allocated dedicated sets from the cache to count the number of misses to each one of them. While the rest of the sets in the cache are called follower sets. The results of applying these policies to their dedicated sets will be used to increment or decrement PSEL as explained earlier and hence, to decide the policy to be used for the follower sets. Figure 3 shows DIP with Set Dueling.



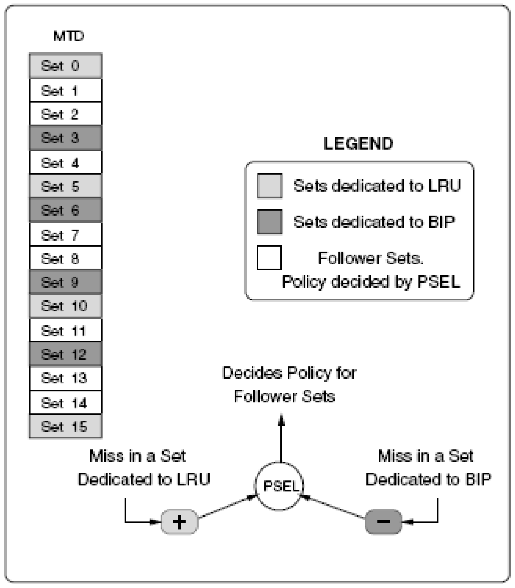


Figure 2: DIP-Global [6] **Figure 3: DIP-Set dueling [6]**

In their research, Qureshi et al [6] indicated according to an analytical model that 32 to 64 dedicated sets are enough to choose the correct policy. The larger the number of dedicated sets, the higher the probability that the policy selection is correct but the more sets that will be using the incorrect policy because of being dedicated. To simplify the process of selecting the dedicated sets, Qureshi et al [6] divided the cache into regions called constituencies. Each constituency contains two dedicated sets, one for each policy. DIP-Set

Dueling requires low hardware overhead: the 10-bit PSEL counter and the 5-bit BIP counter only. Figure 3 shows the hardware required for DIP-Set dueling.

3.3 Other adaptive replacement policies [3 and 9]

In [9] Subramanian et al. proposed a general policy that can adaptively choose among any two replacement policies (e.g. LRU, LFU, FIFO, and Rando). They used the Sampling Based Adaptive Replacement (SBAR) proposed in [7] for the adaptive selection among the policies.

In [3] Megiddo and Modha proposed an adaptive algorithm called Adaptive Replacement Cache (ARC) that dynamically chooses among LRU and LFU. The algorithm implements two additional lists, one that keeps track of pages that were used recently only once (tracking recency), the other list keeps track of pages that were used more than once (tracking frequency). According to misses the policy will adapt the number of pages allocated for each list.

3.4 Shepherd Cache [2 and 11]

Rajan and Govindarajan [2] proposed the Shepherd Cache which tries to mimic the behavior of OPT [1]. Their design splits the cache logically into two caches, one with higher associativity which is the main cache, and another with lower associativity which is the shepherd cache.

The optimal policy proposed by Belady [1] replaces the line that will be used farthest in the future by maintaining what is called the “look-ahead window” which keeps tracks of the lines that will be accessed near in the future. The Shepherd cache mission, in addition to caching lines, is to emulate the look-ahead window of Belady’s policy. For an N-way associative Shepherd cache, there is a counter matrix of N columns for each set in the

cache. These counters, called imminence counters, record the order in which the lines has been accessed since they where entered in the cache. Each time a miss occurs, the new line will be placed in the Shepherd cache on a FIFO basis. The line evicted from the Shepherd cache will be placing the line with the highest counter value in the main cache, since it is the line with the farthest access according to the records. For the counter values that are unknown, which means that the line has never been re-referenced, the LRU policy is used to decide the line to be replaced.

