

Less Reused Filter: Improving L2 Cache Performance via Filtering Less Reused Lines

Lingxiang Xiang, Tianzhou Chen, Qingsong Shi, Wei Hu

College of Computer Science
Zhejiang University, Hangzhou, China
{lxxiang, tzchen, zjsqs, ehu}@zju.edu.cn

ABSTRACT

The L2 cache is commonly managed using LRU policy. For workloads that have a working set larger than L2 cache, LRU behaves poorly, resulting in a great number of *less reused lines* that are never reused or reused for few times. In this case, the cache performance can be improved through retaining a portion of working set in cache for a period long enough. Previous schemes approach this by bypassing never reused lines. Nevertheless, severely constrained by the number of never reused lines, sometimes they deliver no benefit due to the lack of never reused lines.

This paper proposes a new filtering mechanism that filters out the less reused lines rather than just never reused lines. The extended scope of bypassing provides more opportunities to fit the working set into cache. This paper also proposes a *Less Reused Filter (LRF)*, a separate structure that precedes L2 cache, to implement the above mechanism. LRF employs a *reuse frequency predictor* to accurately identify the less reused lines from incoming lines. Meanwhile, based on our observation that most less reused lines have a short life span, LRF places the filtered lines into a small *filter buffer* to fully utilize them, avoiding extra misses.

Our evaluation, for 24 SPEC 2000 benchmarks, shows that augmenting a 512KB LRU-managed L2 cache with a LRF having 32KB filter buffer reduces the average MPKI by 27.5%, narrowing the gap between LRU and OPT by 74.4%.

Categories and Subject Descriptors

B.3.2 [Design Styles]: Cache memories

General Terms

Performance, Design

Keywords

Cache filtering, Less reused line

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...\$5.00.

1. INTRODUCTION

The rapidly growing gap between memory latency and processor speeds emerged as the primary bottleneck to high performance computers. Caches are used to bridge this gap by exploiting temporal and spatial locality. In cache hierarchy, the last level cache (L2 cache in this study) plays a crucial role in overall performance since a miss in it stalls the processor for hundreds of cycles. In some applications, the stall time due to misses in L2 cache even takes 80% of total execution time [1].

Theoretically, in order to achieve the lowest miss rate, the L2 cache should employ the optimal replacement policy (OPT) [2] which selects the cache line that is accessed farthest in the future as the victim for replacement. It is obvious that OPT policy is impractical because it relies on the knowledge of future. Instead, least recently used (LRU) policy is commonly used in practice due to its simplicity. Despite performing well for non-memory-bound workloads, LRU behaves poorly for memory-bound workloads whose working set is greater than the cache, resulting in a huge gap between LRU and OPT. Numerous studies have investigated sophisticated policies to bridge the gap [3, 4, 5, 6, 7]. However, the key limitation of LRU in this situation is that the working set cannot fit into the cache rather than the workload is LRU-averse. In fact, the gap would be dramatically narrowed if sufficient cache capacity were provided.

The solution in this case, as pointed out by Qureshi et al. [8], is to retain a portion of working set (shrunken working set) in cache long enough as what the OPT does, whereby at least the retained part contributes to cache hits. Since there exist a great number of *less reused lines* that are never reused or reused for few times in L2 cache, bypassing the never reused lines, of course, is an approach to shrink the working set. But the performance of such approach is severely constrained by the number of never reused lines. Sometimes even all never reused lines are perfectly bypassed, the shrunken working set is still larger than the cache.

This paper proposes a new *reuse frequency* (the number of references to a line when it presents in cache) based filtering mechanism to achieve the goal of shrinking working set. Through preventing the less reused lines from entering L2 cache, the proposed mechanism increases the chance of fitting the working set into the cache, overcoming the limitation of previous mechanisms [9, 8] that only bypass the never reused lines.

This paper also proposes a *Less Reused Filter (LRF)* to implement the above filtering mechanism. LRF is a separate structure that precedes the conventional L2 cache. In

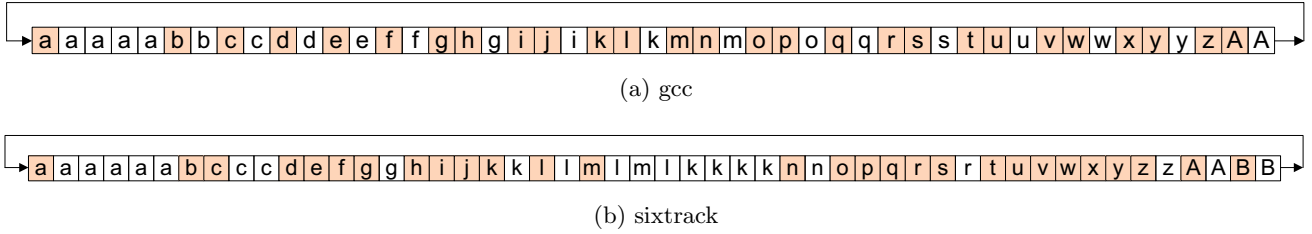


Figure 1: Representative access sequences that occur in Set 0 of 512KB 16-way set-associative L2 cache for (a)gcc and (b)sixtrack. White blocks represent hits, while shadow blocks represent misses.

LRF, a *reuse frequency predictor* is used to predict the reuse frequency for a miss line according to its previous reuse information. Based on the reuse frequency prediction, LRF identifies the less reused lines and filters them out. Once enough less reused lines are filtered out, a portion of working set which consists of non-less reused lines will be retained in L2 cache and therefore the retained part will be protected from miss. This maintains a hit rate proportional to the L2 cache size for the workloads that have a working set larger than L2 cache. Meanwhile, the filtered less reused lines are placed into a *Filter Buffer (FB)* located in LRF. The observation that *the less reused lines have a short life span* enables us to use a fairly *small* FB to fully utilize and fast retire most of them. For those lines that FB cannot retire, we insert them back to L2 cache or discard them according to the *retirement threshold* which is adaptable to workload characteristics. Hence, these filtered lines will not cause extra misses.

We show that the LRF can significantly improve the performance of L2 cache for memory-bound workloads without any modification to the L2 cache replacement policy. On the other hand, it also aids non-memory-bound workloads by accurately kicking out the less reused lines. Experimental results based on execution driven simulations show that augmenting a conventional 512KB 16-way LRU-managed L2 cache with a LRF including a 512-entry filter buffer reduces the average MPKI across 24 SPEC 2000 benchmarks by 27.5%. This translates into an average instruction per cycle (IPC) improvement of 12.9%. Our cache architecture outperforms other recent proposals including the V-Way cache [10], the dynamic insertion policy [8] and the shepherd cache [7] with equal number of overall data lines.

The rest of the paper is organized as follows. Section 2 further motivates the proposed technique. Section 3 presents two key observations that enable the less reused filter. Section 4 elaborates on the less reused filter and the filtering approach. We present details about our simulation environment and benchmarks in Section 5 and experimental results in Section 6. Related work is discussed in Section 7, followed by our conclusions in Section 8.

2. MOTIVATION

2.1 Problem

It has been noted in several studies [5, 10, 8] that a large number of cache lines lack temporal locality in the conventional LRU-managed L2 cache, especially when the size of working set exceeds cache size. We demonstrate this by two real samples.

Figure 1 illustrates two representative reference sequences¹ occurring in the first set (Set 0) of a 512KB 16-way set-associative L2 cache for two SPEC 2000 benchmarks, *gcc* and *sixtrack*.² In *gcc*, the loop processes on a 27 sized working set, from line *a* through line *A*. Unfortunately, the cache set only has a capacity up to 16, which results in a failure mode of LRU [11]. In every iteration, the first access to an address causes a miss. The similar situation can also be observed in *sixtrack*. Even worse, due to the filtering effect of L1 cache, L2 cache lines inherently do not exhibit high temporal locality, i.e., after a miss line is inserted in L2 cache, no or few following references will access (and hit) it. This amplifies the negative effect of the failure mode of LRU policy.

