LIRS2: An Improved LIRS Replacement Algorithm

Chen Zhong
University of Texas at Arlington
Arlington, TX
chen.zhong@mavs.uta.edu

Xingsheng Zhao
University of Texas at Arlington
Arlington, TX
xingsheng.zhao@mavs.uta.edu

Song Jiang
University of Texas at Arlington
Arlington, TX
song.jiang@uta.edu

ABSTRACT

A block replacement algorithm keeps receiving attention on improvement of its hit ratio. Many replacement algorithms have been proposed, among which LIRS stands out with its consistently higher hit ratio across various workloads with low time and space overheads. However, there are still access patterns where LIRS produces sub-optimal hit ratio and has room for further improvement.

In this paper, we replace the locality measure used by LIRS, the reuse distance, with a more stable and thus more reliable measure, to predict future access time. The new measure is the sum of a block's two recent consecutive reuse distances. It addresses the issue with the reuse distance, which is more likely to fluctuate and mislead replacement decisions. We proposed LIRS2 by incorporating this new measure into the LIRS algorithm to reduce its miss ratio. We further propose a replacement design that allows a responsive adaptation between LIRS2 and LRU so that LRU-friendly access patterns can be well accommodated. We have implemented LIRS2 and its adaptive variant. With extensive experiments on traces from different sources, we show that the algorithms can consistently improve cache miss ratio with low overheads.

CCS CONCEPTS

• **Information systems → Block / page strategies**.

KEYWORDS

Storage system, cache, replacement algorithm

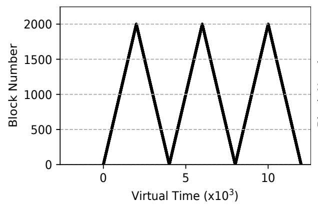
ACM Reference Format:
Chen Zhong, Xingsheng Zhao, and Song Jiang. 2021. LIRS2: An Improved LIRS Replacement Algorithm. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463772

1 INTRODUCTION

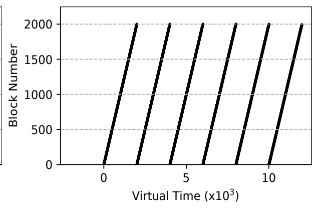
A block replacement algorithm is one of the core components in a memory/storage hierarchy. A block can be a cacheline in a CPU cache, a disk block in an SSD or hard disk, or a unit of data for caching in a distributed file system. Importance of such algorithms is beyond doubt. The algorithm plays a critical role in the efforts of making effective access speed of the hierarchy close to the fastest level's speed. Over the years there have been numerous block replacement algorithms proposed attempting to make the best replacement decisions. Here the "best" block for replacement is defined as the currently cached block whose next access will occur farthest in the future. The distance between the current time and a block's next access time is the locality measure used by the OPT replacement algorithm [1] to decide its replacement block. OPT is provably the optimal algorithm producing the lowest miss ratio. However, OPT is an offline algorithm, whose best replacement choices are not available online.

A basic function of any online replacement algorithm is to collect and interpret history access information and predict future access behavior accordingly. While decision-making of replacement algorithms relies on the existence of locality (the tendency of the same set of data to be repetitively accessed over a short period of time), the interpretation of history access is the process of quantifying strength of the locality. Specifically, the strength measures how repetitively a block is accessed. While everyone agrees that blocks of high locality strength, aka hot blocks, should stay in the cache and the one with the lowest strength should be replaced, the key difference between different algorithms is on the measure they use to quantify the strength.

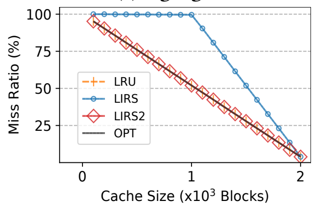
The OPT algorithm uses the time distance between the current time and a block's next access time, named next time distance and denoted T_{next} in Figure 2, which delineates a specific block's access timings in a time line, to measure the locality strength. While OPT's locality strength measure cannot be available online, other algorithms define their measures based on history access information. For example, LRU uses the time distance from a block's last access time to the current time, named last time distance and denoted T_{last}, to measure the strength. LFU uses count of recent accesses (access frequency). To take the frequency into account without carrying too-old history access, an algorithm may adopt



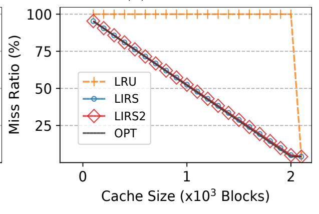
(a) **zigzag**



(b) **scan**



(c) **zigzag**



(d) **scan**

Figure 1: **Accessed blocks at different times for zigzag and scan patterns ((a) and (b), respectively), and their respective miss ratio curves ((c) and (d)). LIRS2 is the algorithm proposed in the paper.**

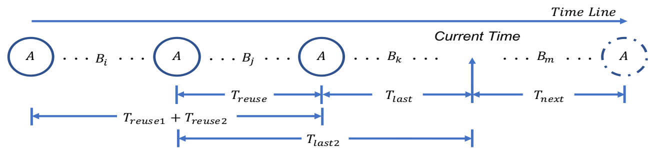


Figure 2: **Example time line showing locality measures for Block A.**

reuse distance, which is the time period between a block's second-to-last access and its last access, denoted T_{reuse}, to measure the strength. Some algorithms consider multiple measures. For example, LRFU [18] uses a weighted combination of the last time distance (recency) and access frequency. ARC [17] integrates the two measures without actual computation for higher efficiency. LRU-2 [23] uses the sum of T_{reuse} and T_{last}, or the distance from a block's second-to-last-access time to the current time, denoted T_{last2}.

Choice of strength measure has a direct impact on the cache miss ratio. For example, let us assume a block is regularly accessed at a fixed interval, or the (next) reuse distance ($T_{reuse} = T_{last} + T_{next}$) is constant. OPT uses T_{next} to make the optimal replacement decision. LRU uses T_{last} to estimate T_{next}. Because T_{reuse} is a constant, the estimation, and thus the prediction, is always wrong. The most recent accessed block (with small T_{last}) is actually the farthest to be re-accessed (with large T_{next}). When the cache is not large enough to hold all the blocks, LRU has a 100% miss ratio. The access pattern is demonstrated in Figure 1b and the miss ratio is shown in Figure 1d. For this particular access pattern, LRU-2 performs similarly as LRU. Algorithms that consider access frequency, such as LFU and LRFU, are able to perform well on the access pattern. However, such algorithms can be too expensive as they need to track every block's access frequency and cannot quickly respond to access pattern change. A milestone development in the efforts of accurately measuring locality strength is the LIRS algorithm [10], which uses the larger of reuse distance and the last time distance ($max(T_{reuse}, T_{last})$), denoted as T_{lirs}, to measure locality. For the aforementioned access pattern, because T_{reuse} is always not smaller than T_{last}, T_{lirs} is constant and every block has the same T_{lirs}. Once a block is selected as a hot one, other blocks will not have a smaller T_{lirs} to make it become a (relatively) cold block. The hot blocks then stay in the cache producing access hits. And its miss ratio is inversely proportional to the cache size, much better than LRU, as shown in Figure 1d. Many studies have found that for most workloads LIRS's performance is significantly better than other state-of-the-art algorithms [15, 17, 32].

