Additional Experiments with Probabilistic Escape LIFO

Mainak Chaudhuri
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@iitk.ac.in

12th October, 2009
Last revision: 16th February, 2010

The probabilistic escape LIFO (peLIFO) policy was proposed as a member of the pseudo-LIFO family of replacement policies [1]. This is a deterministic replacement policy that computes certain probabilities to reach a deterministic eviction decision in last-level caches. This brief note includes a few additional experimental results involving the peLIFO policy, a discussion on the impact of L2 cache replacement policies on the memory system performance, a brief specification of the sampled sets used in the set dueling technique, and a low-overhead approximation of peLIFO. This note should be read as an appendix to [1].

1. Impact of Cache Block Size

As shared last-level caches are getting integrated in chip-multiprocessors, there is a trend toward having same cache block size in every level of the cache hierarchy. While one such configuration has been explored in [1], in the following we experiment with two more configurations. In one configuration, we simulate the private L1 caches with 32-byte block size and the shared L2 cache with 64-byte block size. In the second configuration, we simulate a 64-byte block size in all the caches. Everything else is left unchanged as in [1].

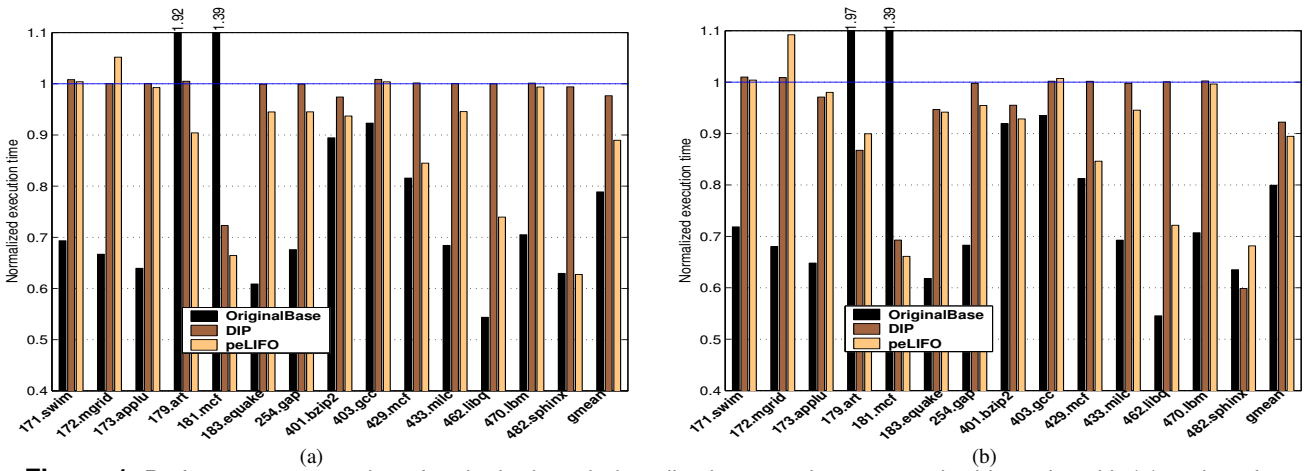


Figure 1. Performance comparison for single-threaded applications running on a cache hierarchy with (a) 32-byte L1 cache block size and 64-byte L2 cache block size, (b) 64-byte L1 cache block size and 64-byte L2 cache block size.

Figure 1 presents the results for the single-threaded applications. We compare DIP and peLIFO. We also include the performance of the original baseline from [1], so that one can understand the impact of reducing the L2 cache block size from 128 bytes to 64 bytes. All the results are normalized to the current baseline. While the original baseline is about

20% better than the current baseline on average, the current baseline performs significantly better than the original one for 179.art and 181.mcf. These two applications are known to have very high volume of conflict misses with little spatial or temporal locality. As a result, a reduction in cache block size leading to a larger number of sets helps improve the situation dramatically. As Figure 1(a) shows, the peLIFO policy performs much better than DIP. However, when both the levels in the hierarchy have the same block size (64 bytes), the performance gap between DIP and peLIFO reduces (see Figure 1(b)).

