MITHRIL: Mining Sporadic Associations for Cache Prefetching

Juncheng Yang
Emory University
juncheng.yang@emory.edu

Reza Karimi
Emory University
rkarimi@emory.edu

Trausti Sæmundsson
CachePhysics Inc.
trauzti@gmail.com

Avani Wildani
Emory University
avani@mathcs.emory.edu

Ymir Vigfusson
Emory University,
Reykjavik University
ymir@mathcs.emory.edu

Abstract

The growing pressure on cloud application scalability has accentuated storage performance as a critical bottleneck. Although cache replacement algorithms have been extensively studied, cache prefetching – reducing latency by retrieving items before they are actually requested – remains an underexplored area. Existing approaches to history-based prefetching, in particular, provide too few benefits for real systems for the resources they cost.

We propose MITHRIL, a prefetching layer that efficiently exploits historical patterns in cache request associations. MITHRIL is inspired by sporadic association rule mining and only relies on the timestamps of requests. Through evaluation of 135 block-storage traces, we show that MITHRIL is effective, giving an average of a 55% hit ratio increase over LRU and PROBABILITY GRAPH, and a 36% hit ratio gain over AMP at reasonable cost. Finally, we demonstrate the improvement comes from MITHRIL being able to capture mid-frequency blocks.

ACM Reference Format:
Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. 2017. MITHRIL: Mining Sporadic Associations for Cache Prefetching. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 14 pages.
https://doi.org/10.1145/3127479.3131210

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3131210

1 Introduction

As cloud tenants use increasing volumes of data, the pressure mounts on the underlying storage systems to prevent high access latencies for end-users. The prevalent techniques for mitigating block storage access latencies are to cache recently accessed blocks [26], and to prefetch blocks into the cache in advance of anticipated accesses [14, 29].

Current approaches to cache prefetching can be divided into two schools. On one hand, sequential prefetching techniques (such as AMP [7]) anticipate access to consecutive block identifiers, but rely on block I/O with progressive data layout. On the other hand, history-based prefetching seeks to find and exploit deep correlations among past accesses but normally at substantial computational cost [18]. To mitigate overhead and to make caching and prefetching more effective, several applications choose to provide additional hints [23] with each access [4, 9, 18, 19, 27]. Passing extra information, however, requires restructuring, reorganization or modification to the software stack [23], and is infeasible in scenarios where parts of the stack is proprietary.

We argue that to avoid becoming a latency bottleneck, modern block storage systems need general prefetching techniques that fulfill the following criteria.

- **Exploit history.** Various lower layers of storage systems perform sequential prefetching so the focus should be on the more spatially and temporally sophisticated patterns of reuse.
- **Have low overhead.** The methods must be simple, online and impose low time and space overhead.
- **Be backward compatible.** The methods should implement standard legacy interfaces and treat other parts of the storage system as a black-box.

Existing approaches fall short of one or more of these goals: probability graphs and variants incur intensive space or computation overhead [10, 18, 29]; QuickMine is an online algorithm but relies on hints from the applications through

modified interfaces [23] with extra hints from system or applications.

In this paper, we propose MITHRIL, a lightweight online history-based prefetching layer which meets all of the goals. MITHRIL can be coupled with any existing caching layer, even composed with a sequential prefetching layer such as AMP [7]. MITHRIL harnesses several concepts from sporadic association rule mining [16] from the data mining literature. The central idea behind MITHRIL is to track temporal associations between only those blocks whose access patterns are moderately frequent. Intuitively, items that are accessed regularly are already handled by an underlying caching system, such as LRU, whereas items that are rarely accessed need not occupy the precious cache memory. MITHRIL detects associated access patterns between pairs of blocks without relying on application-level hints. In contrast to other history-based prefetching algorithms [10, 18, 19], MITHRIL is able to discover relationships between interleaved requests that are not consecutive – a ubiquitous scenario in modern multi-tenant storage systems – without incurring high computation overhead. The focus of this paper is on exploiting patterns in block I/O workloads, but evidence shows that MITHRIL works on proxy workloads as well. We evaluated MITHRIL through experiments on traces from a commercial I/O caching analytics service, CloudPhysics [26], as well as file system traces from Microsoft Research (MSR) [22]. We found that MITHRIL boosts the cache hit ratio by up to 7× over typical cache strategies (LRU) and improves over the state-of-the-art sequential prefetching algorithm AMP by 36% on average.

Our paper makes three contributions.

- A design of a history-based prefetching layer MITHRIL that leverages a novel, low-overhead algorithm to mine for regularity in request timestamps in an optimized manner.
- A trace-driven experimental evaluation of MITHRIL on 135 traces from real storage systems, showing that our MITHRIL layer effectively discovers block associations for prefetching. On average, MITHRIL increased hit ratio by 56% over LRU, and 36% over AMP. We also measured the latency of MITHRIL on a real system.
- A demonstration that MITHRIL discovers associations between separated blocks from interleaved applications, and the power of MITHRIL stems from being able to capture mid-frequency blocks.

2 Background and Motivation

Caching has been widely studied over the past 70 years. The standard algorithm of evicting the least-recently-used elements (LRU) has seen some structural improvements over the years [15, 21, 24, 30]. A complementary approach is to prefetch data into the cache before it is used, typically either based on sequential or historical patterns [23, 29]. We argue there is room for improvement for prefetching on block I/O workloads.

Sequential prefetching is exploited at lower layers. In sequential prefetching, the storage server exploits spatial locality in the I/O request stream by retrieving a batch of consecutive blocks upon detecting a sequential access pattern [6, 17]. Static size sequential prefetching is well-understood, simple to implement and has seen long deployment, but can cause cache pollution in workloads where the sequential correlation length is variable and affect accuracy.

Cloud environments, however, exhibit high levels of concurrency. This results in I/O workloads where multiple applications interleave I/O accesses that break the continuity of consecutive access patterns [23]. Adaptive algorithms such as AMP (Adaptive Multi-stream Prefetching) [6, 7] and TAP (Table-based Prefetching) [17] dynamically decide when and how much to prefetch. AMP, for instance, dynamically adjusts the number of pages to be prefetched to prevent both cache pollution and prefetch wastage when the requests streams are interleaved. AMP increases its prefetch degree if the prefetched blocks are waited on by system, and decreased if prefetched blocks are evicted without being used. Unlike other prefetching algorithms, which use read cache to detect sequential streams, TAP uses a table to detect sequentiality and track longer history. Thus, TAP outperforms AMP on interleaved workloads and at small cache sizes.

Sequential prefetching has been widely deployed and commonly used in operating systems [2, 20], databases [25] and storage controllers [8]. The ubiquity and success of the approach at lower layers, however, makes the approach less attractive for higher layers in the storage hierarchy, such as at the virtualization layer. In modern workloads, the length of contiguous I/O sequences, furthermore, tend to be short at the lowest levels of the storage hierarchy [29] due to virtualization, multi-tenancy, disk encryption and sophisticated file system layouts. Together, these trends reduce the effectiveness of sequential prefetching on today's storage workloads.

History-based prefetching has been expensive. History-based prefetching, in contrast, tolerates discontinuity across repeating patterns at the cost of added complexity and overhead [10, 14]. One approach is to generate a directed probability graph over accessed items, where an arc denotes one item is likely accessed before the other, and arcs are weighed by the probability of an access [1, 11, 29]. Many systems try to prevent graph metadata from becoming unwieldy by operating at the file-level instead of the block-level [1, 10, 11], which has inherent limitations [14].

