Less is More: Leveraging Belady's Algorithm with Demand-based Learning

Jiajun Wang, Lu Zhang, Reena Panda and Lizy Kurian John
The University of Texas at Austin
{*jiajunwang, zhanglu, reena.panda*}*@utexas.edu, ljohn@ece.utexas.edu*

***Abstract*—The increasing gap between fast data processing speed and long access latency has become the bottleneck of memory intensive workloads. One way to reduce average load to use latency is to minimize cache misses. While cold misses can be eliminated via prefetching schemes, conflict misses cannot be avoided without an efficient replacement policy. Given future cache access sequences, Belady's algorithm decides whether a data block should get bypassed or cached and leads to the maximum number of cache hits in the most optimal scenario. However, Belady's algorithm does not distinguish demand accesses such as loads and stores, which may cause pipeline stall due to data dependency, from writeback or prefetch accesses. Maximum number of cache hits does not guarantee highest performance. Therefore, we propose a high performance cache replacement policy which leverages key idea of Belady's algorithm but focuses on demand accesses that have direct impact on system performance.**

I. Introduction

Computer architects place caches between computation units and memory in order to leverage data temporal locality and reduce data access latency. Since cache capacity is still much smaller than application's working set size, cacheline replacement cannot be avoided. The most efficient caching algorithm would be to always discard data blocks that will not be needed for the longest time in the future and keep cache blocks that will be reused in near future. Belady's algorithm [1] gives the most optimal case of cache behavior with the knowledge of future. Although it is impractical to implement Belady's algorithm in hardware, prior work [2] has applied Belady's algorithm based on history information and shown that a memory instruction maintains similar optimal cache behavior through its lifetime, i.e., if Belady's algorithm decides cachelines touched by a memory instruction to be cache friendly based on history information, future data accesses from this instruction are highly likely to be categorized as cache friendly as well.

Although Belady's algorithm guarantees the best cache performance (i.e., maximum number of cache access hits), we find that it does not necessarily suggests the shortest workload run time. For example, in a L2 cache thrashing case shown below, where the number of reused cacheline addresses is larger than cache associativity, Belady's algorithm selects caching address (A,B,C,D) over (B,C,D,E) based on their chronological order. Caching address A or E result in the same number of cache hits, however, average load to reuse latency is shorter when caching E instead of A, as cache misses in writeback A won't affect total load to use latency. Although data access type is not taken into account in Belady's algorithm, it should not be ignored when aiming at improving overall system performance. In this paper, we propose a high performance cache replacement policy, LIME (Less Is MorE), which leverages key idea of Belady's algorithm with additional focus on cache access type. We describe the structure of LIME in section II, compare its performance with LRU in section III, and concludes our paper in section IV.

TABLE I: Cache access example

L2 Access sequence	A	B	C	D	E	A	B	C	D	E
L2 Access type	ST	LD	LD	LD	LD	WB	LD	LD	LD	LD

II. LIME Implementation

Conceptually, LIME adopts Belady's algorithm on demand accesses such as loads and stores, and bypasses Belady's training process for writeback and prefetch accesses. Similar to Belady's algorithm, LIME makes predictions on whether an address should be insert into caches or not, but using history information rather than unavailable future accesses. LIME randomly samples 20 sets and makes prediction based on the cache behavior of the sampled sets. The output of Belady's algorithm is at cacheline granularity. LIME extends its scope to memory instruction level, by associating cacheline behavior with the memory instruction triggering the access. The intuition behind it is that same memory instruction gives similar cache behavior through program execution [2]. We design a set of space efficient classifier to maintain memory instructions which has been categorized. So future accesses from classified instruction does not need to go through Belady's algorithm, but will directly refer to the cache behaivor of the instruction itself. While cache evictions can not be avoided under Belady's algorithm, and writebacks are not allowed to get bypassed in contest simulation infrastructure, LIME leverages the idea of SRRIP [3], and chooses cacheline victims based on Re-Reference Prediction Values (RRPV). Figure 1 shows the overall structure of LIME. Its main components include a Belady trainer adopting Belady's algorithm, a set of PC classifier, and RRPV.