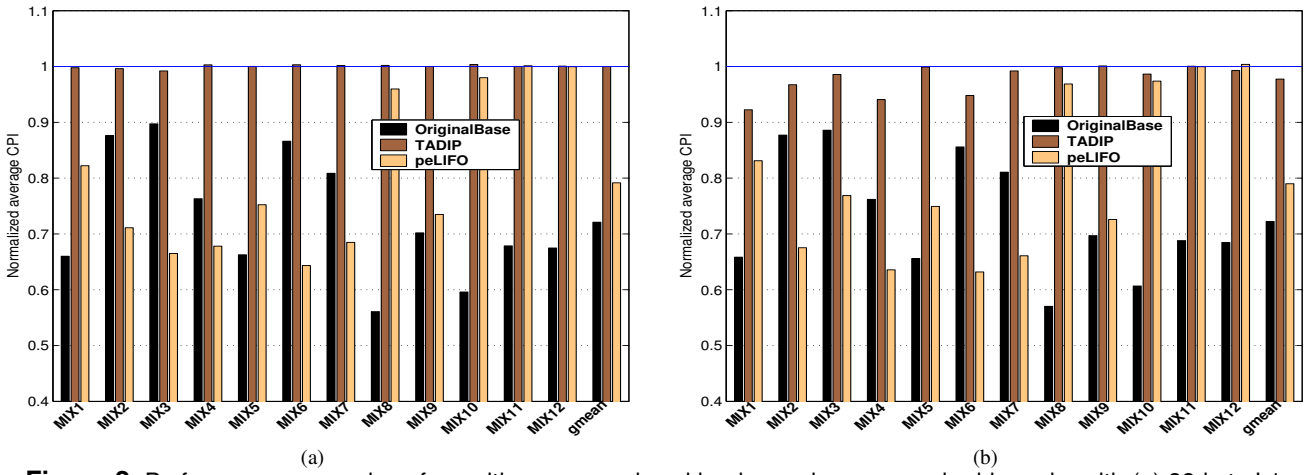


Figure 2. Performance comparison for multiprogrammed workloads running on a cache hierarchy with (a) 32-byte L1 cache block size and 64-byte L2 cache block size, (b) 64-byte L1 cache block size and 64-byte L2 cache block size.

Figure 2 presents similar results for multiprogrammed workloads. In this case also the original baseline performs about 27% better than the current baseline, on average. With the block size equal to 64 bytes in both the levels of the cache hierarchy, the performance of TADIP does improve in several workloads (see Figure 2(b)). However, in both the configurations, peLIFO outperforms TADIP by a large margin.

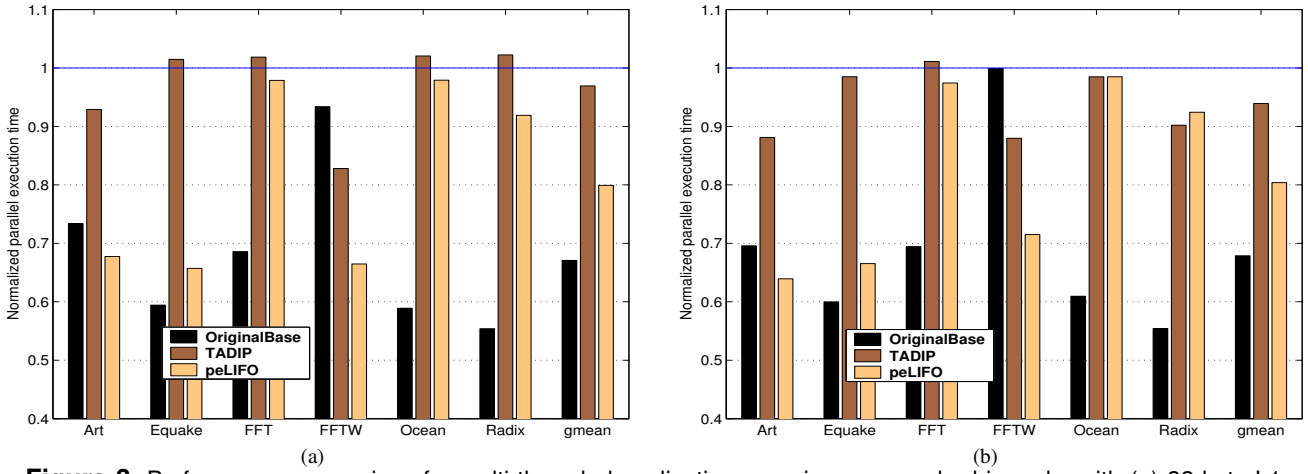


Figure 3. Performance comparison for multi-threaded applications running on a cache hierarchy with (a) 32-byte L1 cache block size and 64-byte L2 cache block size, (b) 64-byte L1 cache block size and 64-byte L2 cache block size.

Figure 3 presents the results for the multi-threaded applications. The original baseline is about 33% better than the current baseline and the peLIFO policy outperforms TADIP by a large margin on both the configurations. As expected, the performance of TADIP improves with respect to the baseline when the block size is made equal to 64 bytes in both the levels of the cache hierarchy (see Figure 3(b)).

2. Impact on Memory System Performance

In this section, we discuss how the L2 cache replacement policies evaluated in [1] interact with the memory system performance. In some of the cases, we have observed significant change in the number of memory bank conflicts due to a change in the L2 cache replacement policies. It is important to factor out these cases when evaluating the fundamental benefits of a replacement policy. We start the discussion by comparing the L2 cache miss counts in the pseudo-LIFO family members with few other policies that offer reasonably good performance.