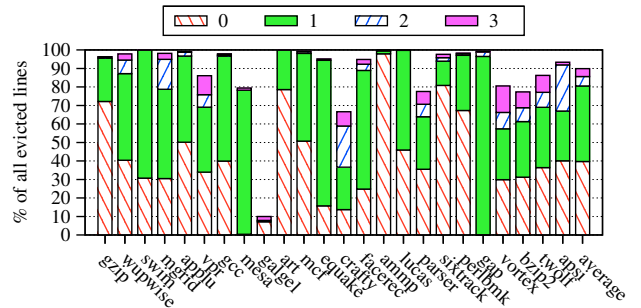


Figure 2: Fraction of the less reused cache lines for SPEC 2000 benchmarks in a 512KB 16-way set-associative L2 cache with LRU replacement policy.

The above two factors together cause the poor locality of L2 cache lines. For convenience, we define the number of references to a cache line between its insertion and eviction as its *reuse frequency*, and call a line with reuse frequency N as a N -reused line. Figure 2 quantifies the fraction of lines with reuse frequency less than 4 in all evicted L2 lines across all 24 SPEC 2000 benchmarks.³ We can see that for all except *galgel*, most evicted cache lines are never reused or only reused few times. Obviously, zero-reused (never reused) and one-reused cache lines take two major portions, accounting for 39.7% and 40.8% on average.

Some cache bypassing schemes [8, 9], we call which zero-

¹For clarity, we replace the line address with letters.

²The detailed experimental methodology is described in Section 5.

³*eon* and *fma3d* are omitted throughout the paper since the miss rate of them is extremely low ($< 0.1\%$).

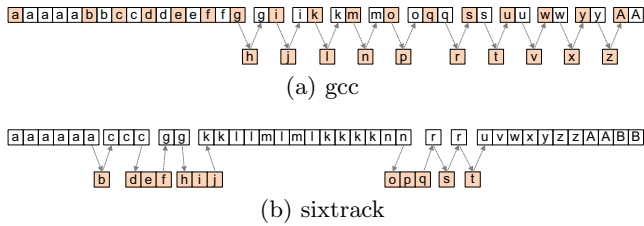


Figure 3: Fitting the working set into the cache via bypassing zero-reused lines. In each subgraph, the upper line indicates the access in L2 cache and the lower line indicates the filtered access.

bypassing schemes, avoid placing the zero-reused lines in cache since they waste the cache capacity unnecessarily. In the context of L2 cache, the essence of these schemes is trying to fit the shrunken working set into the cache. We demonstrate an ideal zero-bypassing scheme in Figure 3, assuming that all zero-reused lines can be perfectly identified. As depicted in Figure 3(b), it succeeds in *sixtrack*. After zero-reused lines (b, d, e, f, j, o, p, q, s, t) are bypassed, the retained working set with 16 lines can wholly fit into the cache. On the contrary, the zero-bypassing scheme fails in *gcc*. Even if all 10 zero-reused lines are bypassed, the size of retained working set is still larger than the capacity of L2 cache. No miss can be eliminated. In fact, if the cache set is less than 11-way, the zero-bypassing scheme will fail in *sixtrack* for the same reason.

2.2 Solution: Less Reused Filtering

The main drawback of zero-bypassing schemes is that they only focus on zero-reused lines. But as severely limited by the number of zero-reused lines, sometimes these schemes fail to deliver benefits due to the lack of never reused lines, especially when the working set exceeds the capacity of L2 cache. Worse still, the growing memory requirement in modern applications makes this case prevalent.

To overcome the limitation of zero-bypassing, we proposes the *less reused filtering* that filters less reused lines (e.g., the lines with reuse frequency less than two) rather than just bypasses zero-reused lines. The new filtering mechanism extends the scope of bypassing, providing more opportunities to fit the working set into cache.

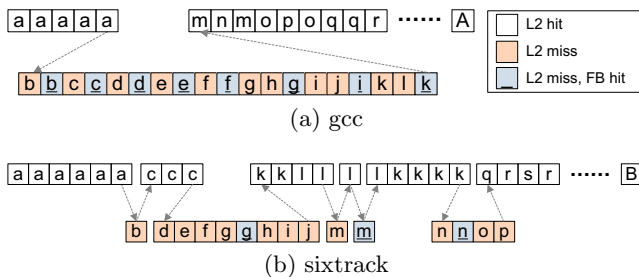


Figure 4: Fitting the working set into the cache via filtering less reused lines.

Reconsider the samples of *gcc* and *sixtrack*. In *gcc*, all lines except a are less reused initially (in the first iteration).⁴

⁴For demonstration, here we assume the less reused lines

In the second iteration, lines *bcdefghijkl* are filtered in sequence because they are less reused. Now, the lines left in the cache are *amnopqrstuvwxyza*. And the following references from *m* to *A* hit in cache. When the third iteration starts, the filtered lines and the retained lines get into a stable state as illustrated by Figure 4(a). The shrunken working set (*amnopqrstuvwxyza*) fits with the cache size, and is protected from miss, leaving only the filtered lines still suffering from miss. Similarly, Figure 4(b) illustrates the stable state for *sixtrack*. Note that although the lines *mnopqrstuvwxyza* in *gcc* were less reused initially, they become non-less reused lines in later iterations.

Nevertheless, the filtered lines will cause extra misses if simply bypassed. For example, two successive references miss on line *d* in Figure 4(a). If line *d* were not filtered, at least the second reference would hit. But as we can observe from the figure, the reuse distance for less reused lines is short. So if the filtered lines are placed in a small *filter buffer (FB)* instead, most of them can be fully utilized and fast retired. The letters with underline are those cache hits retrieved by a 2-entry filter buffer.

Overall, the above filtering mechanism that filters and retires the less reused lines reduces the miss rate from 27/47 to 11/47 per iteration for the reference sequences of *gcc*, and from 28/50 to 12/50 per iteration for that of *sixtrack*.

We refer to the structure that implements such a filtering mechanism as a *Less Reused Filter (LRF)*. So far, two problems still need to be solved for LRF to function properly: 1) *How can we determine whether an incoming line is less reused or not*, and 2) *How many filtered less reused lines can be fast retired in the LRF*. We explore them in the next section.

3. WHY DOES LESS REUSED FILTER WORK?

3.1 Reuse Frequency Prediction

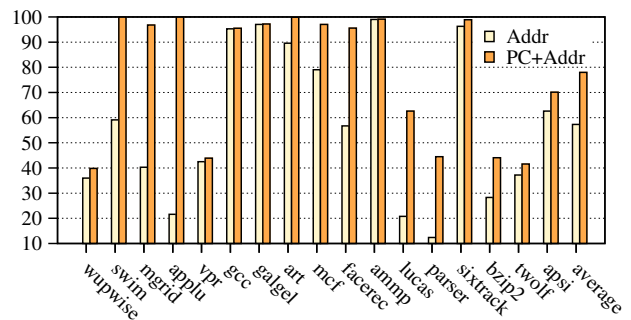


Figure 5: Prediction accuracy of reuse frequency.

As discussed in Section 2.2, the less reused lines are filtered out *before* they enter into the cache. Thus, to enable the LRF, we must employ a mechanism which can determine whether a miss line is less reused before its insertion. Obviously, the determination relies on the future reuse frequency of a cache line. This means such a mechanism is essentially equal to the reuse frequency prediction.

The repetitive memory behavior of a program, as shown in previous work, yields various cache reference patterns that can be perfectly identified. We leave the identification of the less reused lines to following sections.