Another take on history-based prefetching is to leverage data mining techniques to identify repeating sequences. By mapping a block to an item, using frequent sequence mining on the request sequence, we can obtain frequent subsequences in an access stream. A frequent subsequence implies that the

Table 1: *Comparison of common prefetching approaches. Overhead and improvement is measured over over* LRU *on 135 traces (see Sec. 5).* Backward compatible *algorithms require no hints or changes to legacy interfaces.* General *approaches generalize beyond block I/O traces.*

Algorithm	Time overhead	Space overhead	Avg. hit ratio improvement	Max. hit ratio improvement	Online	Backward compatible	General
AMP [6]	Low	Low	12.2%	139%	✓	✓	✗
PG [10]	Low	High	4.1%	156%	✓	✓	✓
C-Miner [18]	High	Moderate	*N/A*	*N/A*	✗	✓	✓
QuickMine [23]	Moderate	Moderate	*N/A*	*N/A*	✓	✗	✓
MITHRIL	Moderate	Moderate	54.3%	740%	✓	✓	✓

involved blocks are frequently accessed together. In other words, frequent subsequences are good indicators for block correlations in a storage system. C-Miner [18] and QuickMine [23] employ this technique to discover block correlations in storage systems. However, precise data mining technique comes with high overhead. C-Miner only runs offline due to its overhead. QuickMine improves on the issue by tagging each application I/O block request with a context identifier corresponding to the higher level application context (*e.g.*, a web interaction, database transaction, *etc.*). The tag enables the request sequence to be split before mining, thus making computation overhead manageable. The key novelty of QuickMine lies in detecting and leveraging block correlations within logical application contexts. Nevertheless, it depends on explicit contextual hints from applications, which makes it hard to deploy and impractical for legacy systems.

Current history-based prefetching approaches may capture complex access patterns, but require either explicit contextual information from applications or suffer from high runtime overheads.

In addition to the high overhead imposed by history-based prefetching itself, the ensuing small random read requests further deteriorates performance on traditional mechanical disks, although the problem is minimized by the rapid proliferation of SSDs.

Temporal block associations should be exploited. Block associations are common in storage systems [18]. Sequential prefetching aims to exploit spatially associated blocks, yet temporal associations are equally important for prefetching. Lacking a fast history-based approach, our goal in this paper is thus to *efficiently find temporally associated blocks*. Table 1 shows the main algorithms for comparison.

3 Data Mining Techniques

In search for an approach to efficiently gather history for cache requests to improve on prefetching, we survey relevant problems from the data mining literature before describing our approach.

3.1 Sporadic Association Rule Mining

Frequent itemset mining aims to discover which items co-occur frequently in a transaction database. In this field, a group of items is called an *itemset*, and the number of transactions containing this itemset in the database is called *support*. Suppose we have a transaction database. We say an itemset A is *frequent* if its support $support_A$ is larger than or equal to some threshold, *minimum support* R.

Association rule mining is the discovery of a relationship between items a and b in a frequent itemset discovered from the previous step. We say $a \Rightarrow b$ if the probability of b appearing given a is above a threshold.

Sporadic association rule mining focuses on associations composed of mid-frequency items. It usually consists of three steps. In the first step, frequent itemsets are generated like before. The following step filters out highly frequent itemsets, which are defined as those appearing more than *maximum support* S times; and the frequent itemsets left are called sporadic frequent itemsets. In the third step, association rule mining is used to generate association rules from the sporadic frequent itemsets. By definition, only mid-frequency itemsets and association rules are discovered during the process [13].

3.2 Generalizing to Block Associations

Let $B = \{b_1, b_2, \ldots, b_n\}$ be a sequence of cache block I/O requests. In order to conduct effective prefetching, we need to identify pairs of requests $\{b_x, b_y\}$ that are likely to co-occur but not too frequently to be captured by the underlying cache. Notice the similarity to sporadic association rule mining: both try to find related items that appear close by and have mid-range frequency.

To discover such an association, the basic idea is to apply an existing available sporadic association rule mining algorithm [16]. However, there are several challenges. A typical storage system can serve up to billions of requests per day, resulting in an unmanageably long request B. In order to conduct sporadic association rule mining on the data, we need to transform the request sequence into a transaction database as the first step.

The first difficulty is determining how to split B into transactions. One approach is to split B according to wall clock

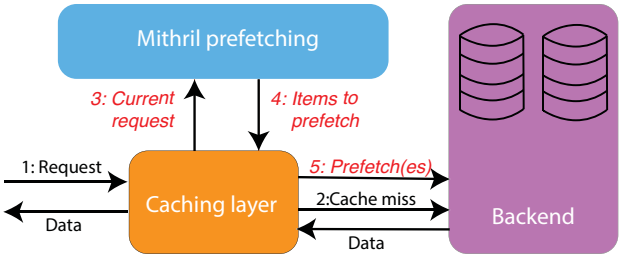


Figure 1: *Schematic of the* Mithril *prefetching layer.*

time, for example, splitting requests into transactions every five seconds. Another approach is to split B using some fixed number of requests per transaction, *e.g.*, group every 20 requests into a transaction. However, both approaches result in information loss, because no evidence indicates that two requests separated in different transactions are not associated. Recall that only items in the same transaction can be discovered as frequent itemsets and as being potentially associated. To address this problem, Soundararajan's approach [23] using a context given by an application to split the sequence is effective but requires changes to the underlying system to obtain such hints, which sacrifices the generality for which MITHRIL is designed.

The second difficulty comes from the high time and space complexity of the currently available sporadic association rule mining algorithms. Koh [16] proposed an optimization for mining sporadic association rules using APRIORI-INVERSE. Their algorithm, however, still requires two phases: mining all sporadically frequent itemsets and discovering sporadic association rules. Although the algorithm avoids generating and storing highly frequent itemsets, APRIORI-INVERSE still needs to store and count all possible associated pairs at significant computation and storage overheads, as confirmed using the SPMF library[5].

To efficiently discover associations between requests without requiring extra application-level hints, we propose the MITHRIL prefetching layer, whose algorithm provides a fast approximation to sporadic association rule mining.

4 Design of MITHRIL

MITHRIL is a prefetching layer between the existing caching layer and the backend, as shown in Figure 1. Without MITHRIL, when a request arrives, it first touches the caching layer; if it is a cache hit, it returns directly from the cache, otherwise, as a cache miss, the application or caching layer needs to go to the backend to fetch the item. When MITHRIL is added, when a request arrives, MITHRIL records the request for mining, checks the potential prefetching list, and sends the request(s) to the caching layer for prefetching.

4.1 MITHRIL Mining

We now describe the algorithm at the core of our prefetching layer. Let B be a sequence of unique block I/O addresses $B = \{b_1, b_2, \ldots, b_n\}$ where a request b_i has a logical *time-stamp* of i, also known as its reference number. Let T be an $n \times S$ matrix for $S = maximum\ support$, where ith row $\vec{T_i}$ corresponds to request b_i, and the cells of each row contain a sorted list of increasing time-stamps. In addition, T is also sorted by the first time-stamp of each block. Figure 2 illustrates the request sequence and corresponding time-stamp matrix T (all the symbols are listed in Table 2).

Table 2: *Symbols used in the text*

Symbol	Meaning
T	*Time-stamp Matrix*
R	*Minimum Support*
S	*Maximum Support*
Δ	*Lookahead range*
M	*Maximum Metadata Size*
P	*Prefetching List Size*

An *associated block pair* refers to two blocks that are *repeatedly* accessed *in sequence*. In modern systems, due to multiple applications interleaving with each other, two consecutive accesses from the same stream may not appear consecutive in the final stream, so we define a *lookahead range* Δ that specifies the maximum allowed distance between two *associated* blocks. In order to establish an association between two blocks, not only do they need to appear within Δ of each other, but also they need to appear with some minimum frequency. We denote this threshold as *minimum support* R. Since our prefetching layer assumes the presence of a cache to catch frequent items, we specify *maximum support* S as the upper bound for items to be considered for mining within a certain time interval. We remark that each of these requirements have conceptual counterparts in sporadic association rule mining.