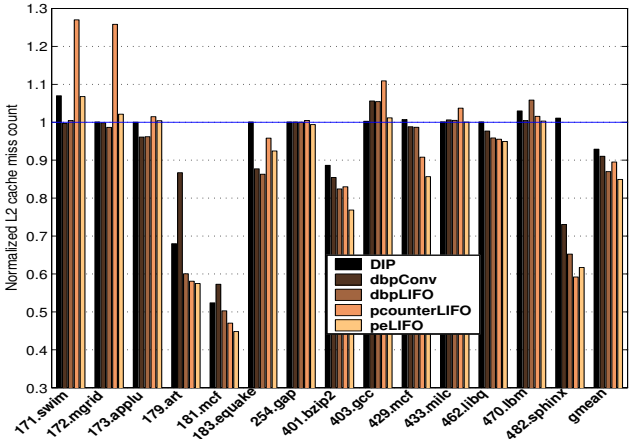
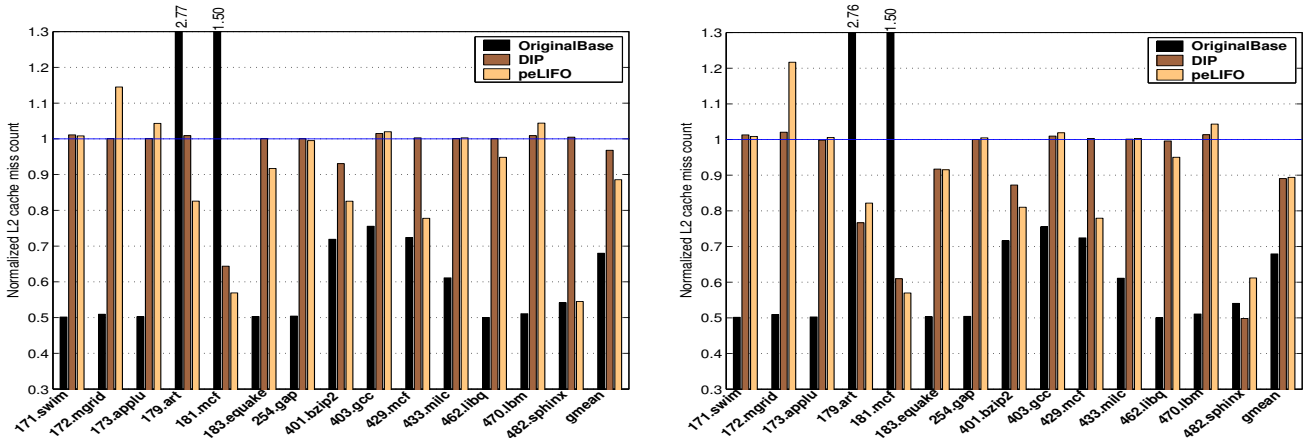


Figure 4. L2 cache miss count normalized to baseline LRU for single-threaded applications running on the cache configuration of [1], i.e., a 32-byte L1 data cache block size and a 128-byte L2 cache block size.



(a) (b)

Figure 5. L2 cache miss comparison for single-threaded applications running on a cache hierarchy with (a) 32-byte L1 cache block size and 64-byte L2 cache block size, (b) 64-byte L1 cache block size and 64-byte L2 cache block size.

Figure 4 shows the L2 cache miss comparison for the cache configuration used in [1], while Figure 5 shows the L2 cache miss comparison for smaller L2 cache block sizes. When these results are compared with the corresponding performance results, an anomaly is observed for 462.libquantum. This application experiences about 5% less L2 cache misses in peLIFO compared to the baseline LRU, but observes a very high performance benefit. We found that the volume of bank conflicts experienced by this application decreases dramatically in all the members of pseudo-LIFO as well as dbpConv when compared to the baseline LRU. We explain this phenomenon below.

In the memory system simulation model, the <controller, bank> id of a physical address is computed by XORing bits $[16:13]$ with the lower four bits of the L2 cache tag, as discussed in [2].[1] Let us assume for simplicity that 462.libquantum is a perfectly streaming application touching data sequentially. As a result, in LRU policy, when a block B is replaced from some set S due to a request for block B' in that set, the address gap between B and B' will be exactly equal to

[1] The lower 13 bits of the physical address are used for column addressing and column offset.

the size of the L2 cache in the virtual address space. This can be true in the physical address space also if the pages are allocated sequentially. If the L2 cache associativity is bigger than or equal to the number of memory banks across all memory controllers, the lower k bits of the tags of B and B' would be identical, where the number of memory banks is 2^k aggregated over all memory controllers. In our model, k is four. If B is dirty, then B and B' will be sent to the same memory bank. In 462.libquantum, our simulation data shows that about 60% of evictions are dirty. This leads to a very high volume of bank conflicts with LRU replacement. When the replacement policy is changed to something other than LRU, the likelihood of a bank conflict between a requested block and the replaced dirty block decreases. This is particularly true for the pseudo-LIFO family members because the address gap between B and B' decreases dramatically. For example, if all evictions happen from the fill stack position zero (a perfect LIFO), the gap between B and B' will be exactly the size of the L2 cache divided by the L2 cache associativity. Referring to Table 4 in [1], we see that the majority of the evictions in 462.libquantum happen from the fill stack position one. This corresponds to a gap of 256 KB between B and B' for a 2 MB 16-way set associative L2 cache. As a result, at least one of the lower four bits of the tags of B and B' will be different (these are $[20 : 17]$ bits of the physical address). Therefore, B and B' are sent to different memory banks. We observed this phenomenon only in 462.libquantum among all the single-threaded applications that we have evaluated.

