# FussyCache: A Caching Mechanism for Emerging Storage Hierarchies

Jit Gupta and Krishna Kant
Temple University, USA
[jit.gupta|kkant]@temple.edu

Ayman Abouelwafa
HP Enterprise, USA
ayman.abouelwafa@hpe.com

***Abstract*—The emerging nonvolatile memory and storage technologies are beginning to fill the huge gap that traditionally existed between the magnetic disk drive based storage and DRAM based memory. It is now possible to have the storage hierarchies where at least some of the adjacent layers may differ only by an order of magnitude in speed. In such cases, a traditional caching mechanism that blindly caches everything from the lower level is not desirable. Furthermore, at the highest level of the storage hierarchy, the overhead of managing the DRAM cache could be comparable to the read/write latency of the storage device. Thus, it may be beneficial to only cache frequently requested data while serving other requests from the next lower level directly. At the same time, the complexity of the caching mechanism needs to be commensurate with the impact of management overhead. In this paper we propose such a mechanism called "FussyCache" that is "fussy" about deciding what to cache. The mechanism is also self-regulating in that it tracks its own performance and switches over to the native mechanism if it is doing worse and switches back when appropriate. Using the available storage system traces, we show that FussyCache for Intel Optane based storage can reduce the average latency by almost 20% compared to the native caching mechanism such as plain LRU.**

***Index Terms*—Storage Hierarchy, Caching, Non-volatile memory (NVM), 3D-Xpoint, Intel Optane**

## I. Introduction

Caching is a crucial and heavily studied subject at all levels from CPU to the storage systems. The sophistication of the caching (and associated prefetching [1]) mechanisms are governed by the overhead vs. benefit of the mechanism. For example, CPU caching tends to be very simple and geared towards doing things in parallel, for example, the way prediction [2], pseudo-associativity, stream detection, stream buffering, etc. On the storage side, the mechanisms tend to be much more sophisticated because of the high latency of traditional hard disk drive (HDD) based storage which is far higher than the latency associated with managing the caching in DRAM.

However, with the introduction of a large variety of storage technologies to fill the huge HDD-DRAM speed gap, the approach requires rethinking. In particular, some of the successive layers in a storage hierarchy such as the one shown in Fig.1 may not differ drastically in latency, and neither a traditional caching (where all referenced data is cached in the higher layer), nor traditional tiering (where the data is selectively moved to the appropriate tier) may be useful. Furthermore, when considering caching of data from a very fast storage device into DRAM or persistent memory, it is crucial to evaluate the overhead vs. benefit carefully.

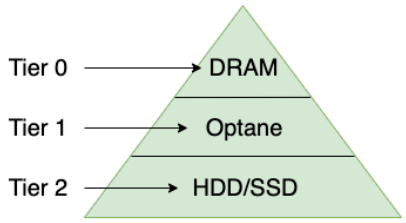


Fig. 1. Storage hierarchy

Traditional HDD storage devices have random access latencies of 4 ms or more [3], which is enormous as compared the the memory latency. For example, in contemporary DRAM, a 4KB chunk of data can be read/written in 1-3 us depending on how it maps to the internal DRAM structures (e.g., ranks, banks, rows, and columns). With SSDs increasingly forming the highest layer of the storage, the gap reduces significantly but still remains large for 4KB transfers. In particular, a fast SSD with the NVMe interface can retrieve a 4KB block in around 100-200us, or two orders of magnitude higher than the memory.

The recently released Intel Optane, and particularly the 2nd generation Optane to be released in early 2020, has a latency of only 10-20us [4], which is becoming close to memory latency ( [5]). It is also worth noting that a 4KB data transfer size is rather small, and applications often use larger transfer granularity. The gap narrows further with larger transfer sizes since all storage technologies suffer a significant initial access latency. For example, the current Samsung Evo Plus SSD can achieve read rate of 3 GB/sec and the 2nd generation Optane should also achieve similar rates. Thus the actual time required to read a 4KB block is only about 1.3us; the rest being overhead. Thus large transfers from SSDs and Optanes may reduce the gap between memory and storage latency to only a few X.