In the meantime, we identify access patterns where LIRS has its critical weakness or has room for improvement. One illustrative example is a zigzag access pattern, where a file or data set is repeatedly accessed in an alternative order, such as reading from the file head to its tail and then reversing the order. As shown in Figure 1c, LRU has miss ratios much lower than LIRS's. The reason is that for most blocks a small T_{reuse} is always followed with a large T_{reuse}; and a larger one is always followed with a small one. Accordingly, LIRS's locality measure (T_{lirs}) keeps fluctuating and predicts wrong next access time. While the exact zigzag access behavior may not be often observed, the fluctuation of reuse distance is common. We use the zigzag pattern to highlight the issue. The fluctuation and its impact on miss ratio in real-world traces will be presented in Section 3 (e.g., in Figure 5). Because of high miss penalties in many systems, even moderate improvement of hit ratio is meaningful. For example, a study on Facebook's Memcached system indicates that an increase of the cache's hit ratio by 4.1% "represents over 120 million GET requests per day per server" that would be sent to the backend system and are now removed [2].

To address this issue, we propose a new locality measure, which is to use two recent reuse distances to replace the single reuse distance used in T_{lirs} and denoted T_{lirs2}. That is, $T_{lirs2} = max(T_{reuse1} + T_{reuse2}, T_{reuse1} + T_{last})$, where T_{reuse1} and T_{reuse2} refer to the most recent reuse distance and second-to-the-most-recent reuse distance, respectively. By considering one more reuse distance, it smooths out momentary variation of reuse distance. Reuse distance is a better locality measure than the last access time (T_{last}) used in LRU, because a block's T_{last} is highly variable (changes with any block access) and does not steadily indicate the block's locality strength. For the same reason, T_{lirs2} can be a better measure than T_{lirs}. For the zigzag access pattern T_{lirs2} is a more consistent locality measure that can effectively reveal true locality strength leading to a lower miss ratio.

2 THE DESIGN

The LIRS2 algorithm is an enhancement of LIRS by replacing the reuse distance in its locality measure with sum of two recent reuse distances. In the design, we need to (1) retain LIRS's features that make it stand out among many competitors, including its time and space efficiency; (2) accurately incorporate the new locality measure into its operations to select replacement blocks; and (3) remove negative impact of the "check and trust" strategy adopted in LIRS/LIRS2 for reliable block categorization according to their locality strength.

2.1 Block Categorization

LIRS and LIRS2 use T_{lirs} or T_{lirs2} as their respective locality measures. These measures are then used to predict a block's next access time. While replacement of the block with the farthest next access time is the best choice, a prediction is used to make replacement decision. In LIRS2, the approach is to compare and categorize blocks according to their T_{lirs2}.

OPT replaces a block whose next access time is max_{opt}. All blocks can be accordingly categorized into two sets: the hot set for blocks whose next access times are earlier than max_{opt}, and the cold set for those whose next access times are not earlier than max_{opt}. In theory, all blocks in the cold set should not be cached, even if they are just read. In practice, a block that is brought into the cache should stay there for at least a minimal amount time before being considered for replacement, regardless of its locality strength. This is named "Correlated Reference Period" for "close-in-time subsequent accesses" to a block [11]. Therefore, a small number of blocks in the cold set may be cached. And the algorithm may choose a block in the cache for replacement.

As an online algorithm does not know the true max_{opt}, it uses its own defined locality measure to estimate it and defines the hot/cold sets accordingly. A block with a smaller locality measurement is likely to be re-accessed in the near future. As an approximation, LIRS2 ranks blocks according to T_{lirs2} values. Blocks of the smallest T_{lirs2} values are in the hot set. If the hot set size is N blocks, the Nth smallest T_{lirs2} value assumes the role of max_{opt} for separating blocks into hot and cold sets. Because we want all blocks in the hot set (the hot blocks) always stay in the cache, we let $N \leq C$, where C is the cache size in blocks. To set aside a small fraction of cache space to hold blocks in the cold set, $N < C$ and $C - N$ is small. $C - N$ blocks of the cache space are allocated to cold blocks. Thus, a majority cold blocks are not cached.

As long as blocks are categorized into the two sets, a simple comparison operation upon each block access suffices to maintain the two sets. When a block (Block A) is accessed, it has a new reuse distance, and has a new T_{lirs2}. Assume max_{lirs2} is the Nth smallest T_{lirs2}, and hot Block B has the T_{lirs2}. It is only necessary to compare the new T_{lirs2} with max_{lirs2}. Assume the former is smaller than or equal to max_{lirs2}. Block A remains as a hot block and there is no change of the sets' membership if Block A was a hot block before the access. Otherwise, Block A is re-categorized (promoted) from a cold block to a hot one, and Block B is demoted into the cold set. In the second case, if Block A is also non-resident in the cache, a miss occurs, and one resident block in the cold set is replaced and becomes non-resident.

Alternatively, we assume the new T_{lirs2} is larger than max_{lirs2}. It can be proved that Block A must be a cold block before its access. Its T_{lirs2} right after the access is $T_{lirs2-after} = max(T_{reuse1-after} + T_{reuse2-after}, T_{reuse1-after} + T_{last-after})$, where $T_{reuse1-after}$, $T_{reuse2-after}$, and $T_{last-after}$ are the block's the most recent, second-to-the-most reuse distances, and the last access time right after the access, respectively. We have $T_{last-after} = 0$, $T_{reuse1-after} = T_{last-before}$, and $T_{reuse2-after} = T_{reuse1-before}$, where $T_{last-before}$ and $T_{reuse1-before}$ are the block's last access time and the most recent reuse distance right before the access, respectively. Therefore, $T_{lirs2-after} = T_{reuse1-before} + T_{last-before}$. The block's T_{lirs2} right before the access $T_{lirs-before} = max(T_{reuse1-before} + T_{reuse2-before}, T_{reuse1-before} + T_{last-before})$, where $T_{reuse2-before}$ is the second-to-the-most-recent reuse time before the access. Therefore, we have $T_{lirs2-after} <= T_{lirs2-before}$. Because $T_{lirs2-after} > max_{lirs2}$, $T_{lirs2-before} > max_{lirs2}$. That is, before the access Block A was a cold block. It remains as a cold block after the access.

With the block categorization, LIRS2 employs a "check and trust" strategy to determine whether a block is unlikely to be accessed soon and should be evicted out of the cache quickly. This "check" takes place at the time when the block is accessed. It is actually a comparison operation between T_{lirs2} and max_{lirs2}. The outcome of the check is a placement of the block into one of the two sets. An untrusted block, which is determined to be unlikely to be accessed soon, is placed in the cold set. Because only a minimal amount of cache space is allocated to the set, a resident block in it will be replaced quickly. This is in sharp contrast with LRU, which allows any block to stay in the cache until its last access becomes far behind even if it will never be accessed again.

2.2 The O(1) Algorithm

Quantitatively measuring of T_{lirs2}, the reuse distances, or the last access time to carry out the comparison is in conflict with the requirement of O(1) time with each block access.