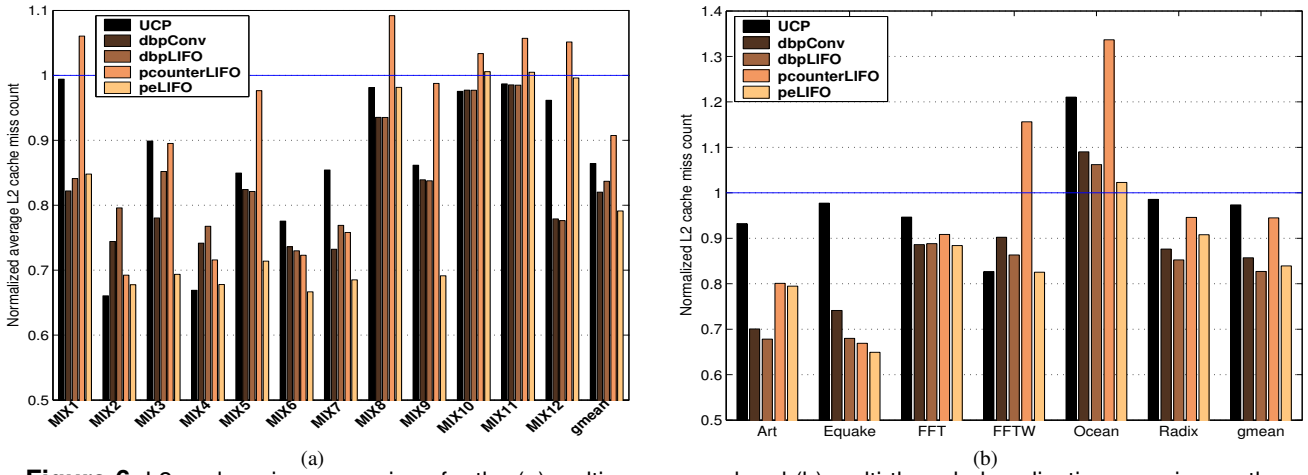


Figure 6. L2 cache miss comparison for the (a) multiprogrammed and (b) multi-threaded applications running on the cache configuration of [1], i.e., a 32-byte L1 data cache block size and a 128-byte L2 cache block size.

Figure 6 shows the L2 cache miss comparison among the policies that offer good performance on the multiprogrammed and multi-threaded workloads running on the cache configuration of [1]. The results are normalized to the baseline LRU policy. For the multiprogrammed workloads, we report normalized average L2 cache miss count i.e.

$$(\frac{1}{4}\sum_i M_i^{NEW})/(\frac{1}{4}\sum_i M_i^{LRU}),$$

where the miss count (M) of each thread i represents the number of L2 cache misses experienced by thread i during its first one billion committed instructions in the multiprogrammed mix. This metric captures the reduction in average L2 cache miss count of a mix when a new policy "NEW" replaces the baseline LRU policy. A comparison of these results with the corresponding performance results presented in [1] uncovers a few anomalies. The anomalies are UCP on MIX1 (saves almost no misses, but observes an 8% reduced average CPI), pcounterLIFO on MIX1 (increases the average L2 cache miss count by about 7%, but observes a 3% reduced average CPI), all policies on Ocean (in general, all policies increase the L2 cache miss count without suffering any performance loss), and pcounterLIFO on FFTW (increases the L2 cache miss count by about 16%, but observes a 30% reduction in execution time). We verified that these effects are due to reductions in the volume of memory bank conflicts when the replacement policy is changed from LRU. Nonetheless, the overall performance trends presented in [1] are accurately brought out in the L2 cache miss trends. However, this analysis points out the importance of the memory system effects when evaluating cache management policies.[2]

[2] One way to address the bank conflict issue of the LRU replacement policy in an $\mathcal{A}$-way set associative last-level cache connected to a memory module with a total number of k banks is to XOR the memory bank bits, the lower $\log_2(k)$ bits of the L2 cache tag, and the next $\log_2(k)$ bits of the L2 cache tag after skipping over the lower $\log_2(\mathcal{A})$ bits. However, this problem arises only if the L2 cache associativity is bigger than or equal to the total number of memory banks aggregated over all the memory controllers.