This narrowing of the gap is significant in at least two ways. First, if the higher layer latency is close to that of the lower layer latency, one needs to be more careful about whether to keep the item in the lower tier or copy or move it to the higher tier. Moving brings additional issues of writebacks and endurance; therefore, we will largely consider copying (i.e.,

caching) in this paper; however, similar arguments apply to moving as well. Second, with very fast storage, the overhead of caching mechanisms in terms of both computation and memory reads/writes becomes significant. For example, a mechanism that requires maintaining substantial amount of information about the access patterns to enable intelligent prefetching or retention could become unattractive unless the additional overhead can be more than compensated by the improvement in the hit rate. Consequently, we will largely use the popular LRU (least-recently used) caching mechanism [6] as a baseline because of its simplicity and effectiveness in practice. However, we will also evaluate some variants of LRU as well, as discussed below; however, it is important to note that sophisticated, high-overhead caching mechanisms, are less likely to be useful as the storage latency goes down. As an extreme case, note that if the storage latency was almost the same as the memory latency, there would be little point in doing the in-memory caching in the first place.

The outline of the rest of the paper is as follows. Section II discusses the related work. Section III discusses the methodology and implementation of FussyCache. Section V then discusses the experimental evaluation. Finally, section VII concludes the paper.

## II. Related Work

The traditional Least Recently Used (LRU) [6] and Least Frequently Used (LFU) [7] cache eviction mechanisms are still among the most efficient solutions with respect to the trade-off between performance and hit rate. Both of these mechanisms use the concept of bringing requested data into the cache with the least recently (least frequently in the case of LFU [7]) used data being evicted from the cache in order to make room for the incoming data. The two mechanisms are combined into an adaptive scheme by the algorithm called ARC (Adaptive Replacement Cache) ( [8]).

ARC dynamically adjusts the boundary between the recently referenced items and the most frequently referenced items and also holds the evicted items temporarily to handle their erroneous eviction. The main idea of ARC is to limit the space allocated to "one-hit-wonders". In a pure LRU mechanism, a first time request to an item will put it at the top of the stack and it may take a long time for the item to be evicted even if it is never used again. ARC initially places an item in the recency directed part of the cache, and if it is used enough number of times, it is moved to the frequency directed part. If the cache is being polluted by many one-hit-wonders, ARC will be able to get rid of them quickly because it would gradually make the space for such entries rather small.

For replacement, ACME ( [9]) is another technique similar to ARC ( [8]) that uses a virtual cache to manage cache replacement policies. ACME maintains data in caches as objects and a set of virtual caches (i.e., tables) is designed to keep past object header information for the distributed caches. However, unlike FussyCache, the virtual cache in both ARC and ACME is not designed for detection or prefetching of sequential streams.

We now look into integrated policies for caching and prefetching. Sequential Adaptive Replacement Cache (SARC) [10] builds on ARC by recognizing that the reference stream may consist of both random and sequential references. It tries to identify the sequential sub-stream and manages it separately based on recency and frequency, including prefetching of sequential data. Another prefetching caching mechanism which takes into account sequentiality while prefetching is Adaptive Multistream Prefetching (AMP) [11]. AMP considers asynchronous prefetching while adaptively changing the prefetching degree and also to trigger to prefetch during hits.

Domino [12] is another prefetching mechanism that capitalises on temporal locality by using past misses in order to predict future accesses. It uses a history table to store past prefetching trigger events and uses last two events combined with the current event so as to determine the address to be prefetched. Similar to SARC, Domino also builds on another state-of-the-art prefetching mechanism - Sampled Temporal Memory Streaming (STMS) [13].

References [14] and [15] develop integrated policies for caching and prefetching. Tombolo [16], a recently developed system, provides a solution to adapt to different workloads by combining several techniques, and thus accommodate for performance enhancements for cloud storage gateways. It takes advantage of SARC's ( [10]) effective cache replacement policy by combining GRAPH, SARC and AMP ( [11]). Tombolo uses AMP to prefetch for sequential access streams and GRAPH for random ones. We believe that this construction of several pieces makes Tombolo flexible and adaptive, however, it also brings in additional parameters that may be workload dependent.