To further distinguish associated block pairs, as illustrated in Fig 2, we define two blocks as being ***weakly associated*** if each time-stamp pair of the two blocks is within Δ; furthermore, if a *weakly associated* pair is accessed strictly consecutively (time-stamp difference 1) at least once, we define it as being ***strongly associated***.

The reason for distinguishing *weakly associated* pairs and *strongly associated* pairs is that two blocks in a *strongly-associated* pair are more likely to be related, which is preferred for prefetching. Moreover, this effectively limits the number of associations we find. In cases where multiple applications interleaving and a *strong association* does not always exist for a block, then we consider its first (also closest) *weakly associated* pairs. Therefore, only a *strongly associated* pair and the closest *weakly associated* pair are considered.

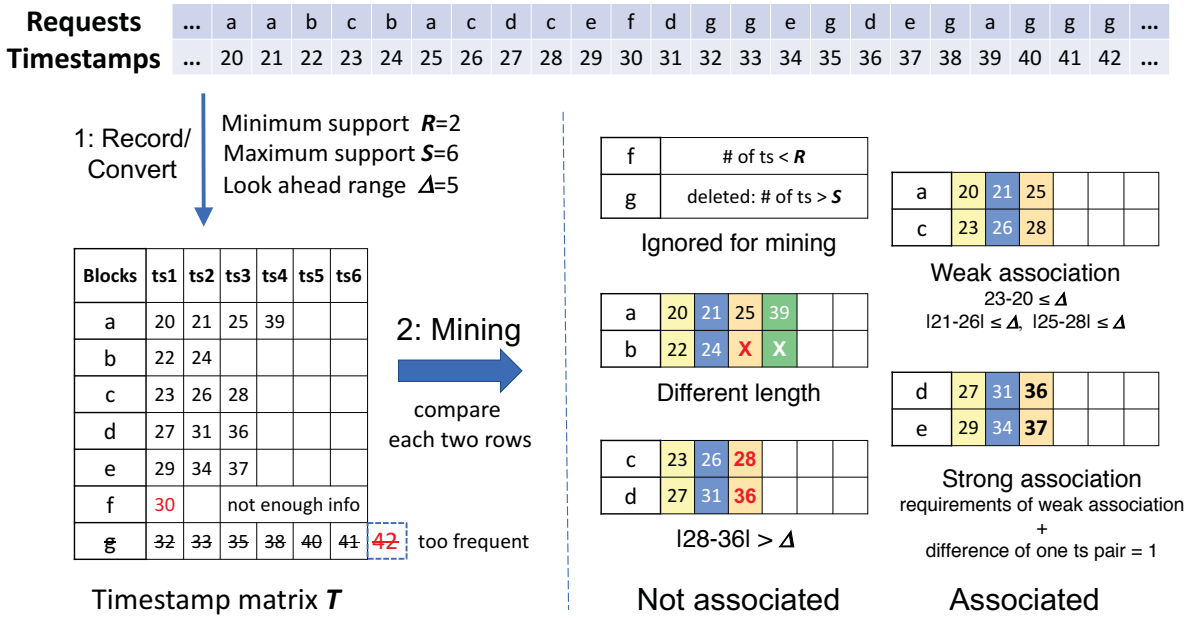


Figure 2: ***Illustration of mining procedure.*** *If input is a request sequence, convert it into time-stamp matrix T. Blocks that have fewer than R time-stamps(ts) or more than S time-stamps are not considered for mining. For each two-block pair, if they have different numbers of time-stamps, or the difference between at least one time-stamp pair is greater than Δ, they are not associated. If all time-stamp pairs are within Δ, they are weakly-associated. Furthermore if they have at least one time-stamp pair with difference 1, they are strongly-associated.*

We present the basic version of MITHRIL in Algorithm 2. The function *checkAssociation* (Algorithm 1) receives two rows from T as input and checks whether the corresponding two blocks are weakly or strongly associated or not.

Algorithm 2 shows the mining procedure, which uses O(N) time to discover associated block pairs. N is the number of unique blocks requested during the recording interval. The input of the algorithm can be the request sequence B or the time-stamp matrix T. If the input is B, then we need to first convert it into T in O(N) time.

In the outer loop, we iterate through all rows in T. For each block b_i, we check all other blocks in the inner loop to find b_j that are either *strongly associated* or are the first *weakly associated* occurrence. Because T is sorted by first time-stamp of each block, so at inner loop at most Δ blocks are checked. Typically, the number of blocks checked is much less than Δ.

After an associated block pair is unveiled, it is stored in the *prefetching table*, which is checked for prefetching upon each request.

Algorithm 1: checkAssociation

Input: Rows $R1$ and $R2$ from time-stamp matrix T, associationType *assoc*, *lookahead range* Δ
Result: Whether b_1 and b_2 are associated

```
consecutive ← False
if len(R1) − len(R2) ≠ 0 then
    return False
for k ← 1 to len(R1) do
    if abs(R1[k] − R2[k]) > Δ then
        return False
    if abs(R1[k] − R2[k]) == 1 then
        consecutive ← True
if assoc == weak then
    return True
else if assoc == strong then
    return consecutive
```

4.2 Optimizations

When MITHRIL is run, a two-dimensional time-stamp matrix T is initialized. For each new request, if it is found in T, the

Algorithm 2: MITHRIL mining procedure

Input: time-stamp matrix T, *minimum support* R, *lookahead range* Δ
Result: Associated block pairs

```
for i=1 to len(T)-1 do
    if len(T[i]) < R then
        continue
    associationType ← weak
    for j=i+1 to len(T) do
        if checkAssociation(T[i], T[j], associationType) then
            addAssociation(block_i, block_j)
            associationType ← strong
        if (T[j][0] − T[i][0]) > Δ then
            break
```

current time-stamp is appended to the corresponding row. Otherwise, the request is recorded in a new row. We append the time-stamp to a row. When the row is full, the block is considered frequent and deleted from the matrix and recorded in the frequent block hashmap. Items from this hashmap are ignored when encountered again before the mining process. When the time-stamp matrix T is full, the mining procedure is called and the associated blocks are saved in the prefetching table. After mining completes, recording starts anew with a clean state.

The version of MITHRIL described so far requires a large matrix with *maximum support S* columns for storing time-stamps, a hashmap mapping from block number to the corresponding row in the matrix and a hashmap for determining whether a block is frequent. Additionally, a prefetching table is needed for storing associated block pairs for prefetching. However, spending limited cache space on tracking large metadata is not desirable. To address the metadata space usage of basic MITHRIL, we made the following optimizations, which use bounded memory in exchange for some added complexity.

4.2.1 Recording and Mining

Splitting recording table. The two-dimensional recording table (time-stamp matrix) is a sparse matrix, since a typical block, by definition, will be requested fewer than *maximum support S* times within a recording period. A naïve implementation uses a linked list for each block instead of a fixed-size array. However, the space for link pointers between time-stamp nodes doubles the space overhead. We exploit the sparsity by decomposing the large matrix into two smaller fixed-sized tables: one with *minimum support R* columns, which is the *recording table*, and the other one with *maximum support S* columns, which we call the *mining table*. The *recording table* is a circular array in which new entries replace old entries in FIFO fashion. The *mining table* is a fixed-size array that triggers the mining procedure when full.

When a block request arrives, the time-stamp is recorded in the *recording table*. If the number of time-stamps in the corresponding row of the *recording table* has reached *minimum support R*, in other words, when the row is full, it is declared to be **mining-ready** and then transferred into the *mining table*, which can store up to S time-stamps for each block. After migrating one row from the *recording table* to the *mining table*, the last row in the *recording table* is moved up to the migrated row to make the table compact. When the *mining table* is full, in other words, when there is no more room to store new mining-ready blocks, the mining procedure is triggered to discover associated block pairs and store them in the *prefetching table* for prefetching. When the mining finishes, the *mining table* is cleared. When the *recording table* is full, we replace the oldest entry with a new entry with the assumption that the oldest block remaining in the table is rare since it has not been requested R times within the interval.