3. Impact of Memory Latency and Access Scheduling

The experimental results presented so far including those in [1] modeled an FCFS memory access scheduler and paid no attention to modeling DRAM row buffers. Both of these are expected to have a significant impact on the performance results. In this section, we upgrade our memory model in three ways.

First, we model the DRAM latency to take care of row buffer hits and misses. More specifically, we assume DDR2-800 parts with 6-6-6 latency distribution i.e., six cycles each for precharge, activate, and CAS, where the cycles are measured with respect to a 400 MHz clock. Therefore, we have a six-cycle row buffer hit latency and an 18-cycle row buffer conflict latency at 400 MHz. Our model does not distinguish between a closed page row buffer miss and a row buffer conflict because in our model, an open page in a bank is never closed until the row buffer of that bank suffers from an access conflict. As a result, except for the first closed page row buffer miss, every memory access to a bank results in either a row buffer hit or a row buffer conflict. Overall, a row buffer hit takes 15 ns and a row buffer conflict takes 45 ns. In both the cases, however, there is an additional 20 ns DRAM channel transfer latency as explained in [1].

Second, we replace the FCFS memory access scheduler by an FR-FCFS scheduler that honors row hits over other access types. Finally, we change the way the DRAM controller and DRAM bank ids are computed as discussed in the footnote in the previous section.

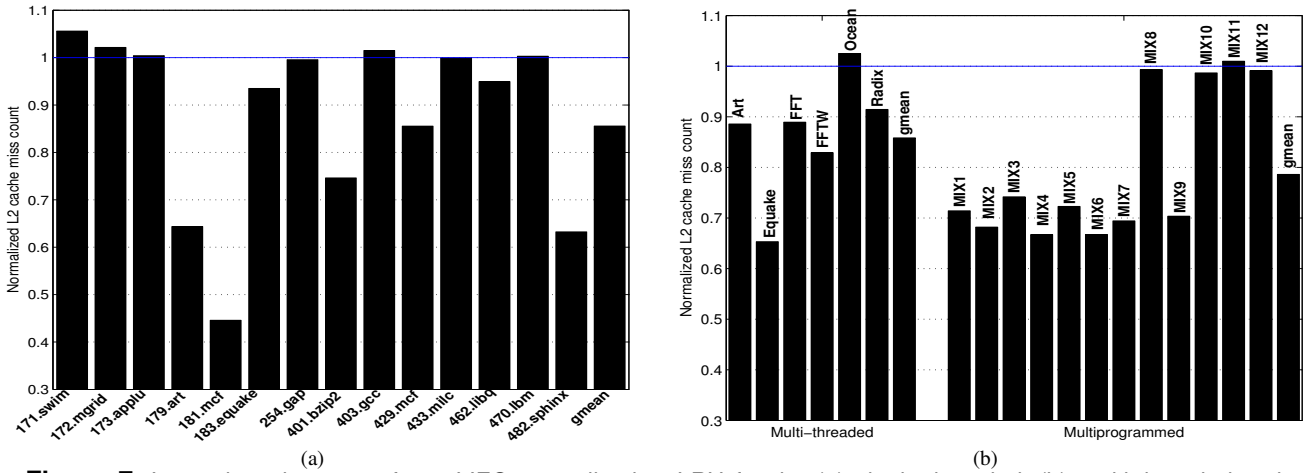


Figure 7. L2 cache miss count for peLIFO normalized to LRU for the (a) single-threaded, (b) multi-threaded and multiprogrammed workloads running on the cache configuration of [1], i.e., a 32-byte L1 data cache block size and a 128-byte L2 cache block size.

Figure 7 first verifies that changing the DRAM configuration does not affect the L2 caches misses saved by the peLIFO policy over LRU. A comparison of these results with those in Figures 4 and 6 confirms that. However, we notice that the percentages of the L2 cache misses saved in single-threaded 179.art and multi-threaded Art have undergone noticeable changes after the DRAM configuration is changed. In both the cases, the percentages of saved misses have dropped in the new configuration. Further investigation into this revealed that the escape points computed for these two applications are very sensitive to the miss-to-refill latency. These two applications suffer from a significant number of short-term conflict misses even in peLIFO i.e., a block is used very soon after it is replaced from the L2 cache and this number naturally depends on the duration for which a block stays in the L2 cache. However, as the DRAM latency drops, the average duration for which a block stays in the cache also drops because a replacement takes place when the block to be refilled arrives at the L2 cache controller. This is the primary cause for the different behavior exhibited by the peLIFO policy in the new configuration.

Figure 8 presents the corresponding performance results. The peLIFO policy continues to save execution cycles by 7%, 10%, and 15% respectively for the single-threaded, multi-threaded, and multiprogrammed workloads, on average, compared to the baseline LRU policy.