Reference [17] proposes an I/O prefetching method with run-time and post-run analysis of applications' I/O signatures. Moreover, efficient pattern discovery and description, coupled with the observed predictability of complex patterns within many high-performance applications offers significant potential to enable many additional I/O optimizations ( [18]). References [19]–[21] classify workloads in the following categories: read/write (read only, write only, read update write, read write mix), sequentiality (sequential, 1d strided, 2d strided, variably strided), request sizes (uniform, variable). [16], building upon references [10] and [11] that address prefetching of sequential streams, differentiate between random and sequential patterns.

Majority of research related to caching has focused on attaining higher cache hit rates by making intelligent decisions corresponding to potential cache candidates. However, for the given storage hierarchy, it could make more sense to do away with the aforementioned mechanisms with a cache that does not prefetch and is not populated whenever a block of data is requested for. This is because serving random accesses from the next layer of storage maybe a more suitable approach due to its low access latency compared to previous traditional

storage systems. Only "popular" data can reside in the cache (which is the fastest storage layer) as requests for this data is not considered to be random. None of the previous work has dealt with the issue of caching from very fast devices and this paper addresses that.

## III. Implementation Methodology

### A. FussyCache Overview

Our approach consists of splitting the usual memory cache into 2 caches - a data cache (DC) and a dynamic metadata cache (DMC). Both caches are block caches i.e. data is assumed to be accessed in the LBA level, although we will treat LBAs more like chunks. The major difference between DC and DMC is that the latter does not store data corresponding to the block addresses. The DC houses only the identified popular data while the DMC stores the frequency corresponding to recently accessed blocks. The DMC also monitors the blocks that have turned popular i.e. blocks whose frequency has crossed a certain threshold value. Newly popular blocks are inserted into the DC along with the data, whereas the blocks evicted from the DC are inserted into the DMC (without the data, and the data is discarded).

DMC also keeps a check on whether the DC's hit rate is decreasing and if so it switches the DC to a traditional native cache, which implements a LRU or other simple mechanism. This is to enforce the "do-no-harm" idea. There are certain scenarios, as discussed later, where the shuffling of the items between the DMC and DC is not useful and thus ends up hurting the performance. In such cases, the mechanism automatically reverts to the native scheme. However, the DMC does continue to run in the background so as to detect whether the original mechanism can be brought to the forefront again. The targets of incoming requests are first checked if they have been marked as popular or not. If they are, their data is cached in the DC. In both cases, the metadata is entered into DMC (for new requests) or updated.

The mechanism currently treats both reads and writes identically i.e. it does not distinguish between read popular data and write popular data. This is reasonable when the read and write latencies do not differ much; however, for storage devices with substantial difference in read and write speeds, it would be useful to weight the reads and writes differently. For example, the current high end SSDs and Optane drives do not show much difference in their read/write latencies. The phase change memory (PCM) based devices, when available, are expected to have significant differences in read and write speed because of the basic nature of the operations (write performed by melting of material and its cooling, whereas read is simply a check on the conductivity properties).

### B. FussyCache Parameters

Our algorithm has the following internal parameters:

- *freqThreshold*: This parameter determines whether a certain block is popular or not. Once a certain block has been accessed more than *freqThreshold* number of times, it is declared to be popular. If aggressive caching is desired, then this parameter is set to a low value such as 3 or 4.
- *accessThreshold*: This parameter is used for two functionalities - hit-rate derivative calculation and population of the DC. The former deals with detecting whether a traditional mechanism is desired over the current mechanism for the given workload while the latter is applicable to when the DMC performs a check on all its contents in order to identify new candidates for DC. Every time the number of processed requests crosses *accessThreshold*, the two mentioned events are triggered.
- *dmcsize*: This determines the size of the DMC. It is calculated during the warmup phase.
- *hrCount*: hrCount is used to determine the number of consecutive times the DMC allows the hitrate to deteriorate. Beyond which it reverts to a traditional LRU cache. It doubles every time a switch is made between the caches. Our implementation starts off with a value of 5 i.e. it checks every *accessThreshold* number of requests whether the hit-rate has deteriorated for 5 consecutive times before it can make the first switch.
- *sleepTimer*: This parameter decides how long the DMC thread sleeps and is dependant on the *accessThreshold* parameter.