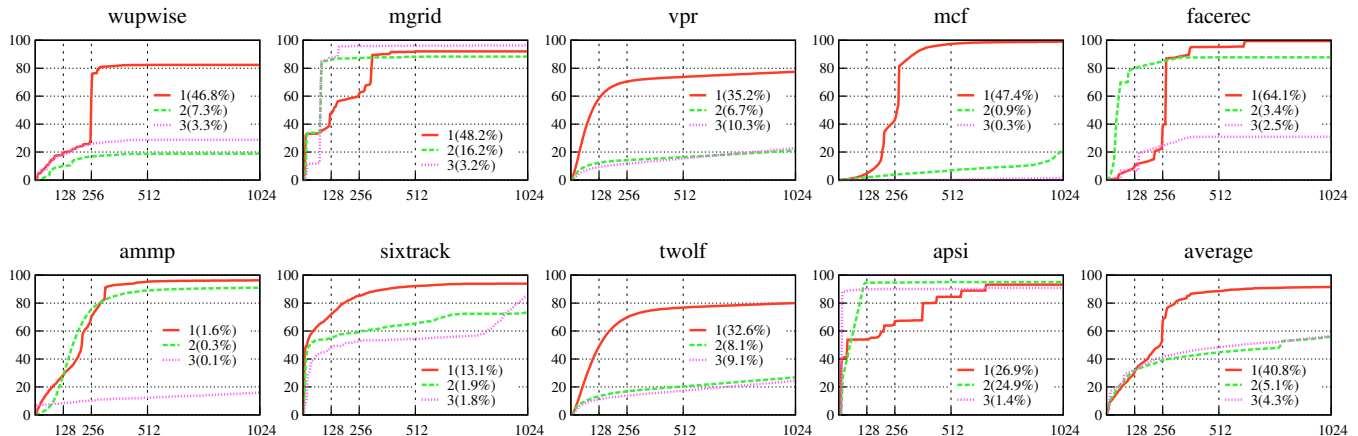


Figure 6: Cumulative distribution of life span for the 1-, 2-, and 3-reused lines in 9 selective SPEC 2000 benchmarks and all 17 benchmarks with MPKI > 1 (“average”). Each cumulative distribution curve shows the percentage of corresponding reused lines have life span less than or equal to a given value.

are regular and predictable, such as next cache way [12] and next cache set [13] to be accessed, live time [14] and instruction sequences [15] to a cache line, and words usage histories in a cache line [16]. To our knowledge, however, no work has given attention to the reuse frequency prediction. Thus, we exploit the predictability of the reuse frequency in this section.

In the experiment, we use the LAST algorithm to predict the reuse frequency for cache line. During the execution, when a cache line missed, we predict that its reuse frequency this time will be the same as its last reuse frequency. Then, we verify the correctness of this prediction after it is replaced.

The columns with the label “Addr” in Figure 5 shows the percentage of the correct prediction for SPEC 2000 benchmarks with MPKI > 1⁵ using the line address to index the last reuse frequency. Some benchmarks exhibit very high predictability (> 90%), while others do not, which, however, does not mean the reuse frequency is inherently unpredictable for benchmarks with low prediction accuracy. In fact, the reason is that using only addresses as indexes cannot distinguish different program phases, within which the same memory address has different reuse behavior. If the PC that misses in a specific address is combined to record the last reuse frequency instead, as depicted by the columns “PC+Addr”, the prediction accuracy rises in all benchmarks, especially in *swim*, *mgrid*, *applu*, *facerec*, and *lucas*. On average, the address and PC combined approach increases the prediction accuracy from 57.3% to 78.0%. These results indicate that the reuse frequency of cache lines is predictable.

3.2 Life Span of the Less Reused Lines

We call the number of *all* cache misses that occur in all cache sets between the insertion and the death (last touch before eviction) of a cache line as its *life span*, which measures the possibility of using a global buffer with specific size to retire cache lines. Instinctively, it is easy to see that life span increases with the reuse frequency. Since LRF only

⁵In these benchmarks, there exist enough cache misses to study the properties of less reused lines.

focuses on the less reused lines, the buffer required to retire the less reused line does not need to be very large.

Figure 6 plots the cumulative distribution of life span for the 1-, 2-, and 3-reused lines in 9 selective SPEC 2000 benchmarks and the average of all SPEC 2000 benchmarks with MPKI > 1.⁶ The shape of cumulative distribution curve varies from benchmark to benchmark. But in general, the 2- and 3-reused lines have longer life span than 1-reused lines. In particular, for 1-reused lines, which take a notable fraction (40.8%) of all lines, the figure shows the major portion of them concentrates at a short life span. On average, almost 70% and 90% 1-reused lines have life span less than 256 and 512. The result implies that it is effective to filter the 0- and 1-reused lines due to their short life span and huge portion, and hence a small sized buffer (e.g., 512-entry) can retire most of them.

4. DESIGN OF LESS REUSED FILTER

4.1 Architecture

The organization of the conventional L2 cache preceded by a small LRF is depicted in Figure 7. The new cache organization is based on the dual cache paradigm. To serve a request from L1 data or instruction cache, L2 cache and LRF are searched simultaneously. If both L2 cache and LRF are missed, the request is sent to the memory controller. In the data stream from memory up to the higher level of the memory hierarchy, LRF predicts the reuse frequency for each cache line, filtering out less reused lines and only admitting non-less reused lines into L2 cache.

The LRF consists of three main components: (1) a reuse frequency predictor, (2) a small filter buffer, and (3) a set of shadow tags. We describe the detailed design of them in the following sections.

4.1.1 Reuse Frequency Predictor

The predictor maintains the recent reuse frequencies of lines not present in the cache. When a miss occurs for

⁶The life span of 0-reused lines is always zero since they are dead immediately after their insertion.

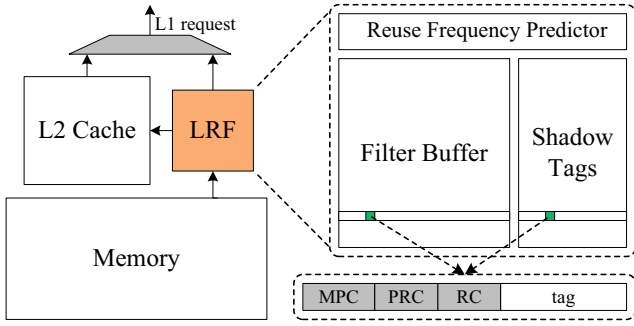


Figure 7: The organization of the conventional L2 cache preceded by a less reused filter (LRF).

a cache line, its reuse frequency is predicted based on its reuse frequency history that is stored in the predictor. In Section 3.1, we have shown that a simple LAST algorithm, which requires only one history per entry to work, has high prediction accuracy for most benchmarks. However, the proposed architecture still leaves room for further adoption of sophisticated prediction algorithms [17].

Each entry in the predictor records the value of reuse frequency. As shown in Figure 2, 80.5% cache lines are reused under two times. Thus, a small 2-bit entry, in which 11_2 indicates a non-less reused frequency, is sufficient for both high coverage and storage efficiency.

To index an entry in the predictor, we use the address and PC combined approach as described in Section 3.1. An index is constructed by concatenating an address part and a PC part. The former contains the lower order bits of a line address. We find the lower 12 bits of line address is enough to distinguish most cache lines in our simulation environment. The latter distinguishes program phases. Since the number of instructions that access the same L2 line is usually small, it does not need as many bits as the address part. In this study, we XOR $MPC[5:2]$ and $MPC[9:6]$ to form a 4-bit PC part, where MPC is the PC of the load/store instruction that misses on the address.⁷

4.1.2 Filter Buffer

In addition to filtering and fast-retiring the less reused lines as discussed earlier in Section 3.2, a Filter Buffer (FB) can be used to remedy the mispredictions in the prediction mechanism. For example, for a line that is predicted to be zero-reused but actually reused for one time, if we place it in FB instead of bypassing it, it still has the chance to provide cache hit.

Due to the short life span of most less reused lines, a small global FB with 512 entries is effective to fully utilize the filtered lines. Compared to a typical L2 cache with 8192 or more entries, the space overhead of LRF is quite reasonable.