The solution is to set up a queue organized as a linked list. Each recently accessed block has two presences, named *instances* representing the block's two recent accesses, in the queue. Each instance records the block number, a flag indicating if the block is hot/cold, a flag indicating if it represents the block's most recent access (*instance 1*) or its second-to-the-most-recent access (*instance 2*), and a flag indicating if

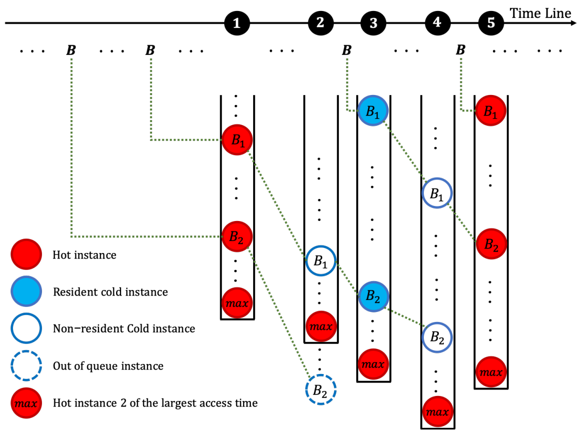


Figure 3: **Illustration of LIRS2's execution on an access sequence focusing on accesses of Block B in the LIRS2 queue. At time ❶, B is a hot block. At time ❷, B's instance 2 leaves queue bottom and B becomes a cold block. At time ❸, B is re-accessed and becomes a (cold) resident block. At time ❹, B is early evicted and becomes a non-resident block due to its cold status. And at time ❺, B is re-accessed and becomes a hot block.**

this block is still in the cache (resident) or has been evicted. When a block is accessed, a new instance 1 is created and pushed into the queue from its head (or at the top position shown in Figure 3). If currently the block has its instance 1 in the list, it changes to instance 2. If the block also has its instance 2 in the queue, remove this history access record.

As shown in Figure 3, the ordered instances in the queue reflect recent block access events with one exception, which is that each block has at most two instances (for two most recent access events) in the queue regardless of how many accesses a block has. The queue is used for tracking the time periods, such as T_{lirs2}, T_{reuse}, and T_{last}. To be precise, these time periods are measured in the number of unique blocks accessed during the period, not necessarily the number of accesses in the period, to reflect the fact that multiple accesses to the same block in a time period do not consume more cache space.

All hot blocks are resident, and their instances are in the queue, named LIRS2 queue. The oldest instance 2 of any hot block is positioned at the tail of the queue (or at the bottom position shown in Figure 3). When this block is re-accessed, and its instance 2 at the bottom is removed, the next instance 2 of a hot block in the queue becomes the new bottom and any instances between the two instance-2s are also removed to maintain the invariant that the bottom instance must be a hot block's instance 2. In this queue, the current time is represented by the queue top, and max_{lirs2} is represented by the queue bottom. For replacement, the resident cold block that has not been accessed for the longest time is replaced. To this end, all resident cold blocks are also maintained in a (short) LRU queue, named CoRe (cold resident) queue.

A key LIRS2 operation is comparison of a block's new T_{lirs2} (at the time when it is accessed) with max_{lirs2} for its categorization. It sounds to be an expensive operation. However, with the LIRS2 queue it becomes surprisingly efficient: if the block's instance 2 is in the queue before the access, it means new $T_{lirs2} \leq max_{lirs2}$ because the bottom instance represents max_{lirs2}. As all operations on the LIRS2 and CoRe queues corresponding to a block access have an O(1) time, LIRS2 is O(1) efficient. Any decision on the set membership (hot or cold) can be made by simply checking if the accessed block exists in a linked list or not. It also takes O(1) time to find the LRU block in the CoRe queue for replacement. LIRS2's operations are illustrated in Figure 3, and its pseudo code is depicted in Algorithm 1.

As the LIRS2 queue contains non-resident blocks, it is possibly much longer than an LRU stack (the cache size). To bound its length, we prune the queue by removing a cold instance that is closest to the bottom instance (max_{lirs2}) from the queue once the queue is longer than a threshold. Similar approach is adopted in LIRS. The threshold is 8 times of the cache size (in terms of blocks).

Algorithm 1: Pseudo code of LIRS2

```
/* During the initiation, any cold block turns into a hot
   block upon its access until there have been N hot blocks,
   where N is the maximal number of hot blocks. */
Upon access of Block B
/* its instance 1 is denoted $B_1$; and its instance 2 is denoted
   $B_2$ */
if $B_2$ is in the LIRS2 queue then
    if B is a cold block then
        Promote it a hot block;
        Demote the hot block currently at the bottom of the LIRS2 queue
          to the CoRe queue;
        Keep removing any instance at the bottom of the LIRS2 queue
          until a hot block's instance 2 is at the bottom.
    Remove $B_2$ from the LIRS2 queue;
else
    B is set as a cold block;
end
if $B_1$ is in the LIRS2 queue then
    Change it to $B_2$ (by updating a flag of the instance)
Insert $B_1$ into the LIRS2 queue at its top;
if B is a non-resident block then // access miss
    Set the bottom block at the CoRe queue as non-resident (the
      replacement block), and remove it from the queue;
    Set B as a resident block;
if B is in the CoRe queue then
    Remove it from the queue;
if B is a cold block then
    Insert it into the CoRe queue at its top;
```

2.3 LIRS2's Optimization

LIRS2 categorizes a block into one of the two sets at the time of its access by checking if it was in the LIRS2 queue, and treats them differently. If a block was not in the queue and is accordingly placed into the cold set, it will be soon replaced because this set has been allocated with very limited cache space. Any resident blocks in it will be replaced quickly if not re-accessed immediately. Otherwise, if it passes the check and is trusted to be a hot block and placed into the hot set, it can safely stay in the cache even without more accesses for a relatively long time period. LIRS2's advantage on hit ratio is mainly attributed to this "check and trust" strategy. However, this strategy has its side effect.

For a block that is accessed for the first time or has not been accessed for a long time to be trusted, it has to first experience three accesses (or three misses) to produce two relatively small reuse distances. For most workloads with access locality, once a block starts to be accessed, it likely stays active for a while with many more accesses, which help amortize the loss of hits during the initial check period. However, there may be workloads with many blocks of very few accesses in an active period. If a block has only two accesses in the period, LRU has one miss and LIRS2 has two misses. If it has only three accesses, LRU still has one miss, but LIRS2 has three misses. That is, the side effect of LIRS2's "check and trust" strategy is avoided in the LRU algorithm. When this side effort is significant, LIRS2 may have a miss ratio higher than LRU, as our experiment results will reveal.

As the strategy is the main source of LIRS2's performance advantage, we cannot remove the side effort by changing LIRS2's core design principle. To this end, we propose an adaptive LIRS2, which opportunistically switches to LRU to remove the side effect once LRU's advantage shows up.