### C. Dynamic Metadata Cache

The contents of the Dynamic Metadata Cache (DMC) are characterized by block numbers and their frequency value (defining the number of times each of these blocks have been accessed). No data corresponding to the blocks are part of the DMC. However, data corresponding to the requests directed at the DMC, are fetched from the next layer of storage while in the meantime, its presence in the DMC is checked. If it is already present, the frequency of the corresponding block is incremented, else the block is inserted into the cache along with a frequency of 1 as this is the first time it has been accessed in the given recent past. In short, the DMC consists of unpopular blocks that have been accessed recently. The frequency of these recently accessed blocks are observed so as to see if any of them turn popular. Old, least accessed blocks in the DMC make way for newly accessed blocks.

While serving requests, the DMC performs two other operations for every *accessThreshold* number of accesses:

- *Identification*: The DMC checks the frequency of every block present in it and if any of them crosses the *freqThreshold* then that block's data is fetched from the storage and inserted into the DC.
- *Cache Change*: The DMC also keeps a check on the hit-rate of the DC. It checks if the hit-rate at the current check is smaller than the one during the previous check. If this check is satisfied for *hrCount* number of times, the entire caching mechanism shifts to a traditional LRU

i.e. all DC misses are brought into the DC. The DMC keeps checking the popularity of incoming blocks so as to observe whether the hit-rate to the DC increases for *hrCount* number of times. If it does, it switches back to the previous methodology. Even if it doesn't it still switches back to the original mechanism after the updated *hrCount* number of times. This is because the hitrate may have fallen because of a change in workload and so FussyCache is introduced again to see if it makes any gains. This switching back and forth does not result in an exorbitant cost because as the parameter *hrCount* is doubled during a switch, the probability of switching back in the near future lessens.

The given pseudocode for the DMC in Algorithm 1 assumes that the DMC (implemented as an array of nodes) has its own *count* data member that measures the number of blocks present in it and the DC cache has a data member calculating the number of accesses to it. *hitrate* and *hrMeasure* measure the hitrate of the DC and the count of the number of intervals for which the hitrate has deteriorated respectively. *LRU* stores whether a switch to LRU has been made. Finally, *accesses* keeps a measure of the total number of accesses that have been processed till now.

### D. Data Cache

The Data Cache (DC) is treated like a traditional LRU cache in terms of its operation. Incoming blocks are placed at the head of the LRU queue and evictions happen only on the basis of the least recently used block. However, not all blocks that are requested for are inserted into the DC. If a certain block is not identified as popular by the DMC, then it remains in the DMC until it is accessed *freqThreshold* number of times. Evictions from the DC are not treated like a conventional LRU. If it is a block that has been written to, then it is written back to the storage device. However, irrespective of a read or write, details of an evicted block from the DC is stored in the DMC. This gives it a chance to be popular again because a block that was popular in the recent past may turn popular again in the near future.

## IV. Implementation of FussyCache

### A. Warmup

As seen up till now, FussyCache is dependent on cache blocks turning popular before they are brought into the DC. If this mechanism starts at time 0, then all blocks will require a certain amount of time before they turn popular. Hence, even though a popular block may be requested for, it may still be fetched from the storage device itself because it does not reside in the DC yet. This leads us to the *Warmup* phase which gives the mechanism a head start so as to populate the DC and tackle the mentioned problem. In our implementation we have used the first two thousand accesses for every trace used for this warmup phase. The *Warmup* phase is also used for calculating