Ideally, the FB should be organized as a fully-associative structure. But as constrained by the energy and latency, such a structure with several hundred entries is impractical. So in the proposed architecture, the FB is organized as a set-associative cache. For each tag in FB, besides the conventional bits such as valid bit and replacement information bits, three additional fields are required as depicted

⁷The instructions in Alpha ISA are 4 bytes wide, so the two least bits are omitted.

in Figure 7. The *miss PC (MPC)* is 4-bit XORed PC of the instruction that misses on this line. The *predicted reuse counter (PRC)* is a 2-bit field that keeps the predicted reuse frequency loaded from the predictor. The *reuse counter (RC)* is a 2-bit counter that indicates the current reuse frequency of a line since its insertion.

4.1.3 Shadow Tags

The introduction of FB brings a problem to lines having a long reuse distance. If a frequently reused line is predicted as less reused and placed in the FB incorrectly, and if its reuse distance exceeds the capacity of the FB, it may be impossible to detect the misprediction. Even worse, when the following references to the line occur, the line will always be regarded as a less reused line since its last reuse frequency is always zero, and it will be filtered by FB again and again. Therefore, there is no hit for this frequently reused line.

Since the above problem is caused by the short stack of FB, it can be addressed by augmenting the FB with a shadow tags (ST) [18, 19]. The ST is also a set-associative structure, but it contains only the tags of cache lines. When a cache line is discarded by FB, its tag (including reuse information) will be retained by ST. The extended stack depth of FB yields a better capability of reuse tracking, avoiding most mispredictions to cache lines that have a long reuse distance.

4.1.4 Modification to L2 Cache

In addition to LRF, in order to track the reuse information of the cache lines after they enter into L2 cache, we augment two fields, a 2-bit reuse counter and 4-bit MPC, to each L2 tag. The two fields perform same functions as those in the FB and ST.

4.2 Operation

To serve a request, a lookup is performed in the LRF in parallel with the conventional L2 cache. Since the data in LRF and L2 cache is mutually exclusive, the lookup will result in a hit in one of them but not both, or a miss in both of them.

For clarity, in following discussion, we refer to the line hits in the cache as H , the line misses in both L2 cache and LRF as M . The current PC which causes a L2 cache miss is denoted by CPC . The victim line from L2 cache, chosen by the L2 replacement policy, is denoted by V_{L2} . And the victim line from LRF is denoted by V_f .

Hit in L2 cache or LRF.

If H appears in L2 cache, the access is handled as in the conventional cache except that the reuse counter associated with H is updated. If H appears in LRF, its reuse counter increases, meanwhile the replacement information of LRF is also updated. In this study, we use the LRU replacement policy for LRF.

Miss.

If the request line misses in both the L2 cache and the LRF, it is sent to the memory controller. During the data fetching, three operations should be processed to handle a cache miss in our architecture, as depicted in Figure 8.

First, the reuse information of M is obtained by first searching the shadow tags, then the reuse frequency predictor. If M is found in the shadow tags, its reuse information

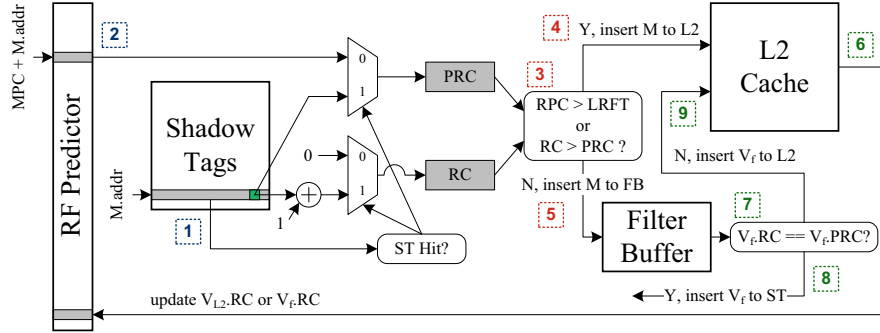


Figure 8: Handling a miss in LRF with three steps: 1) setting up reuse frequency information for the missing line (label 1-2); 2) placing the missing line (label 3-5); 3) dealing with the eviction (label 6-9).

(PRC and RC) and its MPC bits are directly restored from the shadow tags (label 1). Its RC adds one as a result of this reference. If M cannot be tracked by the shadow tags, its PRC is obtained from the predictor using the index formed by concatenating the bits from CPC and M 's line address (label 2), as described in Section 3.1. And its RC is initialized to zero.

Second, based on the reuse information of M learnt in last step, we determine whether M is a less reused cache line, and consequently its placement strategy. If its predicted reuse frequency is greater than the *Less Reused Frequency Threshold (LRFT)* or its actual reuse frequency has already exceeded the predicted one (in the case that M is found in the shadow tags) (label 3), we regard M as a non-less reused cache line and place it in L2 cache (label 4). Otherwise, we regard M as a less reused cache line and insert it into the LRF (label 5).

At last, the evicted line from L2 cache (V_{L2}) or from LRF (V_f) is handled. If M is inserted into L2 cache in last step, then V_{L2} is evicted and V_{L2} 's reuse information is used to update the predictor (label 6). Otherwise, V_f is evicted from LRF. In this case, we check whether V_f 's actual reuse frequency is equal to the predicted one (label 7). If so, V_f is fully utilized. We retire it by inserting its tag and reuse information into the ST (data is written back if dirty) (label 8). If not, that is, V_f is probably to be referenced again or the prediction fails due to the thrashing of reuse frequency, V_f is inserted back to L2 cache (label 9), then another line will be evicted from L2 cache and processed as stated above. In this step, any eviction from ST is written back to the predictor to keep the reuse frequency history up-to-date.

Note that the scope of lines to be filtered in LRF is controlled by the LRFT, thus, LRF can also be used to implement a zero-bypassing scheme by assigning LRFT=0.

Also note that these operations are processed in parallel with fetching data from memory. In a modern processor, it usually costs several hundred cycles to complete a memory visit, so the latency of these operations can be totally hidden.

4.3 Adjustment of the Retirement Threshold

Whether the line V_f evicted from LRF will be inserted back into L2 cache depends on the *Retirement Threshold (RT)*, that is, we retire V_f only if the difference between its reuse frequency and its predicted reuse frequency does not exceed the RT ($|V_f.RC - V_f.PRC| \leq RT$). For example,

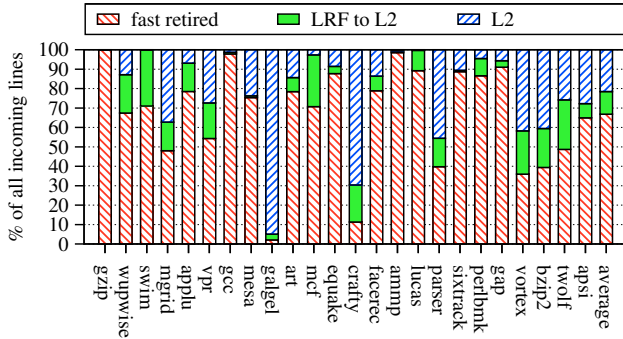
in Section 4.2, $RT = 0$, so if the reuse frequency does not match with the predicted one, V_f will be inserted back to L2 cache. Such a RT is called a *strict* RT. For most workloads, a strict RT works well because it can fully utilize the lines which LRF is unable to retire, and also protect the LRF from mispredictions. However, for some workloads whose working set is much greater than the cache, a *loose* RT (e.g., $RT = 1$) performs better. This is because although inserting these lines into L2 cache guarantees the full utilization for them, discarding them instead can keep a larger portion of working set, which delivers more benefits to these workloads.