We propose *LIRS2-Adapt*, in which both data structures of LIRS2 (the LIRS2 queue and CoRe queue) and LRU (the LRU stack) [1] are maintained. Essentially, LIRS2-Adapt runs the two algorithms simultaneously, monitors their miss ratios, and follows the replacement decisions from the one with lower ratios in an epoch-by-epoch fashion.

In the design we need to consider the mechanism to enable the switch and the policy to carry out the switch. A straightforward approach for the switch is to move all instances of resident blocks from the LIRS2's queues to the LRU stack for a LIRS2-to-LRU switch or move all blocks in the LRU stack to initialize LIRS2's queue for an LRU-to-LIRS2 switch. While this approach functions, it incurs a time cost higher than O(1) during the switch. To address the issue, LIRS2-Adapt chooses an incremental switch method. It allows hot blocks to be non-resident in LIRS2 and blocks in the LRU stack to be non-resident in LRU during the transition period. These non-resident blocks become resident when they are accessed, or exit from either of the data structures if they are not accessed for a while. Following the accesses LIRS2-Adapt naturally aligns itself with LRU or LIRS2, depending on which algorithm it switches to. In this way, all operations are still of O(1) with every access. The adaptation policy is to monitor difference between two algorithms' miss ratios and decide timing to switch. Likely there is performance loss due to the switch. And frequent back-and-forth switches may hurt more than benefit the performance. To this end, we take a conservative approach in making a switch decision. We divide the block accesses over the time into epochs. Each epoch contains accesses that can fill 20% of the cache. LIRS2-Adapt monitors and compares miss ratios of LIRS2 and LRU during each epoch. If the standby algorithm beats the currently effective one for five continuous epochs by 10 percentage points or more on miss ratio value (e.g., 25% vs. 35%), LIRS2-Adapt starts its switch to the standby one in the next epoch. The default initial algorithm during the adaptive execution is LIRS2.

[1]Note that LIRS2's CoRe queue and the LRU stack in the LRU algorithm are different. In LIRS2, the CoRe queue only stores resident cold blocks. In LRU, its LRU stack stores all resident blocks. In all three queues/stack, block instances are organized in the LRU order.

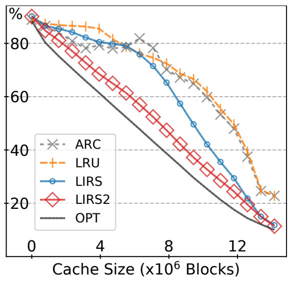
3 EVALUATION

In this section we evaluate and compare performance of the proposed replacement algorithms (LIRS2 and LIRS2-Adapt) with other major algorithms, including LRU, OPT, LIRS, and ARC. ARC is an algorithm that blends recency and frequency in its locality measure by using two queues to separate blocks with only one access and those with multiple accesses. In the evaluation we use 106 week-long virtual disk traces collected by CloudPhysics's caching analytics service in production VMware environments [32] and 4 week-long enterprise server traces collected by Microsoft Research Cambridge [20]. We received the traces from CloudPhysics and use the original trace names in the presentation (*w001, ..., w106, msr_proj, msr_src1,msr_src2,msr_web1*). We also use 5 I/O traces from the UMass Trace Repository (*Financial 1/2* and *WebSearch 1/2/3*) [3]. This is a diverse set of all real-world traces with significant scales and that have been widely used for caching system evaluations [4, 7, 21, 22, 33]. The block size is 16KB.

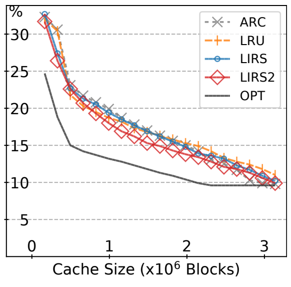
We obtained the miss ratio curves of the algorithms on the 115 traces. To present representative results, we group the traces into four types (A, B, C, and D) according to relative performance of LIRS, LIRS2, and LRU. In Type-A traces, LIRS2 is substantially better than LIRS. In Type-B traces, LIRS2 performs similar to LIRS, but substantially better than LRU. In type-C traces, LIRS, LIRS2, and LRU perform similarly. And in Type-D, LRU performs substantially better LIRS and LIRS2. Out of the 115 traces, there are about 25, 41, 35, and 14 traces that belong to Types A, B, C, and D, respectively. And we choose 7, 5, 4, and 4 traces from each of the four types, respectively, to present their miss ratios in Figure 4.

3.1 Results about Traces of Type A

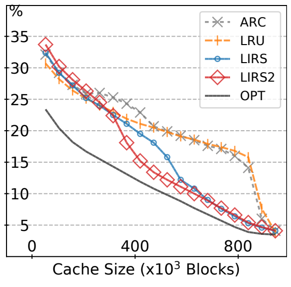
Let's first examine miss ratios of the 7 Type-A traces (Figure 4a-4g). First, the improvements of LIRS2 over LIRS can