**Algorithm 1** DYNAMIC METADATA CACHE

```
1:  hitrate ← 0
2:  hrMeasure ← 0
3:  DC.accesses ← 0
4:  LRU ← false
5:  while true do
6:    sleep ( sleepTimer )
7:    if accesses < accessThreshold then
8:      if LRU then
9:        for i ← 1 to DMC.count do
10:         if DMC.nodes[i].frequency>freqThreshold then
11:           popularBlocks ← popularBlocks + 1
12:         end if
13:       end for
14:       if popularBlocks>DMC.count / 2 then
15:         hrCount ← hrCount * 2
16:         hitrate ← 0
17:         hrMeasure ← 0
18:         DC.accesses ← 0
19:         LRU ← false
20:       end if
21:       if hrMeasure==hrCount then
22:         hrCount ← hrCount * 2
23:         hitrate ← 0
24:         hrMeasure ← 0
25:         DC.accesses ← 0
26:         LRU ← false
27:       end if
28:     else
29:       for i ← 1 to DMC.count do
30:         if DMC.nodes[i].frequency<freqThreshold then
31:           Enqueue(DMC.nodes[i])
32:           DMC.count ← DMC.count + 1
33:         end if
34:       end for
35:       hrTemp ← hitrate
36:       hitrate ← DC.accesses/accesses
37:       if hrTemp<hitrate then
38:         hrMeasure ← hrMeasure + 1
39:       else
40:         hrMeasure ← 0
41:       end if
42:       if hrMeasure==hrCount then
43:         hrCount ← hrCount * 2
44:         hitrate ← 0
45:         hrMeasure ← 0
46:         DC.accesses ← 0
47:         LRU ← true
48:       end if
49:     end if
50:   end if
51: end while
```

the *dmcsize* parameter which, as the name suggests, is the size of the DMC.

### B. Experimental Setup

We implemented our scheme on a machine with a 6 core Intel Core i5 processor and 16GB DRAM. The storage devices consisted of (a) a 64GB Intel Optane Memory, (b) a 256GB Toshiba XG5 SSD, and (c) a 256GB Samsung 970 Evo Plus SSD. The three different devices were used so as to observe the impact of different device speeds and access latencies on the performance.

Managing two caches (along with their synchronization) in the memory can increase latency because every miss to the DC results in several operations, such as fetching the requested block of data from the storage device, populating the DMC with the metadata pertaining to this block, and periodically checking the entire DMC contents. To address this, we use two different threads, each running on a different core, as shown in Figure 2. We also use a third thread to coordinate event handling between DC and DMC, the events being the operations discussed before.

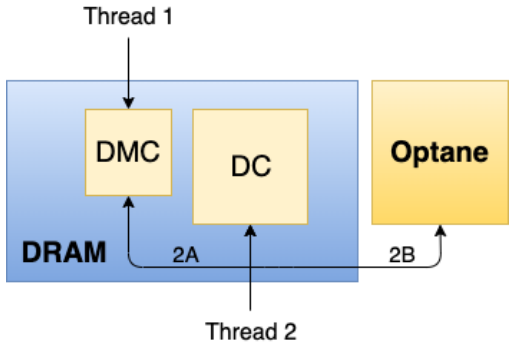


Fig. 2. Use of multithreading

*a)* **Core 1 - Thread 1::** This thread carries out two operations, namely identification of popular blocks in DMC for transfer to the DC, and switch-over from FussyCache to the traditional cache, as driven by the *accessThreshold* parameter. This thread is normally sleeping, and is awakened every time the number of accesses has crossed the parameter *accessThreshold*.After carrying out the mentioned operations, it goes back to sleep. This keeps on repeating until it is signalled by Thread 2 to stop and exit.

*b)* **Core 2 - Thread 2::** This thread handles all incoming requests. The requested block is first checked for presence in DC (via a hash table for fast lookup). If it is present, it operates the DC exactly like an LRU i.e. the requested block is placed at the top of the LRU list. If it is not present, it branches out into two threads:

- $Thread-2A$ : This thread handles device IO. In case of read, it fetches the data from the device whereas in case of a write, the data is directly written on the device without going through the DRAM cache.
- $Thread-2B$ : This thread is used to insert the requested block-descriptor into the DMC. It first checks whether it is already present in the DMC (again, using another hash table).

After processing all requests, Thread 2 signals Thread 1 to exit before exiting itself.