In their study, Rajan and Govindarajan simulated the Shepherd cache and it had much better performance than the LRU and other proposed policies. However, their design requires a significant additional overhead. For an M-way associative cache with N-way associative Shepherd cache, the amount of additional hardware per set includes: an MxN matrix of counters, M flags to differentiate between main cache lines and Shepherd cache lines, M pointers to link each line with its corresponding counter, an array of N counters to keep track of the next counter value, and $\log_2 N$ bits to keep track of FIFO for the shepherd cache. Figure 4 shows the additional hardware required per set for a 16-way set associative cache with a 4-way Shepherd cache and 12-way associative main cache.

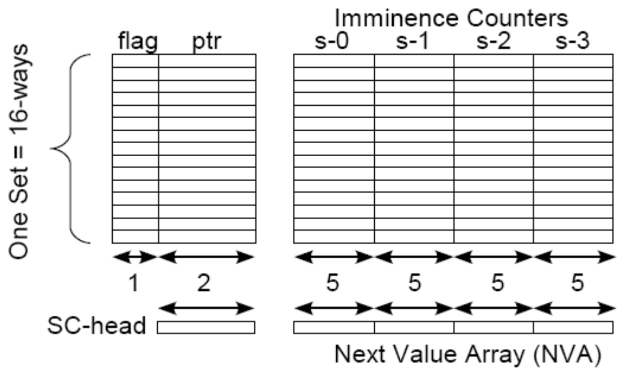


Figure 4: Hardware required per set for a 16 –way cache with a 4-way Shepherd Cache

Zebchuk et al. [11] proposed two variations on the Shepherd cache in order to reduce its companion overhead. In the first proposed variation, the Lightweight Shepherd Cache, if there are lines with unknown imminence value, then LRU will be used to choose the line to be replaced among them. If all lines are with known imminence then the LRU policy will be used to choose the line to be replaced among them. For this policy, the counters are eliminated and replaced with MxN flags to indicate whether the imminence value is known or not. This policy reduced the hardware overhead but it has a weaker emulation of Belady's optimal policy. The second variation, Extra Lightweight Shepherd Cache, further simplifies the Shepherd cache into the following: if the oldest line in the shepherd cache has unknown imminence then it will be evicted. Otherwise the LRU line in the main cache is evicted and the line evicted from the shepherd cache is placed in the LRU position of the main cache. This eliminated the need for the flags of the main cache; flags are needed only for the shepherd cache. Their results showed that the two variations worked similar to the original shepherd cache on the simulated workloads.

4. Recent replacement policies for Chip Multiprocessors (CMPs)

Replacement policies can be used to manage shared caches in CMPs instead of explicit partitioning which is less flexible. For shared caches, the major goals that a replacement policy should achieve include: high performance, low hardware overhead and scalability to the number of cores. [4] This section discusses the proposed extensions of uniprocessor replacement policies so that they are used for shared caches in CMPs.

4.1 Thread-Aware Dynamic Insertion Policy (TADIP) [4]

The LRU policy produces bad performance when used in shared caches since it serves cache accesses from the different cores depending on the demand. This means that a

cache line requested by a core will be placed in the cache and reside until it becomes the least recently used even if it didn't contribute to any hits during its residence in the cache. [4]

In [4] Qureshi et al. studies the allocation of cache resources depending on the benefit rather on the demand. Some workloads do not benefit from more allocated cache resources, these are called cache thrashing or streaming workloads, while others do benefit by contributing to hits, these are called cache friendly workloads. They suggested using LRU with cache-friendly workloads since they benefit from more allocated resources; they deserve to be served depending on their demand. For cache-thrashing workloads, they suggested using BIP, proposed in [6], since this will solve the thrashing problem and at the same time it will limit the resources allocated to these workloads.

Directly extending DIP, proposed in [6], for shared caches is not sufficient since it will choose either LRU or BIP for all the workloads. It doesn't serve each workload with the policy that best suites it. For that, Qureshi et al. [4] proposed the Thread Aware Dynamic Insertion Policy (TADIP). This policy chooses either LRU or BIP for each workload individually. Thus for N cores competing on the shared cache, N-bit string is used to set the policy for each workload: LRU=0, BIP=1. To find the best values for this N bit string, one solution is to execute all the 2^N combinations on the workloads and choose the string that yields the best performance. However this method is impractical; the output varies for different number of cores and different types of workloads and different inputs. Another method which can be used for small number of cores is using Set Dueling (proposed in [6]) so that sets are dedicated to each string for a total of 2^N sets; the string that achieves the best performance is used to decide the used policies. This approach, in

addition of being not scalable, will cause a significant fraction of the cache to be dedicated and using the wrong policy for sets dedicated for other than the chosen string.