4. Details of Set Sampling

The quality of set sampling influences the performance of various policies evaluated in [1]. While ideally one wants to have the sampled sets represent the entire set population in a statistically meaningful manner, it is difficult to ascertain this within a reasonable hardware budget. To improve the reproducibility of the results presented in [1], in the following, we

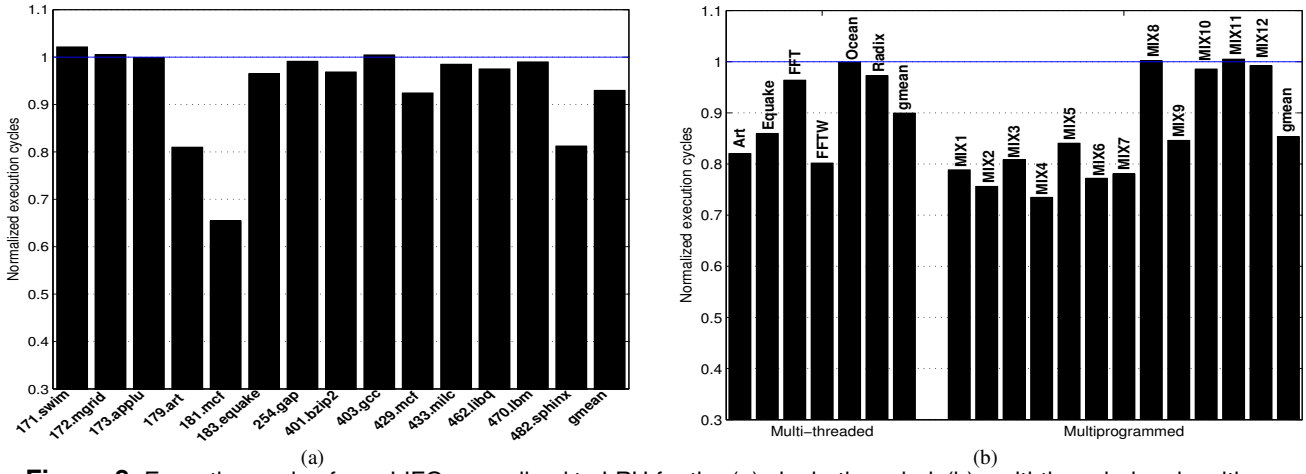


Figure 8. Execution cycles for peLIFO normalized to LRU for the (a) single-threaded, (b) multi-threaded and multiprogrammed workloads running on the cache configuration of [1], i.e., a 32-byte L1 data cache block size and a 128-byte L2 cache block size.

specify the sampling algorithm used in DIP, TADIP, pcounterLIFO, and peLIFO. All these techniques use 32 dedicated set samples per 1 MB L2 cache bank (or equivalently, 64 dedicated set samples per 2 MB L2 cache bank-pair).

In DIP, each 1 MB L2 cache bank dedicates 16 set samples to each of the two dueling policies. The LRU policy is used in the sets that have lower four bits of the index within the bank equal to the next upper four bits of the index i.e., `(index[3:0] == index[7:4])`, where `index` is the set index within a bank. Note that the baseline configuration used in [1] has 512 sets within each bank. The BIP policy is used in the sets that have inverted lower four bits of the index within the bank equal to the next upper four bits of the index i.e., `(∼index[3:0] == index[7:4])`.

For TADIP, each thread is given eight dedicated set samples per policy per 2 MB bank-pair in the multiprogrammed environment, while in the multi-threaded environment each thread is given four dedicated set samples per policy per 2 MB bank-pair. Each 2 MB bank-pair has 1024 sets. Since our multiprogrammed and multi-threaded environments have different number of threads, we use slightly different sampling algorithms in these two cases. In the multiprogrammed environment, we have four threads. Let the upper five bits of the set index within a bank-pair be `upper` i.e., `upper = index[9:5]` and the lower five bits be `lower = index[4:0]`, where `index` is the set index within a bank-pair. In all the sampled sets, `upper & 0x2` is zero i.e., `upper[1]` is zero. All the sets with `(upper == lower)` are dedicated to one thread (say, T_0). All the sets with `(∼upper == lower)` are given to T_1. All the sets with `((upper XOR 0xa) == lower)` are given to T_2. All the sets with `((upper XOR 0x15) == lower)` are given to T_3. Within the set samples dedicated to a thread, those with `upper[0]` equal to zero are dedicated to LRU, while the rest are dedicated to BIP. For the multi-threaded environment, we partition the sets dedicated to thread k in the multiprogrammed environment to satisfy threads $2k$ and $2k+1$ for $0 \leq k \leq 3$. This partitioning is done based on `upper[2]`. If this bit is zero, the set is given to thread $2k$; otherwise it is given to thread $2k+1$. For example, in the multi-threaded environment, T_0 is given the set samples with `(upper == lower)` and `(upper[1] == 0)` and `(upper[2] == 0)`. On the other hand, T_1 is given the set samples with `(upper == lower)` and `(upper[1] == 0)`, but `(upper[2] == 1)`. Similarly, T_6 is given the set samples with `((upper XOR 0x15) == lower)` and `(upper[1] == 0)` and `(upper[2] == 0)`, while T_7 is given the set samples with `((upper XOR 0x15) == lower)` and `(upper[1] == 0)`, but `(upper[2] == 1)`. Within the set samples dedicated to a thread, those with `upper[0]` equal to zero are dedicated to LRU, while the rest are dedicated to BIP. In both the environments, the simulated algorithm is TADIP with feedback.