To adapt the RT for different workloads, we use *Set Duplicating (SD)* proposed in [8]. The key principle behind SD is that the behavior of all cache sets can be approximated by sampling a small subset of them. In [8], SD mechanism is employed to choose between two replacement policies. In this work, it is used to adjust the value of RT for LRF. In elaboration, we constantly dedicate one group of sets in FB to a strict retirement threshold ($RT = 0$), and constantly dedicate another group of sets to a loose retirement threshold ($RT = 1$). Once one group outperforms the other, its RT value will be used for the remaining sets that belong to neither of them. Similar to [8], a 10-bit counter is used to compare the performance of these two groups. If a miss occurs in the group dedicated to $RT = 0$, the counter adds one; if a miss occurs in the other group, the counter subtracts one. According to the *Most Significant Bit (MSB)* of the counter, we can determine which group has fewer misses and thus better performance. We apply a strict RT if the MSB is 1 and a loose RT otherwise.

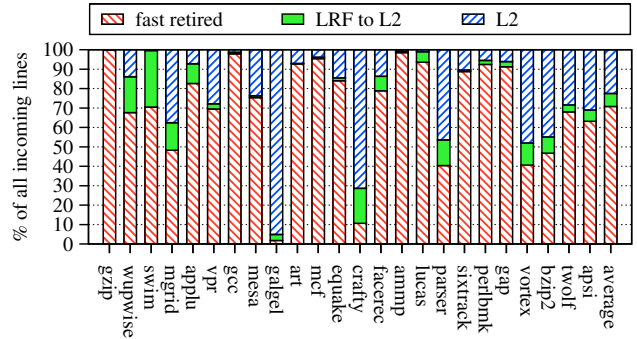
5. EXPERIMENTAL METHODOLOGY

To evaluate the performance of LRF, we use the Alpha binaries for SPEC 2000 benchmark suite available from the SimpleScalar website. For each benchmark, we use the *ref* input set, fast-forward for the first one billion instructions to skip the initialization phase, and execute for two billion instructions to collect the statistics. During the fast-forward phase, we warm up the cache.

The MPKI results are obtained through trace-driven simulation, while the IPC results are obtained through the SimpleScalar 3.0d. Because the SPEC 2000 benchmarks do not stress a very large sized cache, the baseline architecture employs a unified L2 cache at a moderately size of 512KB. However, we also evaluate the performance of LRF under



(a) LRF-sta



(b) LRF-dyn

Figure 9: Less reused filter characterization.

different L2 cache sizes in Section 6.4.1. In the timing simulation, we model a 8-issue out-of-order processor. The overall configuration of the system is summarized in Table 1.

Table 1: System configuration

| Baseline System | |
|--------------------|---|
| Processor | OoO, 8 wide fetch/decode/commit |
| ROB/LSQ | 64/32 entries |
| Branch Pred. | hybrid 32k-entry bimodal + gshare |
| L1 I/D-cache | 16KB, 64B, 2-way, LRU, WB, 2-cycle hit |
| L2 Cache | unified 512 KB, 64B, 16-way, LRU, 8-cycle hit |
| Main memory | infinite size, 400-cycle latency |
| Less Reused Filter | |
| Filter Buffer | 512 entries, 8-way, 64B, LRU, 4-cycle hit |
| Shadow Tags | 768 entries, 12-way, 4-cycle hit |
| Predictor | 64k×2-bit counter, direct-mapped, 1-cycle hit |

Two schemes of LRF are evaluated in this study. Both of them have the same configuration (as illustrated in Table 1) except the retirement threshold. One scheme, namely LRF-sta, uses a strict retirement threshold (i.e. $RT = 0$) constantly, while the other one, namely LRF-dyn, can dynamically adapt the retirement threshold to workloads as described in Section 4.3. In both schemes, $LRFT = 1$, that is, cache lines with predicted reuse frequency less than 2 will be filtered.

6. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the experimental results. We first characterize the behavior of LRF. Then, we present the impact of the LRF on MPKI and overall processor performance. We also give a sensitive analysis for LRF and discuss its storage overhead. Finally, we conduct a comparison of the LRF against several recent related proposals.

6.1 Less Reused Filter Characterization

We characterize the LRF by analyzing the initial and eventual placement of incoming lines. Figure 9(a) plots the destination distribution of all incoming (missing) lines for

LRF-sta. The part “L2” in each column indicates the fraction of incoming lines that are predicted as non-less reused lines and directly placed in L2 cache. The remaining lines are inserted into LRF, and eventually they are either fast retired by LRF (labeled “fast-retired”) or migrate back to L2 cache if cannot be retired by LRF (labeled “LRF to L2”). Since the migration is affected by both the prediction accuracy and the life span, the benchmarks with relatively low prediction accuracy (e.g. *wupwise*) or long life span (e.g. *swim*) usually suffer from a high percentage of migrated lines. On average, 11.6% (up to 28.8% in *wupwise*) of all incoming lines migrate from the LRF to L2 cache in LRF-sta.

The result for LRF-dyn, within which the retirement threshold will be loosened if necessary, is plotted in Figure 9(b). Comparing with Figure 9(a), we see that the benchmarks *vpr*, *art*, *mcf*, *bzip2*, and *twolf* enjoy a loose retiring threshold. Though not fully utilized, most lines in these benchmarks do not migrate to L2 cache. Clearly, the LRF-dyn reduces the overall amount of cache line migration. Only 6.67% incoming lines migrate to L2 cache finally.

These results show that the LRF can effectively retire less reused lines, and its side effect, i.e. line migration, does not have a significant impact on the system bandwidth, especially in the case of LRF-dyn.

6.2 Impact on Miss Rate

In Figure 10, we plot the relative MPKI reduction for two LRF schemes compared to the baseline cache described in Section 5. Because both LRF-sta and LRF-dyn contain a 512-entry filter buffer, to understand the impact of these additional cache lines, we plot the results for other two configurations marked “LRU-17” and “VB-512”. In LRU-17, an extra line is added to each set in baseline configuration, while in VB-512, a 512-entry full-associative victim buffer [20] is deployed to buffer the lines evicted from L2 cache in case they are used again in near future. Note that a full-associative structure with several hundred entries is unrealistic, and it is just used to illustrate the best benefit that a victim structure can provide. Finally, the MPKI reduction by Belady’s OPT replacement policy [2] is also included as the theoretical upper bound.

Comparing two LRF schemes with the baseline, we observe that they improve the MPKI in most cases. Only *wupwise*, *swim* and *gap* suffer more misses, but at most 6%. LRFs reduce the MPKI significantly ($> 10\%$) for 10 out

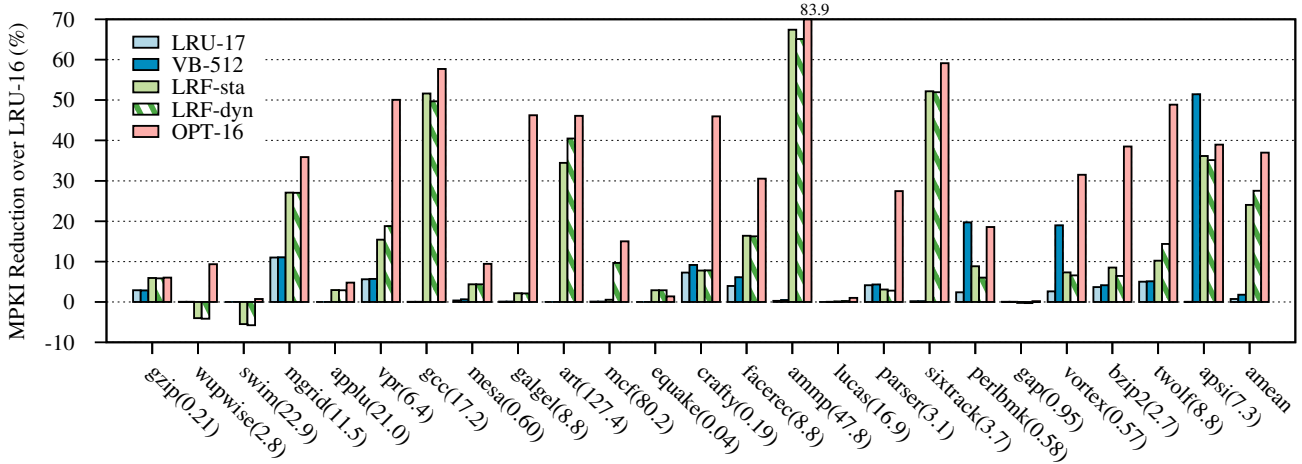


Figure 10: Impact on miss rate. (The number associated with benchmark shows MPKI of baseline.)