Decomposing the original matrix not only saves space, but also allows for more blocks to be tracked. Because the *recording table* does not need to be cleared each time, we retain extra information for blocks that are not mining-ready. In the unoptimized approach, the large time-stamp matrix was cleared each time the mining finishes, discarding all information.

The primary drawback of splitting is that the *mining table* needs to be sorted before mining. This is because Algorithm 2 requires input to be sorted by the first time-stamp, which occurs automatically in our single-table construction. Since our separate *mining table* is created by inserting elements in the order of accumulating R time-stamps, sorting the *mining table* before mining is necessary. In practice, however, the size of the *mining table* is usually small and sorting is trivial. A secondary concern is that when mining begins, some associated blocks may collect more time-stamps than others due to the cut-off (misalignment) between the tables. This behavior is rare and affects only a small number of associations found. Since MITHRIL is an approximation, missing a few associations is not critical. Our focus is instead of on balancing the overhead and the benefits.

Compressing time-stamps. To further reduce the space used by the *recording table* and the *mining table*, we compress time-stamps by storing only the lower 15 bits. This allows us to store four time-stamps in the lower 60 bits of one 64-bit integer with a time-stamp counter stored in the higher 4 bits. Moreover, one could further compress time-stamps by removing the last $\lfloor \log_2(\Delta) \rfloor$ bits – we omitted this optimization in our experiments to limit time overhead.

Removing the frequent block hashmap. A block that is requested more than S times in each recording interval in the original MITHRIL approach is considered to be a frequent

block, so no information should be recorded. To track the requests, one could use a hashmap or Bloom filter, but hashmaps require extra memory and Bloom filters incur extra computation overhead. Instead, we decide to record a block only on cache miss. In this way, all frequent blocks are automatically filtered out by the underlying cache. There are several other benefits. First, MITHRIL need not be invoked when cache sizes are sufficiently large and *minimum support R* is greater than 1. This behavior happens gradually over larger cache sizes since the mining phase will be run less frequently. Second, if a block is accessed frequently over a short period, the optimized recording method cuts down overhead since it only records cache misses, thus precluding spuriously recording frequently accessed blocks. If the cache size is small, recording bursts and thus prefetching frequent items is useful since these blocks are constantly being evicted by the underlying cache.

Our optimizations trade off storage, computation overhead, prediction precision and hit-ratio improvement. The more useful information we record, the higher hit rate and precision can be achieved, but at the same time more overhead is incurred. Besides recording at cache miss as mentioned above, optionally we can also record the time-stamp when a block is evicted from the cache to obtain more information about the block. Recording at eviction is similar to recording at cache miss: in both approaches, the frequent blocks are filtered out by the underlying cache.

4.2.2 Prefetching

Splitting the prefetching table into shards. We use a two-dimensional array instead of lists to store associated block pairs together for storage reduction for the same reason as using an array in the *recording table*. In the *prefetching table*, the first column stores the originated block number b_x, while the rest of the columns store the blocks that are associated with b_y. The number of columns left is the maximum number of possible block pairs, defined as *prefetching list size P*. We use a default of three columns, indicating that, at most two block pairs can be stored for each block. For example, in an association $b_x \rightarrow b_y$, b_x is stored in the first column and b_y is stored in the second column. If there is another association, $b_x \rightarrow b_z$, then the third column stores b_z. If more than two associations are discovered, we replace the old associations in a FIFO manner, which allows MITHRIL to adapt to changing workloads. Meanwhile, we do not differentiate strong and weak associations in the *prefetching table*.

Since cache behavior varies in different workloads, it is impossible to know how many blocks will have associations ahead of time, and thus how much memory will be needed. Therefore, we introduce the concept of shards. A shard is a *prefetching table* with 2000 rows that is dynamically allocated when needed. When a user specifies a *maximum metadata size M* can be used for MITHRIL, an upper bound is placed on the number of possible shards. When all possible shards are allocated, a new row will replace the oldest row.

By introducing shards, we aim to find a balance between frequent allocation and overallocation of memory. In addition to saving metadata memory usage, the maximum memory usage is also bounded by *maximum metadata size M*.

Since prefetched blocks are also added to the original cache pool, it is possible for a prefetched block to be evicted before it is used. As other authors suggest [6, 8], we give the prefetched block a second chance by re-adding it to the MRU end of cache if it is going to be evicted without being accessed.

4.3 Using MITHRIL

Using MITHRIL as a prefetching layer requires minor modifications to the underlying caching layer. The complete flow of MITHRIL is shown in Algorithm 3. A prefetch from MITHRIL requires passing one parameter and two indicators. The parameter is the current block number, which is used for recording, prefetching or both. The two indicators are pFlag and rFlag, which indicates whether it is for recording or prefetching.

There are two scenarios where the MITHRIL API may be called. First, when a request arrives, MITHRIL must check whether prefetching is needed. In this situation, `pFlag = True` and `rFlag = False`. Second, to handle recording when `rFlag = True` and `pFlag = False`. This recording may be invoked (a) at the arrival of each request, (b) only at cache misses, (c) only during cache eviction, or (d) during both misses and eviction. Recording at each request or recording at both misses and evictions increases the computation overhead. As we demonstrate in Section 5.4, recording on the arrival of each request optimizes performance, whereas recording only at cache misses provides similar performance at much lower overhead. In contrast, we find the two approaches (c, d) recording on eviction do not to provide competitive performance.

4.4 Complexity Analysis

Time complexity. Compared to LRU, the only operations added to each request are to record the current logical time-stamps in the *recording table* on a cache miss and check the *prefetching table* and prefetch when needed. Each of these operations has a time complexity of $O(1)$, so the total computation overhead at each request is negligible. Periodically, the mining procedure runs and is dominated by an $O(N \log N)$ sort, where N is a fixed, typically small table size. The mining process can furthermore be run in a background thread and thus avoid blocking new requests.

Space complexity. In the optimized MITHRIL, we store all time-stamps as 15-bit integers with four time-stamps in one 64-bit integer. Thus if we have *maximum support S*=8,

Algorithm 3: The MITHRIL main algorithm.

Input: recording table $rTable$, mining table $mTable$, prefetching table $pTable$, minimum support R, block# b, prefetchingFlag $pFlag$, recordingFlag $rFlag$
Output: blocks to prefetch

```
1  ts ← 0
2  if rFlag then
3      tsRow ← pTable[b]
4      append ts to tsRow
5      if len(tsRow) ≥ R then
6          move tsRow to mTable
7          move last row in rTable to tsRow
8          if mTable is full then
9              mining()
10             clear mTable
11     ts ← ts + 1
12 if pFlag then
13     if b in pTable then
14         return pTable[b]
15 return NULL (no need to prefetch)
```

minimum support R=4, *recording table* size 100,000 and *mining table* size 1,250, recording and mining will need less than 2MiB. When calculating size of `hashtable`, which maps from block address to index in *recording table* or *mining table*, the 8 byte is used for storing block address, the 4 bytes is used for storing the index.

Since all information is stored in a bounded array, the *maximum metadata size M* allocated is usually set to 10%, which is more than enough in most cases. And in our evaluation, we count in the memory usage for all metadata for fair comparison, which means when MITHRIL metadata uses 5% of cache space, then only 95% of space will be used for store cache data.

5 Evaluation

We now characterize MITHRIL experimentally with the following questions in mind:

- How much does MITHRIL improve the hit ratio? What are the best and worst cases?
- How well does MITHRIL work with various cache replacement algorithms, and how precise is prefetching?
- How do parameters affect MITHRIL?
- Is latency improvement enough to justify overhead?
- Why does MITHRIL work?