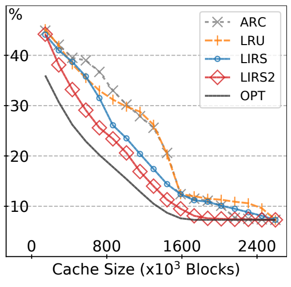
(a) **w021 (Type A)**



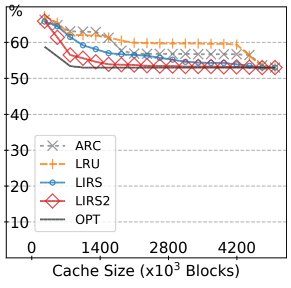
(b) **w032 (Type A)**



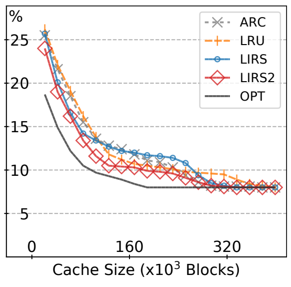
(c) **w038 (Type A)**



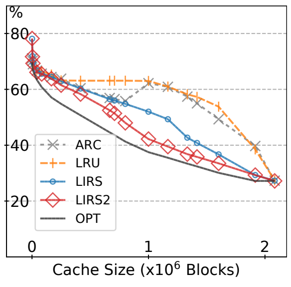
(d) **w049 (Type A)**



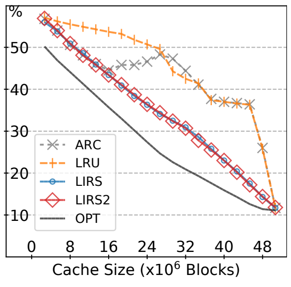
(e) **w086 (Type A)**



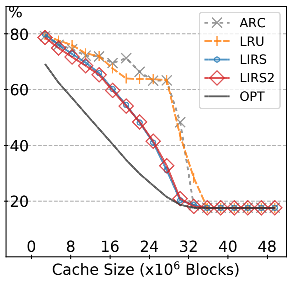
(f) **w103 (Type A)**



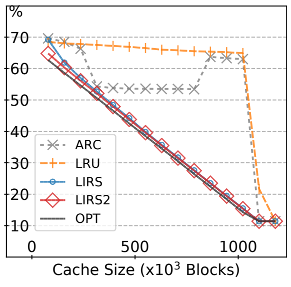
(g) **msr_src2 (Type A)**



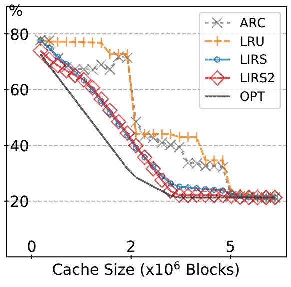
(h) **w004 (Type B)**



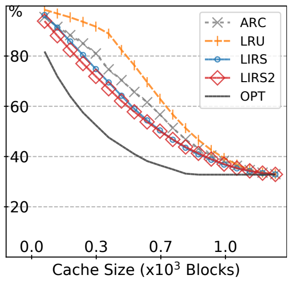
(i) **w020 (Type B)**



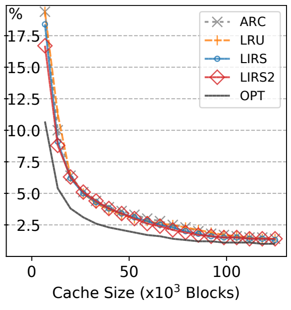
(j) **w099 (Type B)**



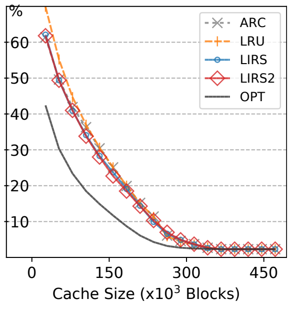
(k) **msr_web1 (Type B)**



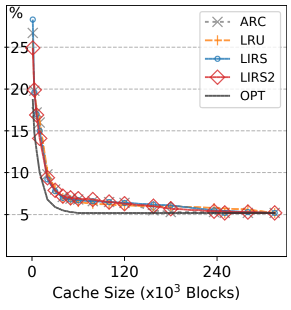
(l) **websearch1 (Type B)**



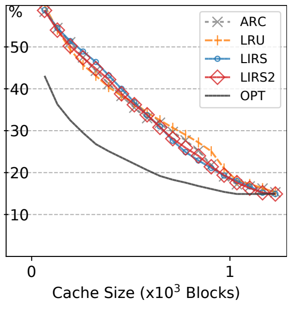
(m) **w006 (Type C)**



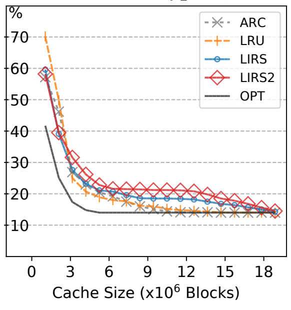
(n) **w059 (Type C)**



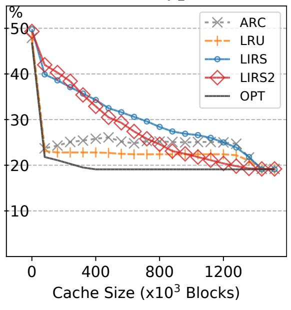
(o) **w106 (Type C)**



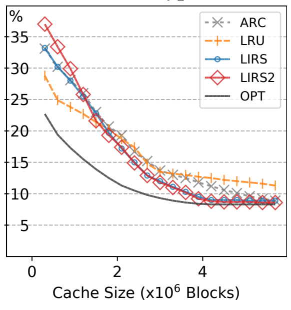
(p) **financial1 (Type C)**



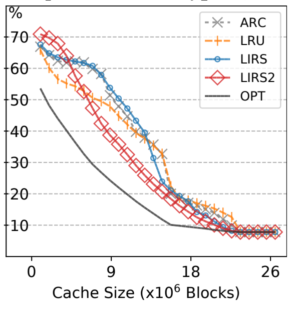
(q) **w046 (Type D)**



(r) **w079 (Type D)**

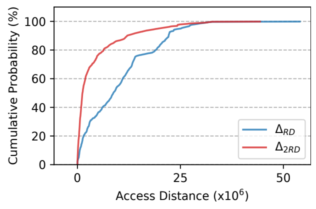


(s) **w013 (Type D)**

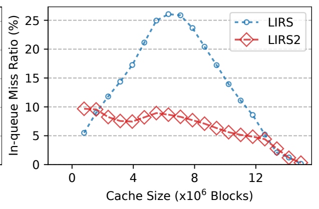


(t) **msr_src1 (Type D)**

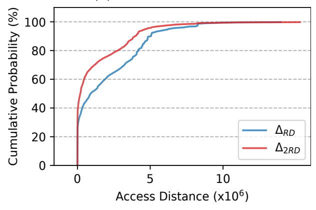
Figure 4: **Miss ratio curves of selected 20 traces belonging to different types.**



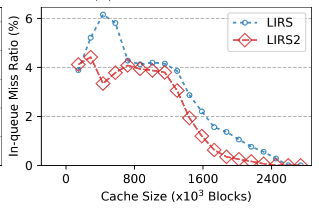
(a) **Trace w021**



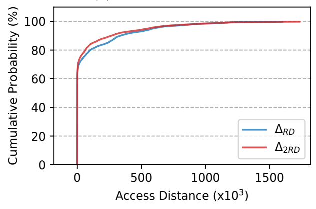
(b) **Trace w021**



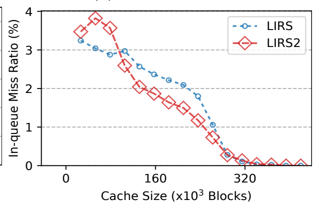
(c) **Trace w049**



(d) **Trace w049**



(e) **Trace w103**



(f) **Trace w103**

Figure 5: **CDF curves for variations of reuse distance and sum of two continuous reuse distances for selected traces ((a), (c), and (e)), and their in-LIRS-queue and in-LIRS2-queue miss ratios ((b), (d), (f)).**

be significant. For example, reductions of miss ratios are as much as 19.1, 5.7, and 9.8 percentage points on traces *w021*, *w038*, and *msr_src2*, respectively. Second, a common observation of miss ratio curve indicative of possible performance issues is stagnant descent, that is, no or slow descent of miss ratio curve with the increase of cache size. Examples are LIRS on the zigzag accesses and LRU on the parallel accesses shown in Figure 1. There are several occurrences of stagnant descent in the LIRS curves, such as one for *w021* at small cache sizes, one for *w038* at medium cache sizes, and one for *msr_src2* at a large range of cache size in the middle, as shown in Figures 4a, 4c, and 4g. LIRS2 eliminates or ameliorates the issue. Third, for traces where LIRS has significantly improved performance over LRU and ARC, *LIRS2* can still further improve its miss ratios, bringing them closer to OPT's. Examples include *w021*, *w049*, and *msr_src2*. Meanwhile, in cases where LIRS performs worse than LRU or ARC, LIRS2 reverses the trend, as shown in Figure 4f for *w103*.