## V. Benchmarks and Parameters

### A. Workloads Used

For evaluating our proposed mechanism, we used the workloads provided in SPECSFS 2014 benchmark suite [22](which is a file system benchmark). We used the 2 database workloads included in the suite (DBTABLE and DBLOG) and also 2 other workloads - SWBUILD (software build workload) and VDA (Video Data Acquisition). Along with this, the SNIA Web Search workload was also tested so as to observe the results on a more realistic workload. We chose these five workloads to explore different mixtures of reads and writes.

The database table workload has a read:write ratio of 4:1 while the software build workload has a read:write ratio of 1:4. The SNIA Web search workload is a read only workload. Finally, both Database Log and VDA are write only workloads. All together, the set of workloads considered here spans a large range with respect to read/write ratios and access characteristics as further elaborated next.

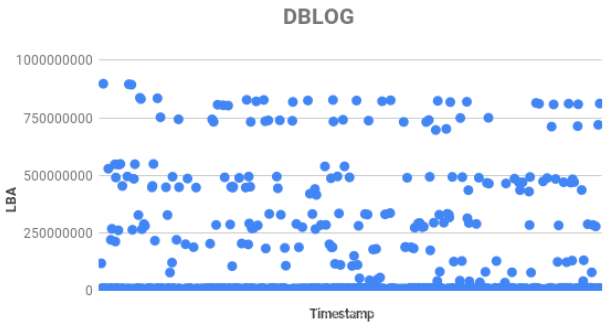


Fig. 3. Database Log

Fig. 3 shows the access pattern for Database Log workload, we can say that this workload is characterized by only a small set of LBAs accessed repeatedly with some random requests interspersed in between. We can see similar characteristics in the SNIA Web Search trace as shown in Figure 4. In Table I we can see that the former has a request size of 18kb, similar to Database Table. Also, the SNIA Web Search workload has a request size of 8kb.

The Video Data Acquisition trace in Figure 5, shows that the LBAs accessed are sequential in nature. This is largely due to the fact that this workload deals with video streams, which are largely sequential in nature. The average request size for this workload is much larger than the other workloads - 457kb (as seen from Table I and that explains the characteristics

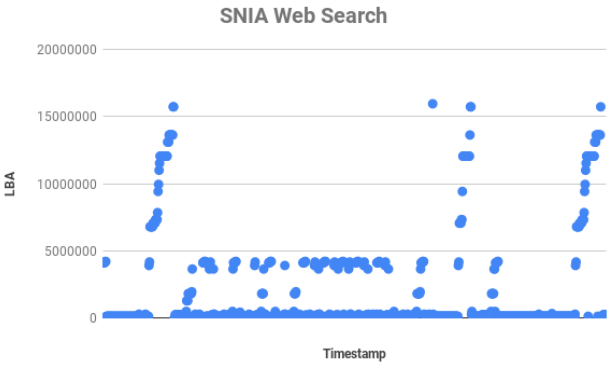


Fig. 4. SNIA Web Search

displayed in the plot. Since the SARC algorithm was designed for such workloads, it may be a better caching solution for this workload.

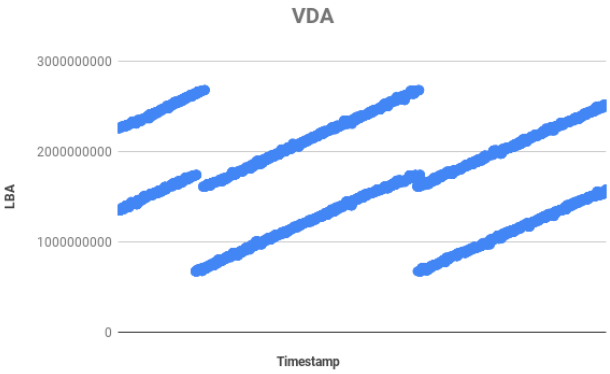


Fig. 5. Video Data Acquisition

The Database Table workload in Figure 6 shows that the accesses are much scattered compared to the previous three workloads. This workload deals with table lookups and updates and that result in a large number of LBAs being accessed. It is to be noted that this workload does not show any signs of sequentiality.