The peLIFO and pcounterLIFO designs have four competing policies $\mathcal{P}_1, \ldots, \mathcal{P}_4$ with $\mathcal{P}_4$ being the LRU replacement policy. However, there is no thread-awareness in any of these policies. The sampled sets are same in both peLIFO and pcounterLIFO. Each of the four dueling policies gets sixteen dedicated set samples per 2 MB bank-pair. Let `upper` and `lower` be defined as in the aforementioned discussion. All sampled sets have `upper[1]` equal to zero. All sets with `(upper == lower)` are dedicated to $\mathcal{P}_1$. All sets with `(∼upper == lower)` are dedicated to $\mathcal{P}_2$. All sets with `((upper XOR 0xa) == lower)` are dedicated to $\mathcal{P}_3$. All sets with `((upper XOR 0x15) == lower)` are dedicated to LRU.

5 Yet Another Cousin of peLIFO

Recall that for computing the escape probability correctly, the fill stack position of the last hit experienced by a block needs to be maintained. On a hit to a block currently at fill stack position p, the k^{th} entry of the $epCounter$ array is incremented by one for all $k \in [h, p-1]$, where h is the last hit position of the block in the fill stack. Let us consider the following alternative, which does not require the last hit position. We increment $epCounter[p]$ by one when a block experiences a hit at fill stack position p for the first time. Subsequent hits to this block at the same fill stack position does not affect the contents of $epCounter$. At the end of each epoch, the sum of the $epCounter$ array is computed and the smallest power of two bigger than or equal to this sum is used as the denominator in the probability computation. We have observed several epochs in some of the workloads which do not experience any hits in certain last-level cache bank-pairs. In such cases, the denominator is taken equal to the number of fills in the epoch. Next, $epCounter[k]$ is updated with $\sum_{i=k+1}^{\mathcal{A}-1} epCounter[i]$ for all $k \in [0, \mathcal{A}-1]$ in an $\mathcal{A}$-way cache. Once this is done, the algorithm proposed in [1] can be executed for computing a very crude estimate of the escape probabilities and the rest of the replacement algorithm remains unchanged. We will refer to this approximation of peLIFO as peLIFOLite.

Although the peLIFOLite policy saves $\log_2(\mathcal{A})$ bits per cache block compared to the peLIFO proposal in an $\mathcal{A}$-way set associative cache, it is important to understand why this method fails to compute the escape probabilities correctly and as a result, the knees in the escape probability curve may not carry any meaning. The peLIFOLite policy arrives at a replacement decision by using the probability of experiencing unique hits (as opposed to the probability that a block would experience hits) beyond each fill stack position. The difference between the escape probability (P_e) computed by peLIFO and the escape probability (P_e^{Lite}) computed by peLIFOLite will become clearer by considering a knee in the respective escape probability curves. A knee at fill stack position k in P_e curve necessarily implies that a number of blocks become dead beyond position k. On the other hand, a knee at fill stack position k in the P_e^{Lite} curve only implies a drop in the density of unique hits beyond fill stack position k. This may not translate to blocks becoming dead. For example, suppose each block in a population of blocks experiences at least one hit at all fill stack positions up to k. However, beyond fill stack position k, on average, half of the blocks in the population experience at least one hit at odd fill stack positions while the other half experience at least one hit at even fill stack positions. Given this behavior, the P_e curve would not have any knee at fill stack position k because there isn't any block that transitions into dead state while crossing this fill stack position. On the other hand, the P_e^{Lite} curve would show a prominent knee at fill stack position k.

It is useful to note when the P_e^{Lite} curve would inherit the knees of the P_e curve. Define for each block B, a function h_B such that $h_B(k)$ for fill stack position $k > 0$ is one if the block B experiences at least one hit at position k; otherwise $h_B(k)$ is zero. Note that it is enough to define h_B only for fill stack positions bigger than zero. We state the following theorem, which can be proved easily.