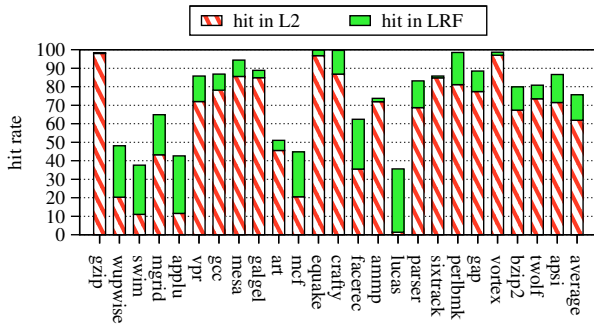


Figure 11: Hit rate distribution in LRF-dyn.

of 24 benchmarks. It is noteworthy that for several high MPKI benchmarks (*gcc*, *art*, *ammp*, *sixtrack*, and *apsi*), the MPKI reduction achieved by LRF-sta or LRF-dyn is quite close to OPT. When all benchmarks are taken into account, LRF-sta and LRF-dyn reduce the average MPKI by 24.1% and 27.5% respectively, narrowing almost 3/4 of the gap between LRU and OPT.

Between two LRF schemes, LRF-dyn clearly outperforms LRF-sta in *mcf*, *art*, and *twolf* because they prefer a loose retirement threshold (i.e. $RT=1$). For those who prefer a strict retirement threshold (i.e. $RT=0$), such as *swim*, *gcc*, *ammp*, *bzip2*, and *apsi*, LRF-dyn is still able to adjust the retirement threshold to 0, resulting in little MPKI degeneration compared to LRF-sta. On average, LRF-dyn performs slightly better than LRF-sta. Moreover, as described in Section 4.3, LRF-dyn only requires line migrations about half as many as LRF-sta. Interestingly, for *art*, *mcf* and *twolf*, although most un-retired lines are discarded (as illustrated in Figure 9), the overall miss rate is obviously reduced. This shows that LRF-dyn can make an optimal choice between ensuring all less reused lines are fully utilized and discarding them to retain a larger portion of working set in L2.

In contrast with LRF, organizing the 512 cache lines as the 17th way or as a full-associative victim buffer does not benefit most high MPKI benchmarks. As an exception, LRF is beaten by VB-512 in *apsi*. This is because of the unbalanced access distribution in the filter buffer for *apsi*.

Figure 11 further plots the local hit rate dedicated to the conventional L2 cache and the LRF and the overall hit rate for each benchmark in LRF-dyn.⁸ From the figure, we can see that a portion of L2 requests is satisfied by the LRF (13.8% on average). In *wupwise*, *swi*, *applu*, *mcf*, and *lucas*, the LRF even provides more hits than the L2 cache.

6.3 Impact on System Performance

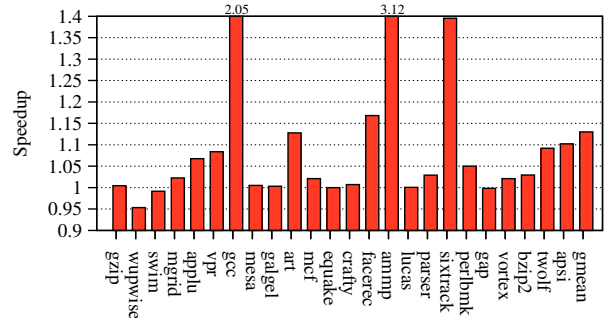


Figure 12: IPC improvement with LRF-dyn.

Figure 12 shows the speedup measured in instructions per cycle (IPC) over the baseline configuration when LRU-dyn is used. The gmean column indicates the geometric mean of IPC speedup in each benchmark. LRU-dyn improves the IPC of benchmarks *gcc*, *art*, *facerec*, *ammp*, *sixtrack*, *apsi* by more than 10%. The performance degradation only occurs in *wupwise* and *swim*. This shows the LRF-dyn performs well for both memory-bound and non-memory-bound benchmarks. Overall, LRF-dyn obtains an average speedup of 12.9%.

6.4 Analysis

6.4.1 Varying the L2 Cache Size

We quantify the impact of LRF-dyn on various L2 cache sizes ranging from 512KB to 4MB. In all configurations, we

⁸In the rest of the paper, we only report the results for LRF-dyn since similar results are observed for LRF-sta.

keep the L2 associativity constantly at 16-way and the filter buffer of LRF-dyn constantly at 512 entries. We use a shadow tags with associativity of 12 for L2 smaller than 1MB, and 32 otherwise.

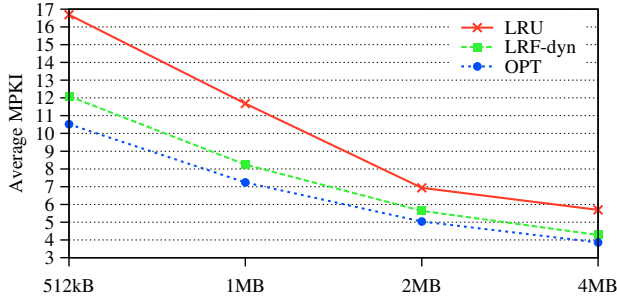


Figure 13: Performance of LRF-dyn on various L2 cache sizes.

Figure 13 plots the average MPKI across all benchmarks for the baseline with and without the LRF-dyn. It can be seen that a LRF-dyn with 512-entry filter buffer improves the performance of baseline significantly. A cache with a LRF-dyn can perform comparably to a double sized cache without LRF-dyn. In the case of 2MB cache, assisted by a LRF-dyn, it even outperforms the 4MB baseline L2 cache. Figure 13 also plots the MPKI achieved by OPT for specific cache sizes. It can be seen that LRF-dyn narrows the gap between LRU and OPT by a large fraction. For 512KB, 1MB, 2MB, and 4MB cache, 74.4%, 77.4%, 68.4%, and 77.2% of the gap between LRU and OPT is narrowed by a LRF-dyn respectively.

6.4.2 Varying the Filter Buffer Size

To evaluate the effects of filter buffer size, we keep the L2 cache size constantly at 512KB, and vary the size of filter buffer among 128, 256, 512, and 1024 entries by adjusting its associativity to 2, 4, 8, and 16 respectively. Meanwhile, the associativity of shadow tags is adjusted to maintain the total associativity of filter buffer and shadow tags at 20 per set.

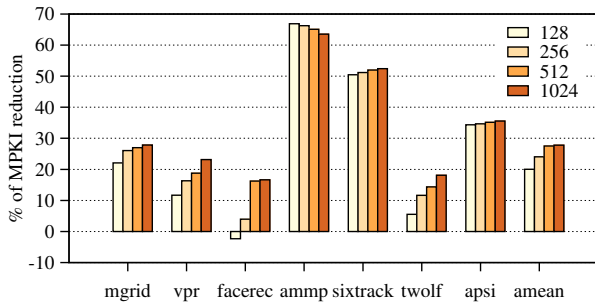


Figure 14: Impact of varying the filter buffer size on the performance of LRF-dyn.

Figure 14 shows the MPKI reduction relative to the baseline configuration for different filter buffer sizes. Several benchmarks are chosen to represent different program behavior. The columns marked with “amean” represent the reduction of average MPKI over all 24 benchmarks.