A. Belady Trainer

Belady Trainer leverages Belady's algorithm and determines the best cache behavior for a memory instruction. Given an data reuse with address X, by learning through a history period of past L accesses to the same set, Belady Trainer suggests whether X should have been cached or bypassed in the past in order to achieve the maximum number of cache hits. We pick the history length L to be 8X of cache associativity (e.g., 16), and we sample 20 cache sets due to budget limit. Each sampled set is associated with one Belady Trainer.

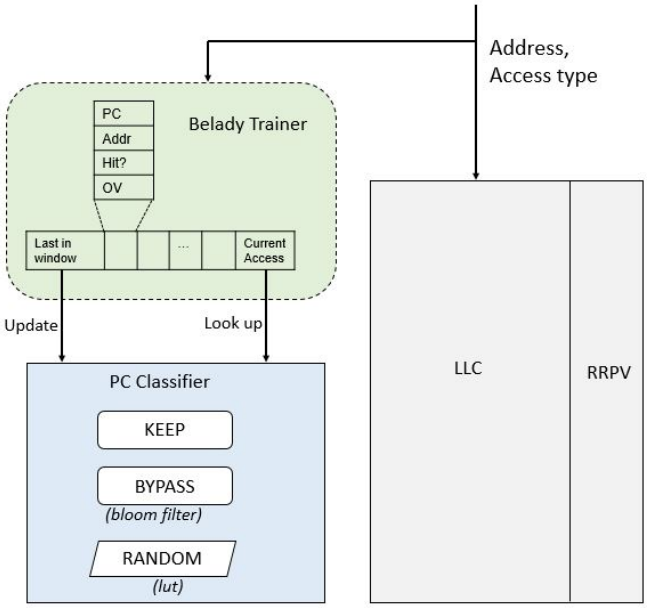

Fig. 1: LIME Block Diagram

Algorithm 1 Find victim in set

Inputs:
AccessType(Load//Store//Prefetch//Writeback),
PCCategory(KEEP//BYPASS//NA)
Outputs:
victim_way
if AccessType == Prefetch OR AccessType == Writeback **then**
$victim_way = 0$
else
while true **do**
Search for first RRPV="maxRRPV" from way0 to way 15
if $found$ **then**
return $victim_way$
else
Aging all lines.
end if
end while
end if

A Belady Trainer is modeled as a First In First Out (FIFO) queue with a size of 128 entries, and each queue entry represents one history access in the associated cache set. One Belady Trainer entry contains four fields: *PC*, *Address*, *Occupancy Vector*, and *Hit*. There are two reasons why PC is kept along with data address. First, since LIME eventually depends on memory instruction category to make replacement decisions, the information of PC is required. Second, Belady's algorithm makes decision until the data is reused, and the previous PC which makes the first data access could be different from the current PC. Since Belady's algorithm makes decision on whether the data should have been cached by the previous PC for future reuse, the previous PC information should be kept. We only keep partial bits of the PC and address due to budget limit. Experiments show that 18 and 37 bits are wide enough to hold information to distinguish different PC and data addresses. We adopt the idea of occupancy vector described in prior work [2]. Occupancy vector reflects the number of occupied ways, and gets incremented every time an access is chosen to be cached. Cache thrashing problem is avoided by ensuring occupancy vector to be always less than set associativity, and the thrashing-causing data gets bypassed when maximum occupancy vector value is equal to the associativity(i.e., all ways are occupied to hold cache friendly data). The field, *Hit*, is a boolean value recording the cache decision made by Belady Trainer. *Hit* is initialized as false, and is set as true until address gets reused and Belady's algorithm decides the reused address to be cached. When the oldest Belady Trainer entry gets pushed out of FIFO, the PC in the entry is categorized as *Keep* or *Bypass* based on Hit is true or not, and the classification is recorded in the PC classifiers. It shoud be noticed that LIME marks a PC as Bypass under two cases, one is when its accessed data never gets reused (e.g., streaming behavior), another is when its accessed data gets reused but refused by Belady's algorithm (e.g., thrashing behavior).