### B. Choice of Parameter Values

During the warmup phase, the size of the DMC is calculated so as to make sure that the blocks get appropriate amount of time to turn "popular" before being evicted from the cache. Also, a cache that is too large, results in more traversal cost during the *Identification* phase. The parameter *sleepTimer* is dependant on the *accessThreshold*. This is because it is decided by the time required to process that many number of accesses.

For *accessThreshold*, an optimal value needs to be chosen (i.e. process *accessThreshold* accesses before entering the *Identification* and *Cache Change* phase) to give it enough time to process the required number of accesses before starting to evict blocks that which may have turned popular during this interval. We have chosen that value to be 200 for our mechanism.

The starting point of *hrCount* value should be chosen as a not too large value (ex. 5) because an extremely high value results in the *Cache Change* taking too long to identify whether it makes sense to change the caching mechanism. The *freqThreshold* parameter which determines the popularity of a block has been tested and chosen to be 4. This implies that a block of data has to be accessed at least 4 times before it can be classified as popular. This is dependant on how aggressively one wants to populate the cache.

Automatic calculation of these parameter values can be a part of a sophisticated training phase. However, as this paper acts as a proof of concept, this is beyond the scope of this paper and can be explored in future work.

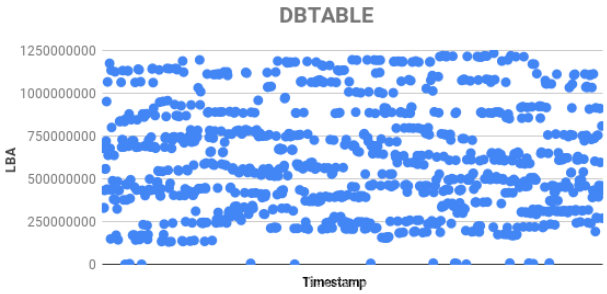


Fig. 6. Database Table

TABLE I
AVERAGE REQUEST SIZE

| Workload | | | | |
|---|---|---|---|---|
| ***DBLOG*** | ***DBTABLE*** | ***VDA*** | ***SWBUILD*** | ***SNIA*** |
| 18 | 18 | 457 | 23 | 8* |

[a]All values in kilobytes.

## VI. EVALUATION OF FUSSYCACHE

### A. Evaluation Metrics and Comparative Performance

In order to compare the caching algorithms at hand with FussyCache, we have used average read and write latency (in microseconds) to measure which one performs better for all three setups. Although we could have considered hit-rate as a metric; the primary quantity of interest in the context of storage caching are indeed read and write latencies.

Caching is an extremely well-mined area and there are numerous caching algorithms against which we could compare FussyCache. However, the main motivation for FussyCache is the need for simplicity to deal with high speed storage devices where the computational and memory access related overhead of complicated mechanisms does become significant. Therefore, we compare FussyCache against the rather simple and popular mechanisms only. We also do not compare FussyCache against mechanisms that work only for very special workload characteristics (e.g., MRU, that evicts the most recently used item [23]).

Accordingly, we consider three existing caching mechanisms for comparison: (a) Sequential Adaptive Replacement Cache (SARC) [10], (b) Adaptive Replacement Cache(ARC) [24], and (c) the Least Recently Used (LRU) [6] cache. The most widely used caching mechanism among these three is LRU due to its simplicity (both in terms of methodology and implementation) and also due to the fact that it performs well over most workloads. However, LRU does not consider frequency as a factor (LFU [7], a variant of it, does) and also does not factor in the eviction history in its policy. ARC, on the other hand, considers both recency and frequency as deciding factors and keeps a separate ghost list that contains recent evictions, thereby giving recent evictions a chance to be a part of the cache again. SARC, is a variant of ARC, which carries out prefetching by classifying blocks as random or sequential (instead of recently or frequently used). SARC has a separate ghost list too (called FreeQ) which runs on a different thread, similar to FussyCache.