To solve this problem, Qureshi et al. [4] proposed two approaches that choose the best policy for each core independently. In the first proposed approach: TADIP-Isolated, a PSEL counter, which is proposed in [6], is allocated for each core. N+1 dedicated sets are required for N+1 strings: one string of 0's called the base-line string in which all cores are set to use LRU, and the rest N strings have only one bit set such that in each string one core is using BIP and the others use LRU. When a miss occurs for the dedicated sets of the base-line string all PSEL counters are incremented. When a miss occurs in one of the other N dedicated sets, only the PSEL counter of the core using BIP will be decremented. Thus the performance of each core under LRU and BIP is used to update its counter independently. However, sometimes replacements done due to references by other cores may affect the misses for the core under each policy. In TAPID-Isolated, when measuring the misses resulting from each policy, it's assumed that all other cores are using LRU which is not the case.

The second approach proposed by Qureshi et al. [4] is TAPID-Feedback. 2N strings are required, 2 strings for each core: in the first string the core is set to use LRU with other cores each using its selected policy, in the second string, the core is set to use BIP with other cores each using its selected policy. This approach allows each core to select the appropriate policy taking in consideration the policies selected for other cores. Both TADIP-I and TADIP-F require relatively low hardware overhead and is highly scalable. Simulation of TADIP-I and TADIP-F proved that they performed better than LRU for all the simulated workloads in terms of speedup and throughput and fairness. Figure 5 shows

DIP, TADIP-I and TADIP-F with the corresponding PSEL counters and set dueling mechanism.