B. PC Classifiers

LIME classifies a memory instruction as three types: Keep, Bypass, or NA (Not Available). When an instruction has not been trained by Belady's algorithm, it is treated as NA. PC classifiers contain three type of bins: KEEP, BYPASS, and RANDOM. The KEEP and BYPASS bins hold PCs that are trained as Keep and Bypass. It is found that accesses from a memory instruction can be determined to be cached by Belady Trainer in one program execution phase and to bypass in another phase. Therefore, such classification altering PC is kept in the RANDOM bin. PCs which have not been trained and remain in NA state are not kept in any of the three bins.

KEEP and BYPASS two bins are implemented using bloom filter data structure. A bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. As LIME is interested only in knowing whether a PC belongs to a specific category or not, such space efficient data structure is the optimal choice considering storage budget limit. The disadvantage is that elements can only be added but not removed. Therefore, for PCs whose category keeps changing by Belady Trainer, their behavior is non-deterministic and they are put into the RANDOM bin additionally. RANDOM bin is implemented as a look-up table (LUT). It records pair information of PC and its latest category, and evicts the oldest pair to make room for new pair. PC behavior is detected as non-deterministic when it is a member of both KEEP and BYPASS bins. If a PC cannot be found in any of the three bins, the PC will be assigned to the NA category. PC Classifiers are indexed in the following ways. PC is searched in three bins in parallel. If PC is found in the RANDOM bin, the category recorded in the according entry is its classification. If a PC is not found in the RANDOM bin, but found in the KEEP or BYPASS bin, it is categorized as Keep or Bypass. If PC is not found in any of three bin, this PC belongs to NA category.

TABLE II: RRPV update policy

PC_Category	Cache Hit	Cache Miss
NA	if($RRPV > 3$) RRPV=3	RRPV=6
KEEP	RRPV=0	RRPV=0, Aging other lines
BYPASS	RRPV=3	RRPV=6

C. RRPV Updating

While Belady Trainer and PC Classifier suggest whether to install an address into cache, LIME leverages the idea of SRRIP [3] to choose replacement victims. Conceptually, each cacheline is associated with 3-bit Re-Reference Prediction Values (RRPV), and victim is selected based on RRPV. RRPV indicates the level of eviction priority, i.e., cacheline with higher RRPV value has a larger chance of getting evicted than cacheline with lower RRPV value. An RRPV of zero implies that a cache block is predicted to be re-referenced in the near-immediate future while RRPV with saturation value (e.g., 7) implies that a cache block is predicted to be re-referenced in the distant future. Table II illustrates RRPV update policy of LIME and Algorithm 1 demonstrates how to select victim in LIME.

D. Handling Prefetch and Writeback Accesses

An efficient replacement policy helps improve system performance by keeping reused data in cache and hence increases cache hits and eventually reduces average data load-to-use latency. However, for accesses like writeback, whose data is written back to memory, and prefetch, whose address may be mis-predicted and not going to be used by execution unit, increasing cache hits of such accesses has little impact on reducing program runtime. LIME redefines data reuse as data is re-accessed because of program demand requests, but not due to writeback or prefetch (unless this is a useful prefetch request). Therefore, our work argues that prefetch and writeback accesses should not be the focus of cache replacement policy. Moreover, considering the large memory footprint of modern workloads, capacity and conflict misses occur frequently, a high performance cache replacement policy should keep data which is going to be reused by core in near future.

LIME dose not apply prefetch or writeback accesses for training, but still choose to put them in cache rather than bypassing. Given the fact that the prefetch accuracy and prefetch coverage of next-line prefetchers are usually low, taking next-line prefetch requests into account will negatively impact LIME accuracy. PC-based prefetcher is more accurate compared to next line, and tend to generate more useful prefetch requests. Thus, bypassing all prefetch requests is not a wise decision because a proportion of prefetched line will be used in future, and they should be kept in cache. Writeback accesses are not allowed to bypass in our framework, and due to write allocate policy, writeback misses trigger cacheline evictions. To address the challenge that, on the one hand LIME does not involve prefetch or writeback in training, while on the other hand LIME can not be bypass those accesses due to performance and infrastructure reason, LIME chooses to limit the cache allocation of these two type of accesses into one way, which is always way 0 in our design.