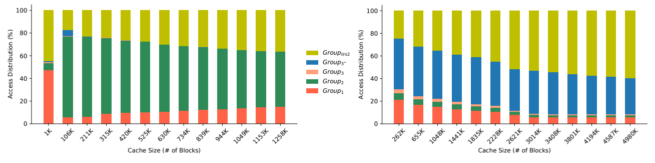
In Section 1, we use a hypothetical zigzag access pattern to illustrate how LIRS may underperform with unstable reuse distance. To show existence of the instability and its impact on replacement algorithms in the real world, we plot variations of reuse distance, denoted RD, and sum of two consecutive reuse distances, denoted 2RD, as CDF (cumulative distribution function) curves in Figures 5a, 5c, and 5e on traces exhibiting different LIRS2's advantage (*w021*, *w049*, and *w103*, respectively). Specifically, Δ_{RD} is the difference between continuous RDs. Δ_{2RD} is the difference between continuous 2RDs. As shown, Δ_{2RD}s are substantially smaller than Δ_{RD}, especially on w021 and w049. Furthermore, their difference roughly correlates with LIRS2's performance improvements. For example, Δ_{2RD}s are much smaller than Δ_{RD} on *w021*. Accordingly, LIRS2 significantly improves LIRS. In contrast, on *w103* both the differences, between Δ_{2RD} and Δ_{RD}, and LIRS2's miss ratio improvement over LIRS are much smaller. This observation confirms existence of LIRS's issue with stability of its locality measure.

To further understand how the (in)stability actually compromises (or helps with) the performance, we measure number of times a block is mis-categorized as a cold block in LIRS and LIRS2. Such a mis-categorization is a mis-prediction leading to a hot block to be placed in the cold set and be replaced quickly. When its next access is a miss and occurs when it is still in the LIRS2 queue, we know that the block has been mis-predicted. Had we known it would be re-accessed at a time earlier than the block at the queue's bottom, we would have kept it in the cache longer by categorizing it as a hot block. The mis-categorization takes place because the algorithm is mis-guided by its last measurement, which is significantly different from the current one. Figures 5b, 5d, and 5f show percentage of accesses (of all accesses) corresponding to misses of cold blocks when they are in the LIRS/LIRS2 queue for the three aforementioned traces. For *w021*, LIRS has much more such misses (or mis-predictions) than LIRS2. This observation is well correlated to the large improvements made by LIRS2 over LIRS. For the other two traces the issue of LIRS for having such misses more than those from LISR2 is less significant. Accordingly, LIRS2's improvements are smaller. Note that prediction of an actually cold block to be a hot block also compromises performance, as a block will stay in the cache longer and wastefully.

3.2 Results about Traces of Types B and C

In the five Type-B traces LRU and often ARC are much worse than LIRS/LIRS2. On these traces LIRS does not show any clear stagnant descents. This indicates that LIRS has effectively exploited the locality, and leaves little room for LIRS2 for further improvement. Therefore, LIRS2 has performance similar to LIRS. However, LRU and ARC show serious stagnant descents on their curves. And LIRS is sufficient to address the issue in these workloads mainly because their reuse distances have been stable to produce credible predictions and it's not necessary to use sum of two reuse distances.

For the four Type-C traces, miss ratio curves of all the algorithms are well overlapped. This happens when a workload has very strong locality. Accesses of a block are highly



(a) **Trace w079** (b) **Trace w013**

Figure 6: **Breakdown of accesses in different groups in two traces.**

clustered, or reuse distances are small (shorter than the cache size). Any of the algorithms can easily keep a block in the cache until it is re-accessed to produce a hit. For such workloads even LRU is sufficient.

3.3 Results about Traces of Type D

Figure 4 shows results for the four Type-D traces, where LIRS/LIRS2 visibly exhibit worse performance than LRU in some ranges of cache size. To understand impact of the LIRS2's check-and-trust strategy on the performance, we breaks down accesses into groups according to their different impacts on LRU/LIRS2's performance. In the grouping, cold access refers to access of a block that has not been accessed for a long time and its history access information, if any, is no longer recorded in the LIRS2 queue. An access sequence of a block is defined as a sequence of its accesses starting from a cold access to the one right before its next cold access or its last access, denoted A_1, A_2, ..., A_k, ..., A_n. Assume the distance between A_{i-1} and A_i is not larger than cache size when $i \leq k$, and the distance between A_k and A_{k+1} is larger than the cache size. Let's first consider the first k accesses. If $k = 1$, both LRU and LIRS2 has a miss. But LRU wastes a cache space for a much longer time. If $n = 2$, LRU has one miss, and LIRS2 has two misses. If $k = 3$, LRU has one miss, and LIRS2 has three misses. When k becomes even larger, additional accesses are all hits for both algorithms. We place the first k accesses (A_1, A_2, ..., and A_k) into $Group_k$ for k = 1, 2, or 3. If $k > 3$, the accesses are placed into $Group_{3+}$. Any accesses between A_{k+1} and A_n are in $Group_{lirs2}$. Accesses in $Group_{lirs2}$ are more likely to be hits in LIRS2 as the block was more likely to be hot than in LRU when the accesses more likely occur out of the LRU stack and are misses.

Figure 6 shows breakdown of the accesses in the five groups for *w079* and *w013*. For *w079*, except for very small cache sizes, there are significant portion of the accesses belonging to $Group_2$ and $Group_3$, giving LRU advantage to achieve more much hits than LIRS2. This explains why LRU has much lower miss ratios for most cache sizes. When the cache size becomes large, accesses in $Group_1$ and $Group_{lirs2}$, which are beneficial to LIRS2, increase, corresponding to improved LIRS2's performance shown in Figure 4r.

In contrast, the breakdown figure for *w013* shows that accesses in $Group_1$, $Group_2$, and $Group_3$ represent a significantly less percentage than those for *w079*. Accesses in $Group_{3+}$ are more likely to favor LRU. And accesses in $Group_{lirs2}$ are more likely to favor LIRS2. Their miss ratios are not significantly different (shown in Figure 4s), indicating LIRS2's check-and-trust's side effect is not serious. With increase of the cache size, the portion of accesses in $Group_{3+}$ increases, and LIRS2's performance improves.