5.1 Methodology

As a history-based prefetching layer, ideally we should compare MITHRIL with C-Miner [18] and QuickMine [23], which are the two state-of-the-art algorithms in history-based prefetching. However, since C-Miner and QuickMine either runs offline or requires context information from the applications, which is not applicable in our setting. Instead we implemented another history-based prefetching technique, PROBABILITY GRAPH (PG) [10], together with a state-of-the-art sequential prefetching algorithm, AMP [6], and LRU to compare to MITHRIL. Note that MITHRIL can be used on top of AMP.

We evaluated algorithms on 106 traces from commercial I/O caching analytics services from CloudPhysics (CP) together with 29 traces obtained by Microsoft Research (MSR) [22] (We omitted traces that have fewer than a half million requests). The CloudPhysics traces are explained in detail by Waldspurger et al. [26]. For simulation-based results, we used the MIMIRCACHE [28] for profiling and analysis on a Microway server of dual E5-2670v3 CPUs with 512GB memory. For the micro benchmark, we modified IOBlazer [3] and ran it on AWS EC2 c3.large instance with an EBS magnetic disk. In this section, if not specified, MITHRIL is used together with LRU, and all experiments showing single trace used trace w94 from CP [26], which is a week-long VM trace. The cache size, if not mentioned, is set to 256MB, which exhibits a range of LRU hit ratios between 10% to 99%. The profiling platform and MITHRIL implementation will be released under open-source after publication [28]. The CP data used in the paper will be released by CloudPhysics separately.

5.2 Overall Hit Ratio Improvement

As a prefetching layer, MITHRIL is unaware of the underlying caching algorithm, which might be either FIFO, LRU, AMP or other possible cache replacement algorithms. In this section, we show that MITHRIL provides benefits for LRU and AMP.

Comparison with PG. PG is the most comparable history-based algorithm, so we compare MITHRIL with PG in this section. In Figure 3, we show the hit ratio of PG and MITHRIL for all the traces. LRU is not shown in the trace because of its high resemblance to PG in terms of average hit ratio and correlation: the Pearson Correlation Coefficient between hit ratio of LRU and PG is 0.993, while it is 0.801 between LRU and MITHRIL. The low correlation between LRU and MITHRIL implies that the performance of MITHRIL does not completely depend on the performance of LRU. Compared to LRU, on average MITHRIL provides 52% relative improvement in the hit ratio on CP traces, and on average achieves 82% of the maximum obtainable hit ratio at small cache size, which is calculated by excluding cold miss. On the 29 MSR traces, MITHRIL provides on average a 64% hit ratio improvement achieving 81% of the maximum obtainable hit ratio. As shown in the figure, the hit ratio improvement for MITHRIL varies between traces. For certain traces, it can provide up to more than 7× improvement, but for some other traces, the

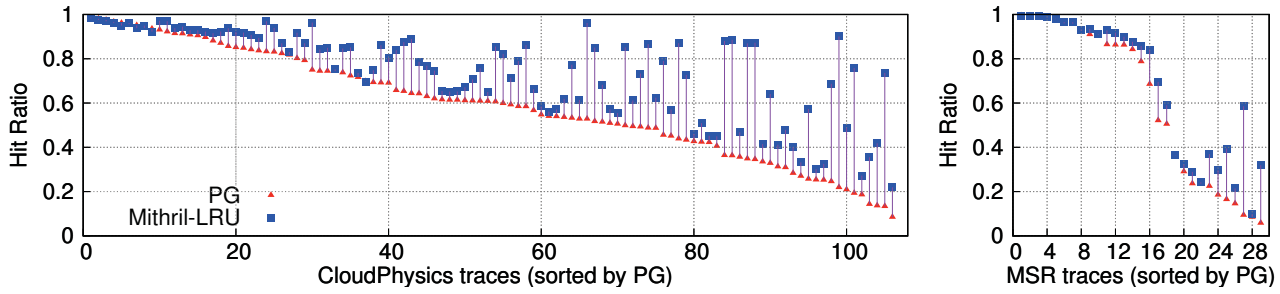


Figure 3: ***Hit ratio of*** **PG** ***and*** **Mithril** ***for 106 CP traces and 29 MSR traces sorted by*** **PG** ***hit ratio.*** *Hit ratio of* LRU *omitted as it is similar to* PG *(Pearson* $r = 0.995$ *compared to* $r = 0.742$ *for* LRU *and* Mithril*). Compared to* PG, Mithril *overall provides significant improvement, even though parameters are not fine-tuned for each trace.*

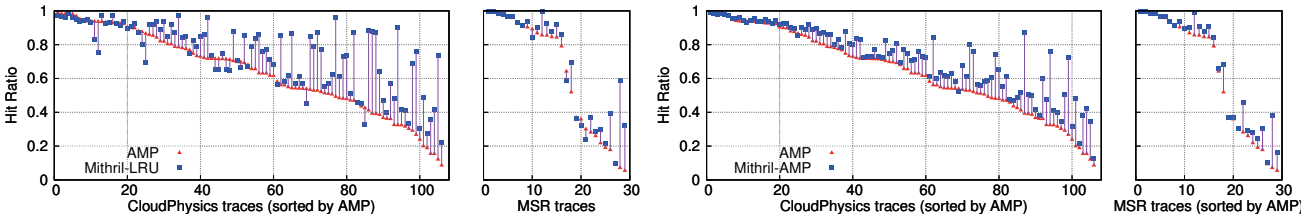


Figure 4: ***Left: Hit ratio of*** **Amp** ***and*** **Mithril-LRU***, right: Hit ratio of* **Amp** ***and*** **Mithril-AMP** ***for CP and MSR traces sorted by*** **Amp** ***hit ratio.*** *Left:* Mithril-LRU *outperforms* Amp *in most traces. For some traces with strong sequentiality,* Amp *has better performance due to its ability to prefetch pages that have never been requested. Right:* Mithril-AMP *improves or matches hit ratio for most traces compared to* Amp.

improvement is more modest, particularly those whose PG hit ratio is already high.

Comparison with AMP. As a prefetching layer, we also compare MITHRIL with state-of-the-art sequential prefetching algorithm AMP, which dynamically captures the spatial associations in the requests. Compared to AMP, MITHRIL on average provides a 31% increase in hit ratio on CP traces and 51% on MSR traces, indicating that by exploring temporal associations, MITHRIL can provide more benefit than AMP. However, as shown in Figure 4, MITHRIL does not always provide more benefit compared to AMP. In some traces where sequentiality is not dominant, MITHRIL provides a great benefit, more than a 7× improvement on hit ratio; in some other traces where sequentiality dominates the disk access pattern, AMP provides more benefit than MITHRIL. The reason AMP outperforms MITHRIL lies in its ability to prefetch blocks that have never been requested. In contrast, MITHRIL does not have this ability. It can only prefetch blocks already seen in the past.

Although AMP surpasses MITHRIL in some cases, MITHRIL as a prefetching layer can be used on top of AMP. In Figure 4, we show the hit ratio obtained by AMP compared to MITHRIL-AMP. Using MITHRIL on top of AMP guarantees at least similar performance as AMP, and still provides a large benefit on most of the traces. This improvement implies that besides spatial-locality, which has been captured by AMP, MITHRIL is capable of further leveraging the temporal-locality associations between requests to gain performance promotion. Note that Figure 3 and Figure 4 cannot be directly compared, because former one is sorted by PG, and latter one is sorted by AMP. However, Figure 4 and Figure 4 are comparable since curves in both figures are sorted by the AMP hit ratios. Adding MITHRIL to AMP guarantees no performance loss compared to AMP, however, MITHRIL-AMP does not guarantee a better performance than MITHRIL-LRU as we see in some of the traces. The reason MITHRIL-LRU can be better than MITHRIL-AMP is that AMP turns some cache misses into cache hits due to its sequential prefetching ability. Thus the relationship seen by MITHRIL is jeopardized, and the associations captured by MITHRIL can be inaccurate. Overall, MITHRIL significantly improves hit ratio over PG and AMP.