TABLE III: Multicore benchmarks set

mix 0	bzip2, gcc, astar, GemsFDTD
mix 1	mcf, leslie3d, sphinx3, tonto
mix 2	omnetpp, soplex, libquantum, lbm]
mix 3	bzip2, bwaves, astar, lbm
mix 4	mcf, zeusmp, sphinx3, libquantum
mix 5	omnetpp, gromacs, soplex, gcc
mix 6	gcc, bwaves, GemsFDTD, mcf
mix 7	leslie3d, zeusmp, tonto, libquantum
mix 8	soplex, gromacs, lbm, mcf

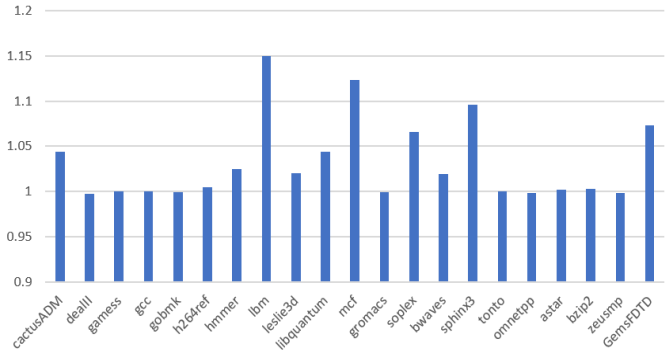

Fig. 2: Performance speedup of configuration 1

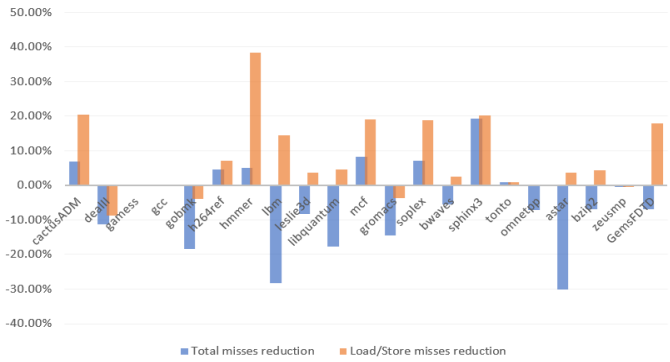

Fig. 3: MPKI reduction of configuration 1

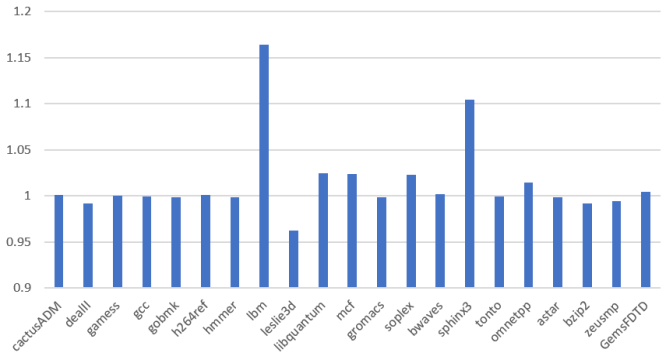

Fig. 4: Performance speedup of configuration 2

TABLE IV: LIME Storage Budget

	Single core	Multicore
Belady Trainer	20 trainers, history length 128	80 trainers, history length 128
PC	20 * 128 * 18 = 5085 B	80 * 128 * 18 = 23040 B
Tag	20 * 128 * 37 = 11840 B	80 * 128 * 37 = 47360 B
Occupancy Vector	20 * 128 * 4 = 1280 B	80 * 128 * 4 = 5120 B
Hit	20 * 128 * 1 = 320 B	80 * 128 * 1 = 1280B
Total	18.09 KB	72.36KB
PC Classifier		
KEEP	512 B	2048 B
BYPASS	512 B	2048 B
RANDOM	152 B	608 B
Total	1.15 KB	4.59 KB
RRPV	2048*16*3 = 12 KB	8192*16*3 = 48 KB
Total	31.2 KB	125 KB