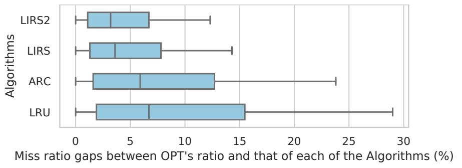


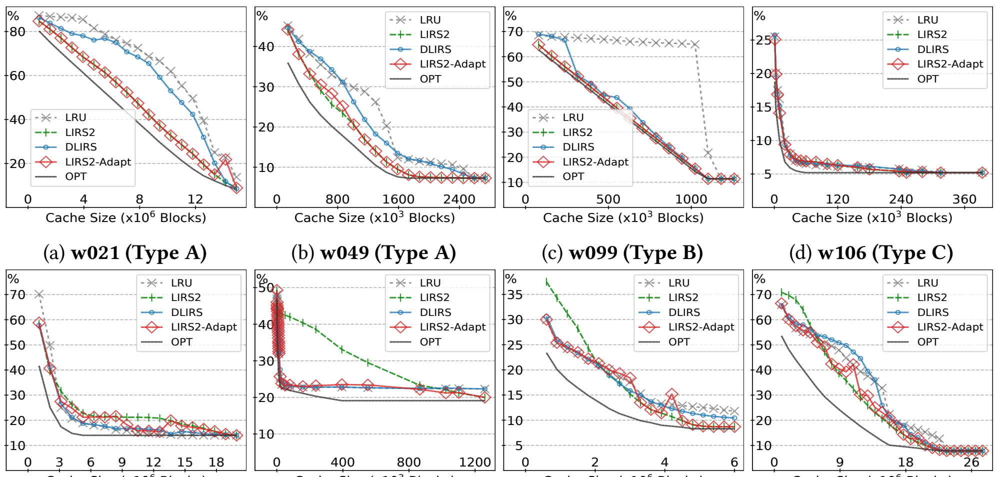
Figure 7: **Miss ratio gaps between OPT's ratio and that of an algorithm under test. For each of the algorithms, 2,300 miss ratios are collected (for each of the 115 traces on 20 cache sizes evenly distributed between 1% of the data set size and the size where the miss ratio starts to plateau out). For each algorithm, a box plot has lines corresponding to first quartile, median, and third quartile. Lines extending from a box indicate minimum and maximum.**

A Bird's-eye View As a summary, the box plots in Figure 7 show how each of the algorithm performs in comparison to OPT on all of the 115 traces and different cache sizes. As shown, the lines for LIRS2's first quartile, median, third quartile, and maximum all indicate the miss ratios closer to OPT's than LIRS. LIRS2 improves LIRS's miss ratio when LIRS's performance issue arises. And often LIRS performs pretty well. Therefore, the statistical comparison doesn't show big improvements upon LIRS. Meanwhile, their improvements over ARC and LRU are much more significant. The miss ratio plot for each of the 115 traces can be found at *https://github.com/zhongch4g/LIRS2*.

3.4 Performance of LIRS2-Adapt

Accesses in $Group_2$ and $Group_3$ expose the weakness of the check-and-trust strategy. This weakness can be effectively addressed by the LRU logic in the LIRS2-Adapt. To observe its effectiveness, we select some traces of Types A, B, and C and all Type-D traces from Figure 4, and show their miss ratio curves in Figure 8. In this experiment, we include DLIRS, an algorithm designed to address the issue by dynamically changing the allocation of cache space between cold and hot sets in LIRS [15]. DLIRS tracks misses of cold in-queue blocks, and proportionally increases the allocation to the cold set to allow untrusted blocks to stay in the cache longer.

Figure 8 shows miss ratios of the algorithms. There are several observations. First, on Type-D workloads, for which LIRS2-Adapt is introduced, LIRS2-Adapt mostly traces the lower of LIRS2 and LRU curves. For example, for *w079* before the cache size reaches about 865K blocks, LIRS2-Adapt performs much like LRU. And after the cache size, it performs as LIRS2 when LIRS2's miss ratios become lower. We do notice that there are a few spots where LIRS2-Adapt does



(a) **w021 (Type A)** (b) **w049 (Type A)** (c) **w099 (Type B)** (d) **w106 (Type C)**

(e) **w046 (Type D)** (f) **w079 (Type D)** (g) **w013 (Type D)** (h) **msr_src1 (Type D)**

Figure 8: **Miss ratio curves for evaluating LIRS2-Adapt.**

not follow the better of the two, as it adopts a conservative adaptation strategy to avoid overaction. This may lead to lost opportunities.

Generally DLIRS performs similarly for Type-D traces, except msr_src1, for which DLIRS is visibly worse. The major issue with DLIRS shows on type-A traces, where its curves are significantly drifted away towards LRU's (see Figures 8a and 8b). LIRS2 out-performs LIRS on these traces, indicating that LIRS encounters highly variable reuse distances. This causes DLIRS to detect many cold-hot block status changes, increasing its cold-block space allocation as its response to hold cold blocks longer in the cache. However, instead of using a different locality measure to accurately identifying cold blocks from hot ones, DLIRS seeks to treat cold blocks more like hot ones, making its performance closer to LRU.

3.5 Overhead Analysis

Both LIRS2 and LIRS2-Adapt are O(1) algorithms. With only a few instructions for updating pointers and block status upon an access, they are as efficient as LRU.

The metadata overhead of LIRS2 and LIRS2-Adapt is moderate and well affordable. As we mentioned, the number of instances in a LIRS2 queue is capped at 8X cache size in terms of blocks. Each instance requires 4 pointers in the linked list. Considering maximally possible number of instances in the queue, we use 4 bytes for each pointer. Each block has up to two instances in the queue. Each instance contains a block number and a few flags occupying 4 bytes. So each block needs up to 40 bytes for the queue. The space overhead for the CoRe queue is negligible as it is very short. In the LRU queue, each block needs only one instance of 20 bytes long. Assuming a 16KB block, the space cost for LIRS2 is up to 1% and 1.1% for LIRS2 and LIRS2-Adapt, respectively. A further reduction of this overhead is possible, as some of the similarly structured algorithms have been implemented in the CPU cache, such as RRIP [9] and Cache Bursts [16].

4 RELATED WORKS

Caching is a fundamental technique that has been applied in numerous scenarios to speed up data access from processor cache, block storage devices, and remote servers in data centers for faster instruction execution, higher I/O performance, or reduced network traffic. Arguably a replacement algorithm is the most performance-critical component in a caching system. While the algorithms may be customized for their applications in different scenarios with features such as ultra-low cost and variable caching object sizes, the fundamental ones are those for using the memory to cache fixed-size blocks on the disks. There are many studies, which are directly related to this work, on these algorithms.

4.1 Block Replacement Algorithms

In the spectrum of the block replacement algorithms, LRU and LFU represent two contrasting design principles. LRU only considers a block's most recent access event to characterize its access pattern, while LFU uses many history accesses attempting to receive a more stable and reliable interpretation of its access behaviors. Each has its drawbacks. LRU is too short-sighted by only considering recency, the last access time. This locality measure is not stable and thus not reliable to predict next access time. In contrast, LFU is not responsive to access pattern change and often performs poorly for dynamic workloads. Additionally, it can be too expensive in a practical system. TinyLFU uses approximate representation of access frequency to significantly reduce the cost [5]. With a front-end window cache whose default size is 1% of the cache size, W-TinyLFU can additionally handle the issue of access bursts on a small set of blocks. Using its very small LRU-managed window cache and frequency-based admission policy, W-TinyLFU shares the weakness of LIRS2 on workloads with many blocks of a few accesses, an issue that is addressed by LIRS2-Adapt.

To reduce responsiveness issue, frequency-based replacement algorithm (FBR) is proposed to factor out too-old history accesses and give recent history a higher weight in the calculation of frequency [26]. The MQ (Multi-Queue) algorithm separates blocks of different frequencies into different queues [35]. LRFU includes both recency and frequency in the formulation of its locality measure [18]. These algorithms have two major issues. One is their use of hard-to-tune and performance-sensitive parameters, such as relative weight between recency and frequency. Second is their often too-high time overhead. LIRS does not have the issues. It does not have any parameters that must be carefully tuned to receive expected performance. It has a O(1) overhead.