Behavior on representative traces. To better illustrate the hit ratio improvement, we select six traces (three from CP and three from MSR) to show typical examples of large (top two), modest (middle two) and small (bottom two) performance gains for MITHRIL in Figure 5. The top two traces show the cases where MITHRIL outperforms the corresponding caching algorithm by at least doubling the hit ratio. The middle two figures show the traces that have relatively high hit ratios under LRU. Adding MITHRIL provides a modest

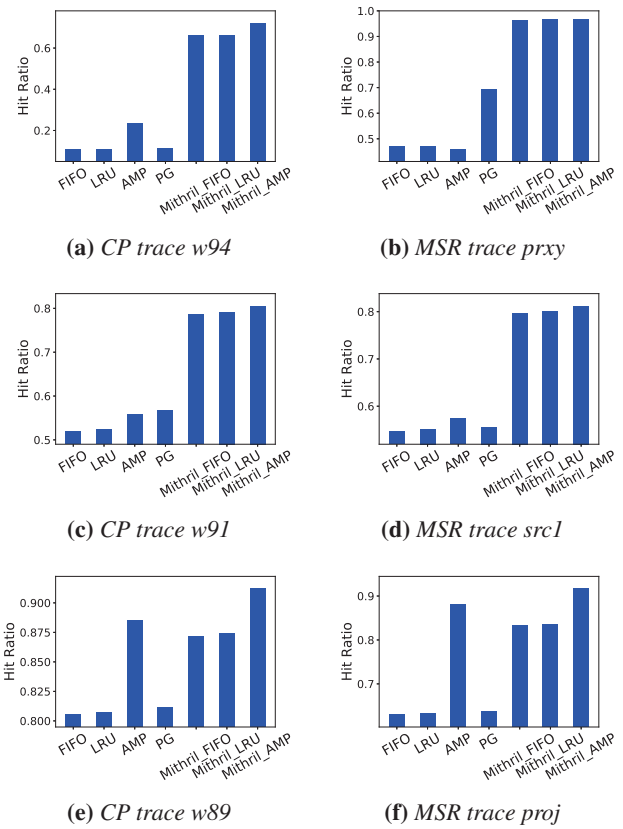
performance improvement. In the bottom two traces, AMP outperforms MITHRIL-LRU by being able to prefetch unseen blocks. However, this can be changed by using MITHRIL with AMP. Still, in these cases, MITHRIL-AMP usually does not win much over AMP in terms of hit ratio because the hit ratios of AMP are often already high, yielding limited potential benefit. In addition, MITHRIL can only prefetch blocks that have already been seen, capping the maximum hit ratio at 1 − *cold miss ratio*. PG is the only prefetching algorithm in same category as MITHRIL. Its performance is unstable, sometimes better than AMP, most of time worse than AMP. For most traces, it outperforms pure LRU and is outdone by MITHRIL.

MITHRIL is compatible with a range of caching algorithms. The figures compare performance of using MITHRIL on top of LRU, FIFO and AMP to that of the original cache replacement algorithms. Adding MITHRIL consistently boosts hit ratio, particularly for simpler cache replacement algorithms. For example, by adding MITHRIL to FIFO, the performance of MITHRIL-FIFO is similar to MITHRIL-LRU, which is much better than FIFO. This property of MITHRIL opens the possibility of using MITHRIL with particular cache replacement algorithms in appropriate situations, for instance when running off of SSDs [24], MITHRIL with FIFO may achieve the best performance. Investigating whether MITHRIL can supplement a wider range of existing cache replacement algorithms is left as future work.

5.3 Cache Size and Precision

To focus the discussion, we will hereby focus only on LRU and MITHRIL-LRU. Our results so far are based on performance at a single cache size. We now show the performance of MITHRIL under a range of cache sizes. Figure 6 shows the hit ratio curve (HRC) of LRU, PG and MITHRIL along with the prefetching precision of the latter two. Shown in HRC, the performance PG is always better than LRU, and as the cache size increases, the gap between PG and LRU increases due to more space allocated for PG'S pair-wise probability matrix. However, the improvement of PG is limited due to its large matrix. In contrast, MITHRIL provides a hit ratio boost even at a small cache size.

The precision curve of PG has several peaks and troughs because the size of its comprehensive conditional probability matrix depends on cache size. As the cache size increases, the matrix size grows. However, precision may not benefit from the increasing probability matrix size due to wrong new predictions. Similarly, the precision curve for MITHRIL is also not monotonic, especially with a small cache size, due to the eviction of prefetched blocks before being requested. When comparing the prefetching precision of PG and MITHRIL, we see that, in most situations, MITHRIL has better precision than PG.



(a) *CP trace w94* **(b)** *MSR trace prxy*

(c) *CP trace w91* **(d)** *MSR trace src1*

(e) *CP trace w89* **(f)** *MSR trace proj*

Figure 5: ***Hit ratio of different algorithms.*** *Example traces where* Mithril *significantly improves hit rate (top two), where* Mithril *shows modest improvement (middle two), and where* Mithril *shows little or no performance gain (bottom two).*

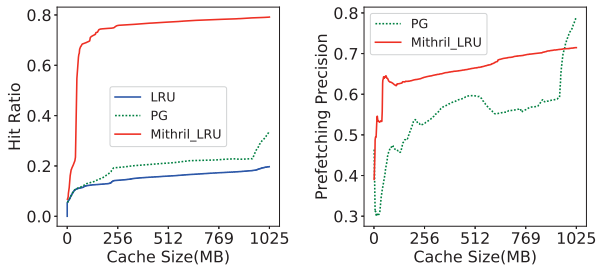
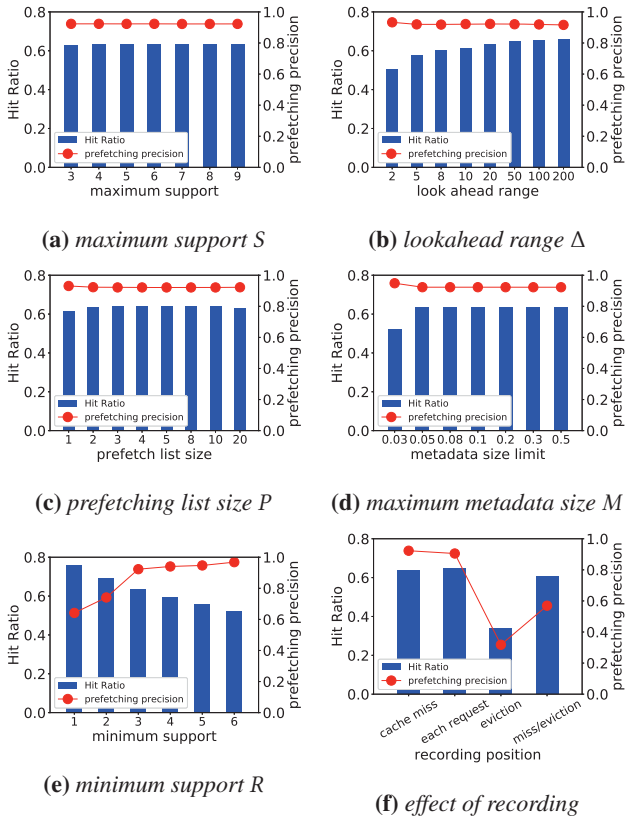


Figure 6: ***Hit ratio curve and prefetching precision of*** **LRU, PG *and* Mithril** *Left:* Mithril *outperforms* LRU *and* PG. *Right: The prefetching precision of* Mithril *is higher than* PG *and both two curves are not monotonic.*

5.4 Effects of Parameters

MITHRIL uses several parameters that now investigate in isolation in terms of impact on hit ratio and prefetching precision using a representative CP trace (w94).



(a) *maximum support S* **(b)** *lookahead range* Δ

(c) *prefetching list size P* **(d)** *maximum metadata size M*

(e) *minimum support R* **(f)** *effect of recording*

Figure 7: *Effect of parameters in* Mithril.