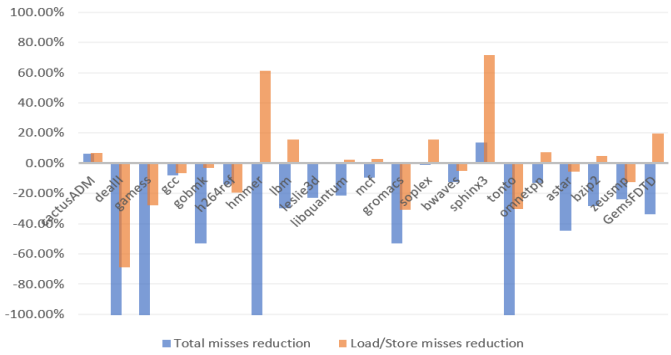

Fig. 5: MPKI reduction of configuration 2

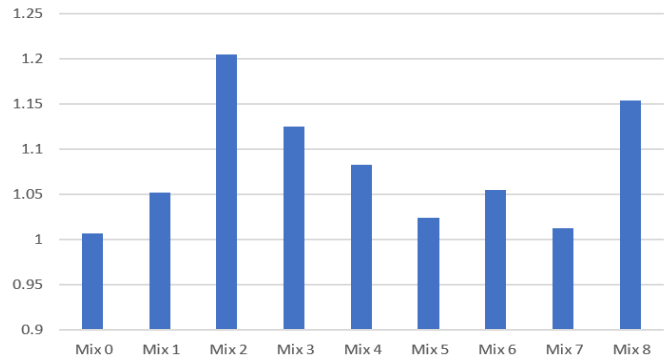

Fig. 6: Performance speedup of configuration 3

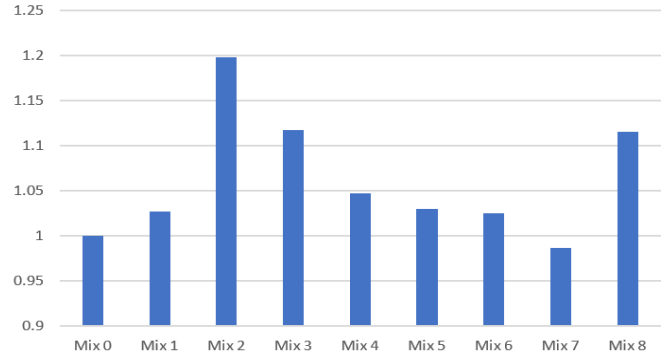

Fig. 7: Performance speedup of configuration 4

III. Evaluation

Result. The performance of LIME are evaluated with four different configurations: configuration 1 has one core, 2MB LLC and no prefetcher, configuration 2 is similar but with prefetcher; both configuration 3 and 4 have four cores and 8MB LLC. Configuration 4 has prefetcher while configuration 3 doesn't. The IPC and miss rate (both total misses and load/store misses) of LIME are compared against LRU. Figure 2 and figure 3 plots the speedup as well as misses reduction with configuration 1 and figure 4 and figure 5 plot the same for configuration 2. For multicore configurations 3 and 4, figure 6 and figure 7 show the speedup against LRU separately.

Budget. LIME makes use of 31.2KB storage with single-core configuration and a total of 35KB storage with 4-core configuration. The detailed breakdown of the usage is listed in table IV.

IV. Conclusion

LIME last-level cache replacement policy is introduced in this report. It leverages the observation that there's a strong correlation between the past and future access patterns with the same PC. Therefore, Belady trainer is used to predict the future accesses based on the near history. Belady's algorithm is applied in order to determine the best cache replacement behavior when selecting victims.

Moreover, LIME respects the observation that load/store misses are more likely to cause pipeline stall than writeback and prefetch misses. In our implementation, when incoming cache accesses are writeback or prefetching, their history won't be kept, i.e., the Belady trainer will not be updated. Our study shows that significant IPC improvement can be achieved with LIME, even with increasing total misses in some cases.

References

[1] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[2] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 78–89.

[3] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815971