Comparing Figure 14 with Figure 6, we find that the program behavior is heavily related to the life span distribution of one-reused lines. The filter buffer achieves a considerable performance only when a significant portion of incoming lines have a life span less than the size of filter buffer. For instance, in *facerec*, most lines have life span between 256 and 512, thus a 128-entry or 256-entry filter buffer performs poorly since most lines cannot be retired. In contrast, a 512-entry filter buffer is big enough to retire them (95.2% one-reused lines have life span less than 512), resulting in a good performance. However, since there leaves little room to retire more lines, a larger filter buffer at size of 1024 fails to improve the performance considerably further. On the other hand, in *mgrid*, *sixtrack* and *apsi*, where one-reused lines concentrate at a short life span, even a 128-entry buffer can retire a notable part of incoming lines compared to a larger one, therefore, these benchmarks are almost insensitive to the filter buffer size. Besides, for those benchmarks having low prediction accuracy of reuse frequency, such as *vpr* and *twolf*, a larger filter buffer can tolerate more mispredictions, so the MPKI reduces gradually as the filter buffer grows.

On average, the MPKI reduction is 20.0%, 24.1%, 27.5%, and 27.8% for 128, 256, 512, and 1024 buffer entries respectively. Enlarging the filter buffer from 128 to 256, or from 256 to 512 improves the performance dramatically, but doubling it from 512 to 1024 helps little. These results show that a 512-entry filter buffer is sufficient to obtain most of the benefits, confirming the observations in Section 3.2.

6.4.3 Storage Cost

The hardware required by the LRF consists of the following: (1) the predictor, (2) the filter buffer, (3) the shadow tags, and (4) reuse frequency and MPC bits in each tag-store entry of the L2 cache. The total storage requirement for a 512-entry LRF is calculated in Table 2.⁹ We assume a cache linesize of 64B and a physical address space of 40 bits.

Table 2: Storage cost of a 512-entry LRF

| | | |
|--------------------------------|--|----------------------|
| Filter Buffer | 512 entries, 551 bits per entry | 34.4 KB |
| Shadow Tags | 512 × <i>TDR</i> entries, 36 bits per entry | 2.25 × <i>TDR</i> KB |
| Predictor | 65,536 counters, 2 bits per counter | 16 KB |
| Additional bits in L2 cache | <i>N</i> lines in L2, 6 bits per lines | 0.75 × <i>N</i> B |

In a baseline cache with 512KB data and 32KB tag, where $TDR = 1.5$ and $N = 8196$, the LRF requires about 59.8KB storage overhead, which is less than 11% the area of the baseline cache. Furthermore, as the size of baseline cache increases to 1MB, 2MB, and 4MB, the storage overhead decreases to 6.56%, 3.83%, and 2.47% respectively. Compared to the performance gain, the storage cost paid for LRF is marginal.

6.5 Comparison with Recent Proposals

We compare the performance of LRF-dyn with three recent proposals: V-Way cache [10], dynamic insertion policy (DIP) [8] and shepherd cache (SC) [7].

⁹In Table 2, “TDR” stands for *Tag Depth Ratio*, the associativity ratio of shadow tags to filter buffer.

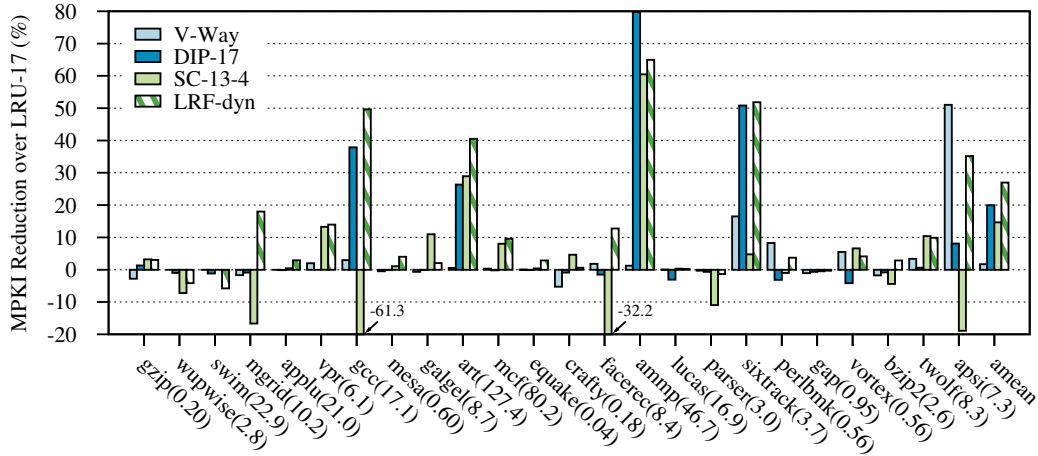


Figure 15: Comparison of the MPKI reduction achieved by V-Way cache, DIP, SC, and LRF-dyn. The baseline configuration is a 544KB 17-way L2 cache managed using LRU.

V-Way cache tries to address the problem that the distribution of memory access is non-uniform across different cache sets in set-associative caches. It decouples the tag-store and the data-store, using a pair of pointers to link a tag and the corresponding data line together. The number of tags per set is doubled, so a cache set provides twice as much associativity as the original one. The data-store is monolithic and is managed by a reuse-based global replacement policy that searches for a victim among all cache lines.

DIP, as mentioned in Section 2.1, is a kind of zero-bypassing schemes. When necessary, the incoming lines are inserted into the LRU instead of MRU position, thus a portion of zero-reused lines can be bypassed.

SC emulates the OPT policy for set-associative L2 cache by logically dividing it into a shepherd cache and a main cache. The former temporarily stores new incoming lines, providing the latter with a future lookahead window for the access order based optimal decision.

Because LRF uses 512 additional lines, in order to make a fair comparison, we add one line per set for these schemes. Thus, all schemes have the same number of data lines, i.e., 8704 lines, or 544KB data size. In V-Way cache, we assume the tag-to-data ratio to 2 (i.e. 34 tags per set) and limit the victim searching distance to five. In DIP, we chose 32 dedicated sets¹⁰ and use an epsilon of 1/32 for the bimodal insertion policy. In SC, the MC-associativity and the SC-associativity are 13 and 4 respectively.

Figure 15 shows the MPKI reduction achieved by above three proposals and LRF-dyn, relative to a 544KB 17-way baseline (LRU-17). V-Way cache aids a limited scope of benchmarks, only clearly outperforming LRF-dyn in *apsi*. SC-13-4 gains notable benefit for some high MPKI benchmarks (*vpr*, *art*, *mcf*, *ammp*, and *twolf*) and provides them comparable MPKI reduction to LRF-dyn. Nevertheless, as limited by the associativity of shepherd cache which is fixed at the design time, the lookahead window provided by SC may be not suitable for some workloads. As a result, SC does not express a good robustness. As shown in the figure, SC-13-4 even causes more than ten percent MPKI degener-

ation for five benchmarks (*mgrid*, *gcc*, *facerec*, *parser* and *apsi*).

Comparing DIP with LRF-dyn, we can see LRF-dyn beats DIP in all benchmarks except *ammp*, *wupwise*, *swim*. Those benchmarks that benefit from DIP also benefit from LRF-dyn. Furthermore, through exploiting more opportunities to retain a larger working set, LRF-dyn usually delivers more benefits to these benchmarks. On the other hand, for those benchmarks in which DIP fails to reduce the miss rate, such as *mgrid*, *vpr*, *mcf* and *facerec*, LRF-dyn still provides a considerable MPKI reduction. In fact, even though we assign LRFT=0 in LRF to implement a zero-bypassing scheme, LRF-dyn is still slightly better than DIP, because the reuse frequency prediction mechanism used by LRF can identify zero-reused lines more accurately.

Overall, the figure shows that LRF-dyn aids the largest scale of benchmarks and provides the best performance improvement among these proposals. Relative to LRU-17, LRF-dyn reduces the average MPKI by 26.9%, while the percentage in V-Way, DIP-17 and SC-13-4 are 1.7%, 19.9% and 14.6% respectively.