Maximum support *S* decides the maximum allowed degree of hotness of a block. This is decided by considering the row length of the mining table. If a block is requested more than *S* times before mining, it gets kicked out as a frequent block. As shown in Figure 7a, *S* has a small effect on hit ratio and prefetching precision since most of the frequent blocks are already filtered out by an underlying caching layer. Recall that MITHRIL records blocks only during cache misses.

Lookahead range Δ decides the maximum allowed timestamp difference for two blocks to be considered associated. It is obvious that Δ should be a parameter related to the number of concurrent running processes. If too large, non-associated block pairs will be mistaken as associated, thus increasing the false positive rate. On the other hand, being too small will result in many associations being ignored and thus a high false negative rate. As shown in Figure 7b, when Δ is small, as Δ increases, the hit ratio increases substantially, while prefetching precision decreases slightly. After certain threshold, further increasing Δ will not increase hit ratio. This is because the best Δ should relate to the number of concurrent running applications (at least as large as it), the given trace shown in the figure has its best Δ around 50.

Prefetching list size *P* determines the space that can be used for storing associated blocks, which is the row length of the prefetching table. Recall that when more than *P* associated blocks are discovered, the old blocks are replaced in a FIFO manner. Figure 7c shows that increasing *P* dramatically reduces prefetching precision because a large *P* means stale associations are also stored for prefetching. On the other hand, the hit ratio first increases and then decreases with an increasing *P*. We notice that setting *P* as 2 gives an acceptable trade-off between hit ratio and precision across the various datasets we considered.

Maximum metadata size *M* decides the maximum space MITHRIL can use for the recording table, mining table and prefetching table. As illustrated in Figure 7d, if *M* is too small, there are not enough spaces for the prefetching table, dramatically reducing the effect of MITHRIL. After a threshold, further increasing *M* won't increase the hit ratio. However, setting *M* too large in situations that MITHRIL does not have good performance will waste space which should be used for caching. We thus recommend a default value of 10% of the entire cache space based on traces we have tested.

Minimum support *R* has the largest effect on the performance of MITHRIL. It decides when a request is ready for mining and is the row length of the recording table. In Figure 7e, we can see that increasing *R* will increase prefetching precision, while reducing the hit ratio. Two requests are required to appear closely *R* times to be considered associated, and when we have a larger *R*, the requirement for being associated is stricter, which diminishes the number of associations and grows the confidence of discovered associations.

Different recording locations also have a large effect on the performance of MITHRIL. As mentioned in Section 4, we record only at cache misses, which reduces computation by recording only the most important information. As shown in Figure 7f, besides recording a) at cache miss, we can also record b) when a block is evicted from cache, c) at cache miss and eviction, or d) each time a request arrives. Using c) and d) usually give more information to MITHRIL at a cost of more computation. In other words, we can trade CPU cycles for potentially better hit ratio and precision. As we observe across the traces, recording at evictions (b) usually cannot provide good performance; recording at evictions and misses (c) occasionally provides similar performance to the other two approaches a and d, but most of the time only slightly better than recording at evictions (b). In contrast, recording at the arrival of each request (d) usually gives the best performance with the highest precision. As an alternative, recording at cache misses (a) can greatly reduce the overhead of MITHRIL, while, as we have evaluated in most traces, it provides less than a 10% performance loss compared to recording at each request.

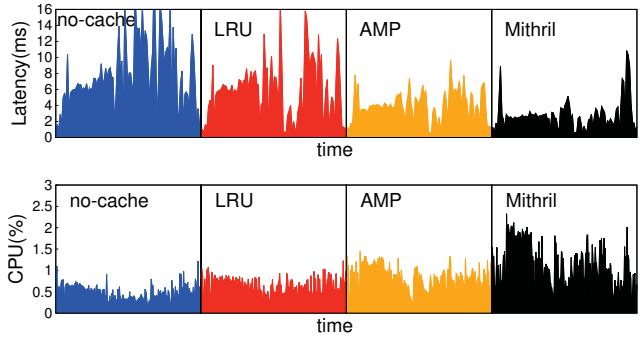


Figure 8: ***Latency and CPU usage of using no cache,*** **LRU, Amp** ***and*** **Mithril-LRU.** *On the top, each latency point is the average latency of 40000 requests. At the bottom, it shows the relatively increased CPU usage of* Mithril *due to mining and prefetching, compared to* LRU *and* Amp*, the increase is less than 1%.*

Table 3: *Latency Percentile (microseconds)*

Percentile	50%	75%	90%	99%
no cache	6971	10403	13625	18072
LRU	5883	7452	10038	15801
AMP	3934	5097	6163	7765
Mithril	2551	3063	4173	10330

5.5 Real System Performance

Latency. A high hit ratio may not mean low latency in a real system because of factors such as CPU overhead and late prefetch. Especially for a history-based prefetching, the cost of prefetching a random block is large. In Figure 8, we justify the overhead compared with benefit. It shows the latency of four approaches on CP trace w94: using no cache, using LRU cache, using AMP and using LRU cache with a MITHRIL prefetching layer. *Compared to no cache,* LRU *reduces average latency by more than 26%, especially at the peaks, where the no-cache system shows a high latency. Using a sequential prefetcher* Amp*, the latency further decreases by 32% over* LRU *on average, whereas* Mithril *with* LRU *reduced latency by 52% over* LRU. Besides average latency, the latency percentiles in Table 3 further illustrate the effectiveness of MITHRIL on reducing latency. However, we do see that at 99% percentile, MITHRIL has a higher latency over AMP, which is caused by latency peaks discussed below.

Late prefetches. Although latency reduction due to MITHRIL prefetching is evident, we also see that 22.4% of prefetches are late, which means the arrival of prefetched blocks happen after the time they are requested. Late prefetches affect the performance of MITHRIL by wasting one disk read unless caught by the disk controller.

MITHRIL warm up time. In Figure 8, focusing on the first 5% percent of the requests in a system with MITHRIL, we can see the there is no latency reduction at beginning, and latency decrease as time goes from 0% to 10%. The decrease occurs because MITHRIL needs sufficiently many requests for warm-up before it conducts mining and prefetching.

Existence of latency peaks. MITHRIL does not eliminate all latency peaks. The peaks stem chiefly from two phenomena: they are due to long disk rotational latency or a burst of requests, or a mix of these aspects. When the peaks occur due to long disk rotational latency, MITHRIL can effectively reduce latency by prefetching. One extreme case would be if each block request demands the disk to rotate half way to retrieve the content, causing peaks in a system without MITHRIL. However, in systems with MITHRIL, associations between these requests would be unveiled and harnessed. In other words, MITHRIL would prefetch associated block into the cache ahead of its request time, thus lowering latency. On the other hand, if the latency peak is caused by a large number of outstanding I/Os [12], MITHRIL provides less benefit because issuing prefetches only increases the burden on the disk. Consequently, not all latency peaks can be removed by MITHRIL.

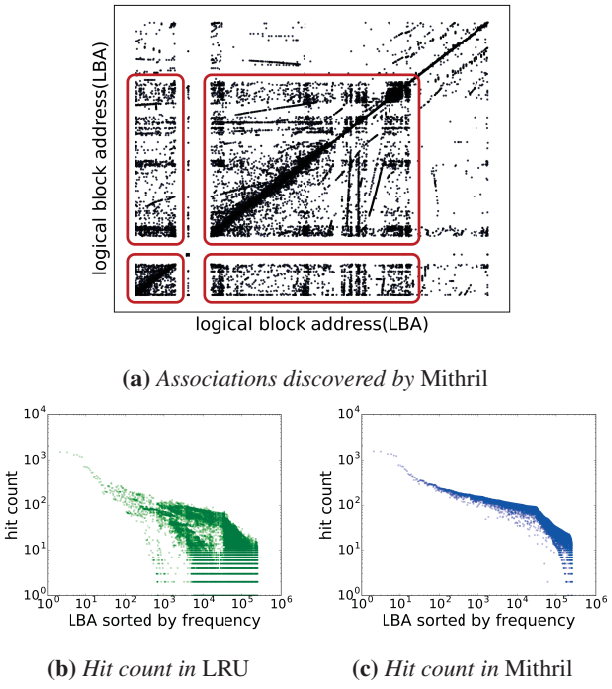
CPU usage. MITHRIL is based on approximate association mining, which might be CPU-intensive. As shown in the figure, we see some CPU consumption increase for MITHRIL, however, the increase is minor and within the limits afforded by many storage systems.