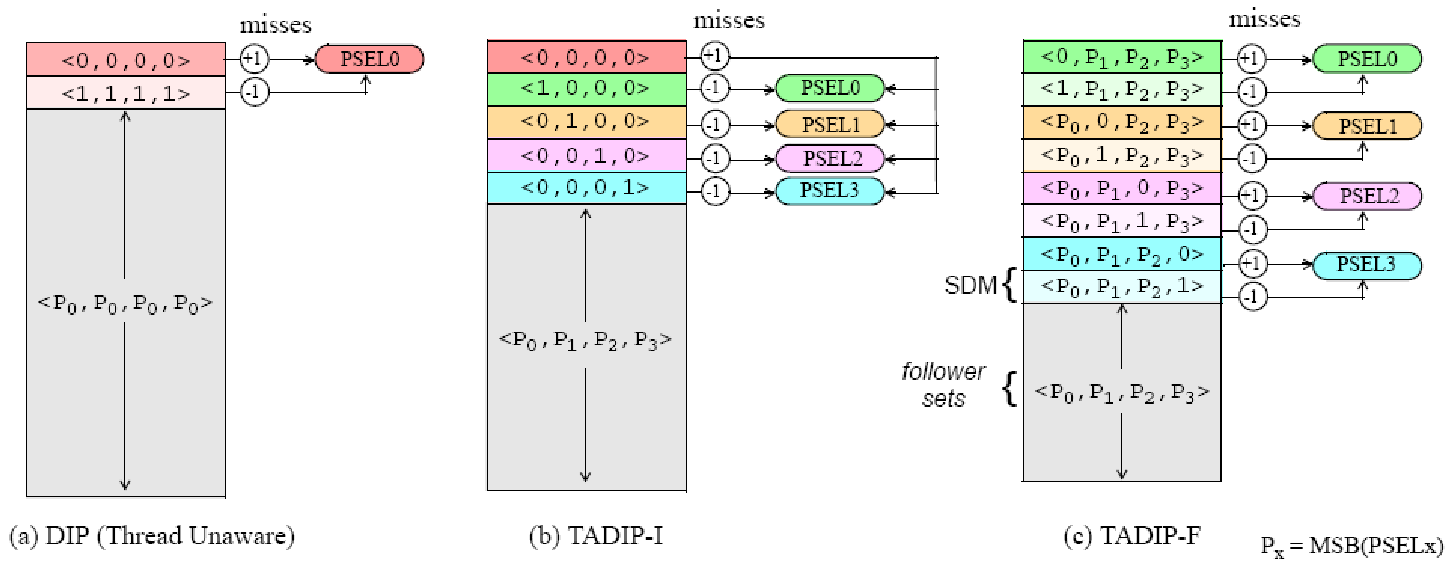


Figure 5: DIP, TADIP-I and TADIP-F each using set dueling [4]

4.2 Double-DIP [2]

In [2] Kron et al. modifies DIP [6] so that it is optimized to work for CMPs by introducing the notion of promotion policies. When the selected policy in DIP is BIP, once a line is re-referenced it will be promoted directly to the MRU position. In CMPs, this may result in one core accessing the cache more frequently to push other core's lines to the least recently positions and being evicted with a higher probability.

Kron et al [2] proposed a policy that does not allow a line re-referenced by BIP to be promoted directly to the MRU position; instead the promotion is done gradually only one step up in the recency stack each time the line is re-referenced. This policy is called the Single-step Incremental Promotion Policy (SIPP). They also proposed using set dueling [6] to dynamically choose among two possible promotion policy; either the conventional policy

or SIPP depending on the performance of each on their dedicated sets. To achieve that, two separate PSEL counters are used, one for the replacement policy (IPSEL) and one for the promotion policy (PPSEL). As proposed in [6], the most significant bit in PPSEL will be used to choose one of the promotion policies.

Finally, Kron et al. [2] optimized their promotion policies to include different promotion increments in a new policy called Dynamic Promotion with Interpolated Increments Policy (D-PIIP). Instead of having only two choices for the promotion value: either 1 or directly to the MRU, they included two additional increments which are 2 and 4. In this case the most significant two bits are used to determine how much the line should be promoted. This policy allowed for moderate increments to be used where needed. The combination of D-PIIP and DIP produces Double-DIP.

Although Double-DIP produces better performance than LRU for CMPs, it still cannot choose different policies for different cores depending on their workloads. However, it produces a better way to manage the recency stack in shared caches where multiple workloads are competing for the cache.

4.3 Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches (PIPP) [10]

In [10] Xie and Loh proposed a policy that combines promotion and insertion policies to implicitly partition the cache among multiple cores. Unlike other approaches that explicitly divide the cache into fixed partitions, this approach allows for "capacity stealing" where parts of the cache that are not used by one core are utilized by other cores.

Xie and Loh [10] uses a mechanism called Utility-based Cache Partitioning (UCP) to decide the number of partitions allocated to each core. This mechanism uses additional tags to compute the number of misses that would occur for each core if it's allocated the entire

cache. Depending on the results it partitions the cache among the cores in order to reduce the number of global hits to the minimum. According to these partitions, each core is given a priority position that indicates where its lines should be inserted in the recency stack. For example for a 16-way set associative cache, the values for the priority positions for 4 cores are {6, 4, 4, 2}. This means the lines of the first core will be always inserted in the sixth position in the recency stack and so on. The line in the least recently used position is the one to be evicted. And the promotion is done one step up in the recency stack each time the line is re-referenced.

PIPP provides a flexible implicit way for partitioning without strictly assuming that the exact number of partitions is assigned for each core at any time. Instead, this approach allows capacity stealing in situations where partitions of one core are not being utilized. Moreover, it allows faster eviction of dead lines since they are not instantly promoted to the MRU as in TADIP.