In addition, we also evaluate these proposals in 1MB, 2MB and 4MB L2 caches. For a considerable number of benchmarks, the potential improvement (the gap between LRU and OPT) in larger L2 caches is indeed small, so all proposals perform similarly. However, for some high MPKI benchmarks, like *art* and *mcf*, LRF still provides a good performance. In a baseline with 2MB L2 cache, for example, the average MPKIs across all benchmarks achieved by SC-13-4, DIP-17, LRF-16 and OPT-17 are 6.15, 6.10, 5.61 and 4.91.

7. RELATED WORK

Sophisticated replacement policies.

The poor performance of LRU policy for low locality workload has motivated many studies to enhance it by utilizing either the reference frequency [21, 3, 22] or the history information [11, 23]. In addition to these variations of LRU, the IGDR [4] makes a replacement decision depending on the reference interval distributions of cache lines. Rajan and

¹⁰We chose 32 dedicated sets because it performs better than 16 or 64 dedicated sets for most benchmarks.

Ramaswamy [7] recently proposed using a shepherd cache to emulate the OPT replacement. In particular, some policies, so-called *early eviction*, evict the lines as soon as they are dead. The death lines can be identified by the last touch prediction [15, 17], lifetime prediction [14, 24, 25], and locality profiling [6].

On the other hand, based on the observation that some replacement policies have better performance than others for different workloads, hybrid replacement schemes such as SBAR [26] and AC [5] have been proposed to dynamically adapt between two policies. But as shown in [8], even the best performing hybrid replacement cannot provide as much benefit as a zero-bypassing scheme such as DIP.

Because the filter mechanism used by LRF requires *no* modification to the L2 cache replacement policy, some of these policies can be integrated into our architecture to obtain further improvement.

Cache bypassing.

Cache bypassing schemes bypass lines with poor locality or filter them into another buffer in case of the cache pollution. Our work falls into this category. Although some static bypassing schemes are investigated [27, 28], run-time bypassing schemes are more efficient since they can dynamically adapt to program behaviours. González et al. [9] propose the *dual data cache* which consists of a *spatial cache* and a *temporal cache* for vector processors. The incoming lines with a large stride will be placed in the *temporal cache* or bypassed based on their stride size. Rivers and Davidson [29] propose the *Non-Temporal Streaming (NTS)* cache to aid the direct-mapped cache. The NTS cache identifies non-temporal lines and places them into a full-associative NT buffer instead of main cache. Johnson et al. [30, 31] divide the memory address into macro-blocks and maintain their accumulative access count in the *memory access table*. The less frequently accessed lines, which are determined by the reference count of corresponding macro-block, are filtered into a bypass buffer on conflict misses. Etsion et al. [32] achieve the similar goal through a random sampling based probabilistic filtering mechanism.

However, most existing bypassing schemes are targeted at the small L1 cache [29, 32, 33, 34, 35] or reducing conflict misses in low-associativity cache [30, 31]. On the contrary, our work focuses on the large-sized, high-associativity L2 cache which tremendously influences the overall system performance.

Although [33, 35, 34] also exploit the prediction mechanisms to enable bypassing, they only predict whether a line will be used and only work for L1 cache. This paper investigates a practical reuse prediction mechanism for L2 cache, which predicts not only whether a line will be used but also how many times it will be used for.

Alternative cache structures.

The indirect index cache [36] is proposed to achieve full-associativity through software management. The similar goal can be more effectively reached by V-Way cache [10]. Recently, Basu et al. [1] propose the Scavenger which divides the total cache storage into two equal sized parts, a conventional set-associative cache and a large victim file (VF). The VF identifies and retains the cache lines that are most frequency missed in the cache for they are more likely to be used in the future.

8. CONCLUSIONS

The traditional LRU-managed L2 cache performs poorly for workloads whose working set is larger than L2 cache. In this case, the performance of LRU policy can be significantly improved by retaining a shrunken working set in cache long enough. This paper proposes the *less reused filter*, a separate structure that precedes L2 cache, to achieve this goal. The major contributions of this paper are as follows:

- We propose a prediction mechanism to accurately predict the reuse frequency of L2 cache lines.
- We propose a reuse frequency based filtering mechanism for large-sized, high-associativity L2 cache. It extends the filtering scope by filtering the less reused lines instead of just never reused lines, overcoming the drawback of previous bypassing schemes.
- We propose to use a *filter buffer (FB)* to efficiently utilize filtered less reused lines, based on our observation that *less reused lines have a short life span*. Placed in a small FB, most of these filtered lines are fully utilized and do not cause additional misses. We further enhance the performance of FB by dynamically adapting its retirement threshold for different workloads.

The results of this paper indicate that the major limitation of current L2 cache is that the working set of applications cannot fit into the cache rather than the lack of appropriate replacement policies. By carefully filtering out some less reused lines and well handling them, even the simple LRU replacement policy can provide a considerable performance.

There are several possible future directions for this work. The LRF could be extended to a multi-core environment to help the performance of last-level cache where the problem revealed by this paper also exists. Besides, our reuse frequency prediction mechanism could possibly be modified to implement an early eviction policy as well. Finally, we intend to benefit the less reused filter with the help of compiler in our future work.

9. ACKNOWLEDGMENTS

We thank Jiangwei Huang and Liangliang Tong for their helpful discussions during the development of this work, and the anonymous reviewers for their comments and suggestions on the paper. We also thank Man Cao, Yufeng Shen, and Chunhao Wang for improving the writing of the paper. This work was supported by NSFC 60673149 and 863 Program 2007AA01Z105.

10. REFERENCES

- [1] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *MICRO-40*, 2007.
- [2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, pages 78–101, 1966.
- [3] H. Dybdahl, P. Stenström, and L. Natvig. An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. In *MEDEA '06: Proceedings of the 2006 workshop on MEMory performance*, 2006.

- [4] M. Takagi and K. Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *ICS-18*, 2004.
- [5] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *MICRO-39*, 2006.
- [6] W. A. Wong and J.-L. Baer. Modified lru policies for improving second-level cache behavior. In *HPCA-6*, 2000.
- [7] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO-40*, 2007.
- [8] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.
- [9] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS-9*, 1995.
- [10] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: Demand based associativity via global replacement. In *ISCA-32*, 2005.
- [11] Y. Smaragdakis, S. Kaplan, and P. Wilson. The eelru adaptive replacement algorithm. *Perform. Eval.*, 53(2):93–123, 2003.
- [12] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, 1999.
- [13] B. Calder and D. Grunwald. Next cache line and set prediction. In *ISCA-22*, 1995.
- [14] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *ISCA-02*, 2002.
- [15] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA-28*, 2001.
- [16] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. In *HPCA-12*, 2006.
- [17] W. Lin and S. Reinhardt. Predicting last-touch references under optimal replacement. *Technical Report CSE-TR-447-02*, University of Michigan, 2002.
- [18] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD '93*, 1993.
- [19] T. R. Puzak. *Analysis of cache replacement-algorithms*. Ph.D. thesis, 1985.
- [20] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.
- [21] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, 2001.
- [22] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008.
- [23] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.
- [24] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, 2005.
- [25] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the 2005 international Conference on Computer Design*, 2005.
- [26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ISCA-33*, 2006.
- [27] C.-H. Chi and H. Dietz. Improving cache performance by selective cache bypass. *System Sciences, 1989. Vol. I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, 1:277–285 vol.1, 1989.
- [28] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.
- [29] J. Rivers and E. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP '96*, 1996.
- [30] T. L. Johnson and W. mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA-24*, 1997.
- [31] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, 1999.
- [32] Y. Etsion and D. G. Feitelson. L1 cache filtering through random selection of memory references. In *PACT-16*, 2007.
- [33] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Trans. Comput.*, 48(11):1244–1259, 1999.
- [34] J. Jalminger and P. Stenstrom. A novel approach to cache block reuse predictions. In *ICPP '03*, 2003.
- [35] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA-25*, 1998.
- [36] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual international Symposium on Computer Architecture*, 2000.