5.6 MITHRIL Analysis

In this section, we analyze the behavior of MITHRIL underlying its performance. Figure 9a shows the associations discovered by MITHRIL after a full trace run. Both horizontal and vertical axes are logical block addresses (LBA): if two blocks b_x and b_y are associated, a dot is placed at point (x, y). The association plot clearly shows that MITHRIL not only discovers sequential block associations, denoted by the diagonal in the graph, but also many non-sequential block associations.

As mentioned earlier, MITHRIL is designed to catch the mid-frequency blocks since frequent blocks are captured by the underlying caching layer and rare blocks are by definition not worth chasing after. Figure 9b and Figure 9c show the hit count obtained by LRU and MITHRIL; the horizontal axis is sorted by the frequency of blocks in the original trace. LRU gets cache hits on most of the frequent blocks (left part of the figure). For mid or low frequency blocks, LRU shows a bushy image because whether LRU can catch a mid or low frequency block depends on if the block shows small-range locality. If a block shows small-range locality, it can be caught by LRU. For example, if a block is accessed only twice throughout the trace and the two accesses are just separated by a few requests, then it will be captured by LRU. However, if its two accesses are far away from each other, then it won't be captured by LRU. For MITHRIL, besides

high-frequency blocks being captured, mid-frequency blocks can also be captured because MITHRIL can predict its access ahead of time. As shown in the figure, MITHRIL has higher hit counts for most blocks in the mid-frequency range. These two figures illustrate the crux of why MITHRIL provides a high hit ratio: *it discovers sequential associations and non-sequential associations, capturing the mid-frequency blocks that tend to be ignored by common cache replacement policies.*



(a) *Associations discovered by* Mithril

(b) *Hit count in* LRU

(c) *Hit count in* Mithril

Figure 9: Mithril *Analysis.* *a): associations discovered by* Mithril *contains both sequential associations and non-sequential associations. The four rectangular areas in the figure may represent two major applications that interact with each other. b), c): hit count of blocks sorted by frequency in original trace illustrates* Mithril *is able to capture mid-frequency blocks, while* LRU *cannot.*

6 Conclusion

Storage systems increasingly rely on effective caching layers to sustain mounting demands for performance. We proposed a novel general history-based prefetching layer, MITHRIL, to supplement the caching layers. MITHRIL is based purely on the logical timestamp of cache requests without any extra hints, making it easy to use and integrate into existing systems. We evaluated MITHRIL on 106 week-long CP traces and 29 70-day-long MSR traces of real storage systems in terms of the hit ratio. Our experimental results suggest that MITHRIL is lightweight compared to other history-based approaches, and provides 7 × greater hit ratio over LRU and 36% greater hit ratio over AMP sequential prefetching algorithm at modest costs.

Combining effective cache replacement algorithms with MITHRIL may create a low-overhead caching strategy for capturing often overlooked mid-frequency items and bolster cache performance in today's cloud storage systems. To further explore the capabilities of MITHRIL, future work will further consider a wider range of cache replacement algorithms and evaluate the performance gain from intelligent prefetching. Finally, the MITHRIL algorithm would benefit from being self-adaptive to remove the need for optimizing parameters.

Acknowledgments

We thank Carl Waldspurger, Irfan Ahmad and others at CachePhysics for valuable discussions and feedback on the paper, as well sharing the CloudPhysics traces. We also want to thank our shepherd, Irina Calciu, and the five anonymous reviewers for their careful reading and insightful comments. Our work is supported by NSF CAREER grant CNS-1553579 and funding from Emory University.

References

[1] AMER, A., LONG, D. D., AND BURNS, R. C. Group-based management of distributed file caches. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 525–534.

[2] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1991), SOSP '91, ACM, pp. 198–212.

[3] BERGAMASCO, D. Ioblazer. https://labs.vmware.com/flings/ioblazer. Accessed: 2017-01-30.

[4] CHANG, F., AND GIBSON, G. A. Automatic i/o hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 1–14.

[5] FOURNIER-VIGER, P., GOMARIZ, A., GUENICHE, T., SOLTANI, A., WU., C., AND TSENG, V. S. SPMF: a Java Open-Source Pattern Mining Library. *Journal of Machine Learning Research (JMLR) 15* (2014), 3389–3393.

[6] GILL, B. S., AND BATHEN, L. A. D. AMP: adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX conference on File and Storage Technologies* (2007), USENIX Association, pp. 26–26.

[7] GILL, B. S., AND BATHEN, L. A. D. Optimal multistream sequential prefetching in a shared cache. *ACM Transactions on Storage (TOS) 3*, 3 (Oct. 2007).

[8] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *USENIX Annual Technical Conference, General Track* (2005), pp. 293–308.

[9] GNIADY, C., BUTT, A. R., AND HU, Y. C. Program-counter-based pattern classification in buffer caching. In *6th Symp. Operating Systems Design & Implementation (OSDI)* (Dec 2004), pp. 395–408.

[10] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *USENIX summer* (1994), pp. 197–207.

[11] GU, P., ZHU, Y., JIANG, H., AND WANG, J. Nexus: a novel weighted-graph-based prefetching algorithm for metadata servers in petabyte-scale storage systems. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on* (2006), vol. 1, IEEE, pp. 8–416.

[12] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 19:1–19:14.

[13] HAN, J., PEI, J., AND KAMBER, M. *Data mining: concepts and techniques*. Elsevier, 2011.

[14] JIANG, S., DING, X., XU, Y., AND DAVIS, K. A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 10.

[15] JIANG, S., AND ZHANG, X. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[16] KOH, Y. S., AND ROUNTREE, N. Finding sporadic rules using apriori-inverse. In *Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining* (Berlin, Heidelberg, 2005), PAKDD'05, Springer-Verlag, pp. 97–106.

[17] LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. Tap: Table-based prefetching for storage caches. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 6:1–6:16.

[18] LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. C-Miner: Mining block correlations in storage systems. In *FAST* (2004), vol. 4, pp. 173–186.

[19] LI, Z., CHEN, Z., AND ZHOU, Y. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS) 1*, 2 (2005), 213–245.

[20] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 181–197.

[21] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[22] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 17:1–17:15.

[23] SOUNDARARAJAN, G., MIHAILESCU, M., AND AMZA, C. Context-aware prefetching at the storage server. In *USENIX Annual Technical Conference* (2008), pp. 377–390.

[24] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 373–386.

[25] TENG, J. Z., AND GUMAER, R. A. Managing ibm database 2 buffers to maximize performance. *IBM Syst. J. 23*, 2 (June 1984), 211–218.

[26] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 95–110.

[27] WONG, T. M., AND WILKES, J. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), ATEC '02, USENIX Association, pp. 161–175.

[28] YANG, J. mimircache. https://github.com/1a1a11a/mimircache. Accessed: 2017-01-30.

[29] YANG, S., SRINIVASAN, K., UDAYASHANKAR, K., KRISHNAN, S., FENG, J., ZHANG, Y., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Tombolo: Performance enhancements for cloud storage gateways. In *IEEE 32nd Symposium on Mass Storage Systems and Technologies, MSST 2016* (2016).

[30] ZHOU, Y., PHILBIN, J., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 91–104.