In order to reduce the additional hardware required for UCP represented in the additional tags, Xie and Loh [10] proposed the In-Cache Estimation Monitor (ICEmon). These monitors are somehow similar to the dedicated sets used in duel sampling proposed in [6]. For each core, there is a dedicated set called an ICEmon. ICEmons are used to approximate the number of misses each cache would incur if it was allocated the whole cache. In each ICEmon the number of lines that can be used by other cores is limited by the Private Monitoring Boundary; lines from other caches cannot be allocated higher recency than this boundary. Their simulation for the proposed policy results in better performance than TADIP for dual core and quad core processors.

5. Conclusion

Many recent studies attempt to propose replacement policies that improve the hit rates of last level caches without adding significant hardware overhead. Some of the proposed policies aim to reduce memory thrashing due to memory intensive workloads (such as LIP and BIP), other policies aim to efficiently utilize MLP (MLP-aware policies). Adaptive policies dynamically choose the replacement policy to be used by dynamically measuring the miss rates of the candidate policies (such as DIP). Other non-adaptive policies include the Shepherd cache, which mimics Belady's OPT.

Many mechanisms are proposed to measure the miss rates for the candidate policies, competing for the adaptive selection, with differing additional hardware requirements. Set dueling which used dedicated sets from the cache proved to be the mechanism with the least hardware overhead.

Some of the policies were optimized to be used for shared caches in CMPs. Some of the policies extended the existing adaptive policy so that it can choose the best policy for each core individually (TADIP). Other policies included the introduction of promotion policies which were combined with existing replacement policies to provide control on the changes in the recency stack (such as double-DIP and PIPP).

All studies included simulation for their proposed policies in comparison with the conventional LRU and sometimes with other proposed replacement policies. However, each simulation used different cache sizes, different caches associativity and different benchmarks. Therefore, this report lacks accurate performance comparison between the studied policies. A unified simulation should be implemented for all policies to compare their performance.

6. References

[1] Belady L. (1966). ***A Study of Replacement Algorithms for a Virtual Storage Computer*****. IBM Systems Journal, vol.5, pp. 78-101.**

[2] Kron J., Prumo B. & Loh G. (2008). ***Double-DIP: Augmenting DIP with Adaptive Promotion Policies to Manage Shared L2 Caches*****. In the 2nd Workshop on CMP Memory Systems and Interconnects (CMP-MSI), Georgia Institute of Technology.**

[3] Megiddo, N. & Modha, D.S. (2004). ***Outperforming LRU with an Adaptive Replacement Cache Algorithm*****. Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference.**

[4] Qureshi M., Jaleel A., Hasenplaugh W., Sebot J., Jr. S. & Emer J. (2008). ***Adaptive Insertion Policies for Managing Shared Caches*****. Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACt'08), pp. 208-219.**

[5] Qureshi M., Jaleel A., Patt Y., Jr. S. & Emer J. (2008). ***Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching*****. IEEE Micro, pp. 91-98.**

[6] Qureshi M., Jaleel A., Patt Y., Jr. S. & Emer J. (2007). ***Adaptive Insertion Policies for High Performance Caching*****. Proceedings of the 34th annual international symposium on Computer architecture (ISCA'07), pp. 381-391.**

[7] Qureshi M., Lynch D., Mutlu O. & Patt Y. (2006). ***A Case for MLP-Aware Cache Replacement*****. Proceedings of the 33th annual international symposium on Computer architecture (ISCA'06). pp. 167-178.**

[8] Rajan K. & Ramaswamy G. (2007). ***Emulating Optimal Replacement with a Shepherd Cache*****. In Proceedings of the 40th International Symposium on Microarchitecture (Micro'07), pp. 445-454.**

[9] Subramanian R., Smaragdakis Y. & Loh G. (2006). ***Adaptive Caches: Effective Shaping of Cache Behavior to Workloads.*** **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'06), pp. 385-396.**

[10] Xie Y. & Loh G. (2009). ***PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches*****. Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09), pp. 174-183.**

[11] Zebchuk J., Makineni S. & Newell D. (2008). ***Re-examining Cache Replacement Policies*****. IEEE International Conference on Computer Design, pp. 671-678.**