Further away from LFU, algorithms were proposed to consider only a few most recent accesses of a block in the locality measure. Among them, LRU-K uses the Kth-to-last access time, and chooses the block with the largest of the times for replacement [23]. In practice, it is preferred to let K = 2. This sounds similar to LIRS2's locality measure $T_{lirs2} = max(T_{reuse1} + T_{reuse2}, T_{reuse1} + T_{last})$. However, LRU-2's locality measure is fundamentally different. It is $max(T_{reuse1} + T_{last})$, where a change of recency (T_{last}) changes the measurement, making the algorithm carry LRU's weakness. T_{lirs2} doesn't have the issue. Though LRU-2 departs away from LFU by considering two recent accesses, its time overhead is *O(log N)*, where *N* is the cache size. Instead, LIRS2 considers three recent accesses with O(1) overhead.

2Q [11] and ARC [17] also consider only the most recent two accesses of a block, but with O(1) overhead. They use two queues to enable the check-and-trust strategy. Their critical issue is that the condition for passing the check is not well calculated. They simply employ a fixed-size queue and always compare with the size for block categorization. In contrast, LIRS2 uses well-reasoned $T_{lirs2} \leq max_{lirs2}$ condition and uses max_{lirs2}, which changes dynamically to reflect currently true locality strength, as the checking condition.

LIRS2-Adapt dynamically switches between multiple algorithms to receive the best miss ratios. This approach has been employed in a few previous works. SEQ runs between the LRU and MRU (Most Recently Used) algorithms depending on whether a pattern of sequential access of blocks is detected [6]. DEAR detects a few predefined access patterns, such as sequential, looping, temporally-clustered, or probabilistic, and applies different algorithms on blocks of different access patterns [12]. UBM is similar except that it considers file-system-level information for access pattern detection [14]. In contrast, LIRS2-Adapt builds on LIRS2, which has consistently outperformed the state-of-the-arts with only one well-understood exception on a particular access pattern. As this is an issue that can be effectively addressed by LRU, LRU is employed to handle the exception.

4.2 Other Replacement Algorithms

There have been a large number of replacement algorithms for processor cache. Among them, some recent works also adopt the check-and-trust strategy to determine if an incoming cache line is cache-friendly and cache them differently, such as Hawkeye [8], RRIP [9], SHiP [34], and SDBP [13]. The idea of LIRS2-Adapt's dynamical selection from two competing algorithms was also applied in the hardware cache, such as Dynamic Set Sampling (DSS) [24], Sampling Based Adaptive Replacement (SBAR) [25], and Adaptive Cache Management [30]. These algorithms are more concerned with leveraging hardware features to customize general-purpose replacement algorithms for very high efficiency with hardware. The effectiveness of LIRS2 and its O(1) overhead can be helpful to inspire new innovative CPU cache designs.

Recently machine-learning (ML) technique has been employed in the design of replacement algorithms, such as Glider [28], LeCaR [31], DeepCache [19], LRB [29], and CACHEUS [27]. A strength of using ML in the design is that more relevant factors about a caching object can be easily introduced into a learning framework. Though they can be much more expensive and may not be applicable for systems demanding high efficiency, they are more versatile. General-purpose algorithms like LIRS2 usually only consider temporal locality. It is an interesting topic to study how to integrate ML techniques into it in various system scenarios.

5 CONCLUSIONS

In this paper we propose block replacement algorithms (LIRS2 and LIRS2-Adapt) to improve the well-recognized LIRS algorithm. The key contributions are a locality measure covering two reuse distances for more reliable and accurate replacement decision and its effective incorporation in replacement algorithms in a lightweight fashion. Results from extensive experiments show that the algorithms can often substantially outperform state-of-the-art algorithms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Patrick P. C. Lee, for their valuable feedback. We thank Irfan Ahmad and CloudPhysics for supplying us with a large number of real-world I/O traces. Caleb Jiang initially suggested the LIRS2 locality measure. This work was supported by the US National Science Foundation under grant CCF-1815303.

REFERENCES

[1] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. 1971. Principles of Optimal Page Replacement. *J. ACM* 18, 1 (Jan. 1971), 80–93. https://doi.org/10.1145/321623.321632

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[3] Ken Bates and Bruce McNutt. [n.d.]. Storage - UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann

[5] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4, Article 35 (Nov. 2017), 31 pages. https://doi.org/10.1145/3149371

[6] Gideon Glass and Pei Cao. 1997. Adaptive Page Replacement Based on Memory Reference Behavior. *SIGMETRICS Perform. Eval. Rev.* 25, 1 (June 1997), 115–126. https://doi.org/10.1145/258623.258681

[7] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage Using Extent Based Dynamic Tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California) *(FAST'11)*. USENIX Association, USA, 20.

[8] A. Jain and C. Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 78–89.

[9] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 60–71. https://doi.org/10.1145/1816038.1815971

[10] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *SIGMETRICS Perform. Eval. Rev.* 30, 1 (June 2002), 31–42. https://doi.org/10.1145/511399.511340

[11] T. Johnson and D. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*.

[12] Jongmoo Choi, S. H. Noh, Sang Lyul Min, Eun-Yong Ha, and Yookun Cho. 2002. Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme. *IEEE Trans. Comput.* 51, 7 (2002), 793–800.

[13] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 175–186. https://doi.org/10.1109/MICRO.2010.24

[14] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2000. A Low-Overhead High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 9, 16 pages.

[15] Cong Li. 2018. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '18)*. Association for Computing Machinery, New York, NY, USA, 59–64. https://doi.org/10.1145/3211890.3211891

[16] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 222–233. https://doi.org/10.1109/MICRO.2008.4771793

[17] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (San Francisco, CA) *(FAST '03)*. USENIX Association, USA, 115–130.

[18] S. Min, D. Lee, C. Kim, J. Choi, J. Kim, Y. Cho, and S. Noh. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* 50, 12 (dec 2001), 1352–1361. https://doi.org/10.1109/TC.2001.970573

[19] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*. 48–53.

[20] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Trans. Storage* 4, 3, Article 10 (Nov. 2008), 23 pages. https://doi.org/10.1145/1416944.1416949

[21] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/1519065.1519081

[22] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[23] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Rec.* 22, 2 (June 1993), 297–306. https://doi.org/10.1145/170036.170081

[24] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 381–391. https://doi.org/10.1145/1273440.1250709

[25] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. *SIGARCH Comput. Archit. News* 34, 2 (May 2006), 167–178. https://doi.org/10.1145/1150019.1136501

[26] John T. Robinson and Murthy V. Devarakonda. 1990. Data Cache Management Using Frequency-Based Replacement. *SIGMETRICS Perform. Eval. Rev.* 18, 1 (April 1990), 134–142. https://doi.org/10.1145/98460.98523

[27] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 341–354. https://www.usenix.org/conference/fast21/presentation/rodriguez

[28] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3352460.3358319

[29] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/nsdi20/presentation/song

[30] R. Subramanian, Y. Smaragdakis, and G. H. Loh. 2006. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 385–396.

[31] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[32] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[33] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 95–110. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[34] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. 2011. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 430–441. https://doi.org/10.1145/2155620.2155671

[35] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 91–104.