Theorem 5.1. *The P_e^{Lite} curve would have all non-trivial knees of the P_e curve at identical fill stack positions if h_B, when plotted against fill stack position, has at most one rising edge and at most one falling edge for all blocks B and in the case where both the edges exist, the rising edge must come before the falling edge i.e., h_B is either monotonic or bitonic of one particular type.*

Note that in such situations the P_e^{Lite} curve can have other additional knees also. The good news is that h_B is usually monotonic or bitonic of the required type and does not fluctuate much. For the single-threaded, multiprogrammed, and multi-threaded workloads, on average, 92%, 95%, and 94% of the L2 cache blocks possess the desired h_B function. These percentages are first computed for each application and the geometric mean for a group of applications (i.e., single-threaded, multiprogrammed, or multi-threaded) is reported here. Note that even if a small population of blocks exhibits arbitrary square-wave h_B functions (as opposed to monotonic or bitonic sequences), the peLIFOLite policy will end up taking the right decision for a vast majority of the blocks. Overall, the peLIFOLite policy can be implemented with just five additional bits per cache block in a 16-way set associative cache i.e., $(1 + \log_2(\mathcal{A}))$ additional bits per cache block in an $\mathcal{A}$-way set associative cache.

Figures 9(a), 9(b), and 10 present a comparison of L2 cache miss count among DIP (or TADIP, as appropriate), peLIFO, and peLIFOLite on a cache configuration with equal block sizes across the hierarchy for single-threaded, multiprogrammed, and multi-threaded workloads. The results are very encouraging and the last-level cache miss volume of peLIFOLite matches that of peLIFO, even though the storage overhead of the former is almost half of the latter. For comparison, we also include the L2 cache miss volume of the baseline used in [1].

Acknowledgments

The author thanks Aamer Jaleel for useful discussion while analyzing the performance anomaly in 462.libquantum.

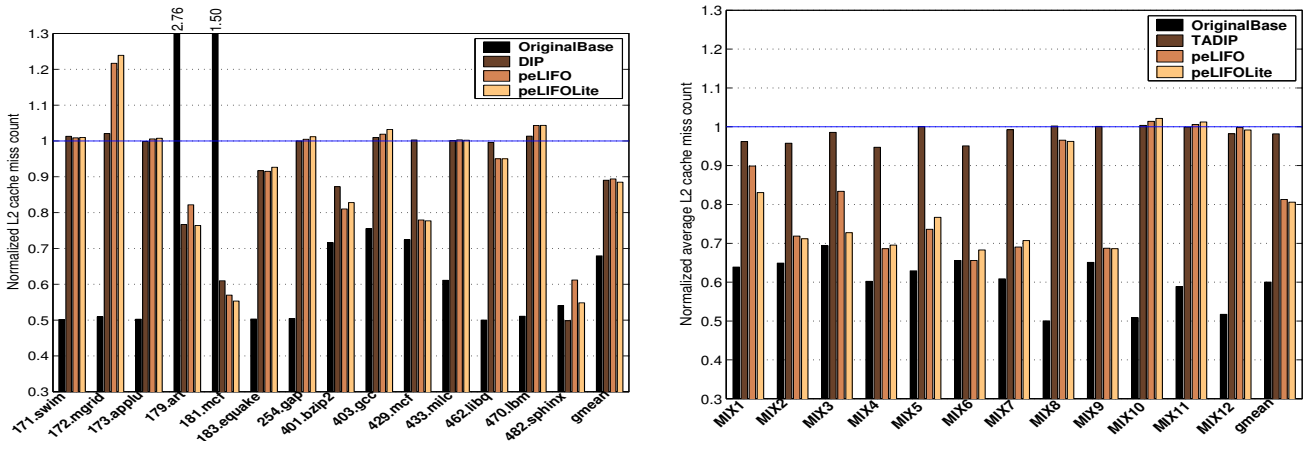


Figure 9. L2 cache miss count normalized to LRU for (a) single-threaded and (b) multiprogrammed workloads running on a cache configuration with 64-byte L1 data cache block size and a 64-byte L2 cache block size.

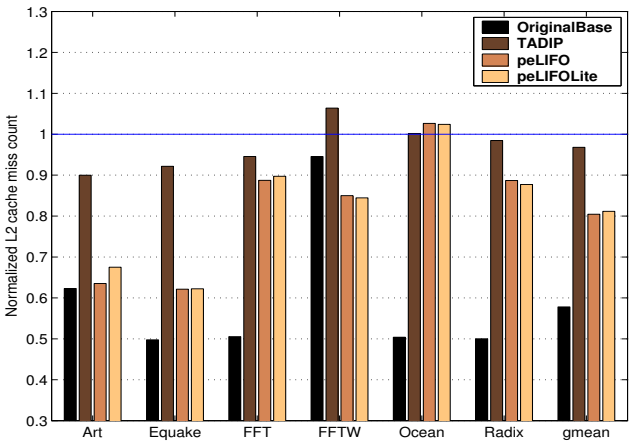


Figure 10. L2 cache miss count normalized to baseline LRU for multi-threaded applications running on a cache configuration with 64-byte L1 data cache block size and 64-byte L2 cache block size.

References

[1] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, December 2009.

[2] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 32–41, December 2000.