The original SARC publication talks about the implementation details of the mechanism [10]. It mentions how even though both the random and sequential caches can be implemented as one list, the pros of having separate lists for both outweighs the cons. For SARC implementation in the context of FussyCache, although all three lists (sequential, random and FreeQ) can be executed on different threads, the interactions between the sequential and random lists (along with the switching between these two and the FreeQ) are higher compared to the interactions between the DC and DMC in FussyCache. Hence, it makes more sense to have only the FreeQ on a separate thread.

For the mechanisms being considered, it is expected that SARC performs well in workloads that exhibit sequentiality. ARC is expected to achieve a higher hit rate than LRU because of the fact that it considers both recency and frequency unlike LRU. However, dealing with these other factors makes ARC a more computationally heavy mechanism than LRU. Hence it is expected to perform worse than LRU.

### B. Results and Discussion

FussyCache was compared with the mentioned policies across three different setups for the mentioned workloads:

1) *Emerging NVME – ENVM*: This setup involves a first generation Intel Optane Memory as the backend storage device.
2) *Low Latency SSD – L2S2D*: In this setup we have the Samsung 970 Evo Plus SSD as the backend. It has a higher latency than Intel Optane Memory but lower than the Toshiba XG5 SSD.
3) *High Latency SSD – HLS2D*: This setup has the Toshiba XG5 SSD which has the highest access latency compared to the other two setups.

The idea of using two different SSDs lies on the basis that the difference in performance reported may be amplified due to the difference in average latency between the two SSDs in question. All the latency values are in microseconds and 4K blocks are considered. The front end for all the three setups is a 16GB DRAM.

*1) Emerging NVME - ENVM:* In this setup, our backend device is much faster compared to the subsequent *L2S2D* and *HLS2D* setups. The average read and write latency for ENVM is in the range of tens of microseconds which is almost ten times smaller than the average SSDs.

For the SNIA Web Search workload (which is a read only workload), we can see in Fig. 7 that only ARC and SARC perform better than LRU but FussyCache performs the best. This is due to the fact that FussyCache leverages the frequently accessed blocks well but at the same time serves the unpopular blocks from the backend device. ARC does better than LRU and SARC because it also considers frequency. But existing caching solutions take into account only the frontend device where the cache itself resides and cache every item accessed indiscriminately. They are not guided by the backend device's capabilities. FussyCache performs almost 15% better tan ARC and close to 25% better than LRU for this workload.

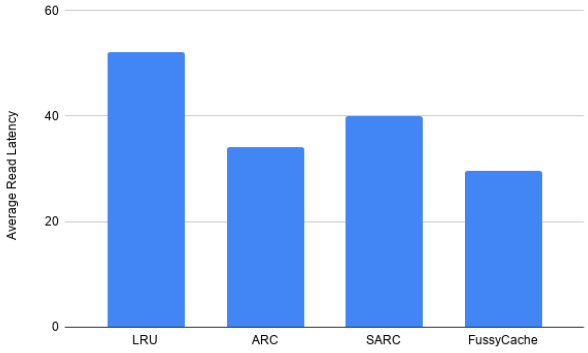


Fig. 7. Average Read Latency for SNIA Web Search Workload on ENVM

The VDA workload (a write only workload) is interesting due to its highly sequential nature. The large request size seen in Table I also shows that a prefetching algorithm that accounts for sequentiality in workloads may do well here. And that is mirrored in the results observed in Fig. 17. SARC performs better than LRU and ARC by a huge margin due to this aspect. However, FussyCache wins in this case too as recently accessed blocks have a propensity of getting accessed again in this workload i.e. it has an inclination towards requesting the same starting block with varying request sizes. Hence it outperforms ARC by almost 30% and LRU by about 20%. It beats SARC marginally in spite of its simplicity since SARC is built for workloads such as VDA which exhibit such high sequentiality.

For the Software Build workload (a write dominant workload), ARC and SARC perform similarly, with LRU again doing the best. FussyCache outperforms LRU by almost 20% for reads and close to 17% for writes. Compared to ARC and SARC, FussyCache does better by 28% for reads and as much as 30% for writes.

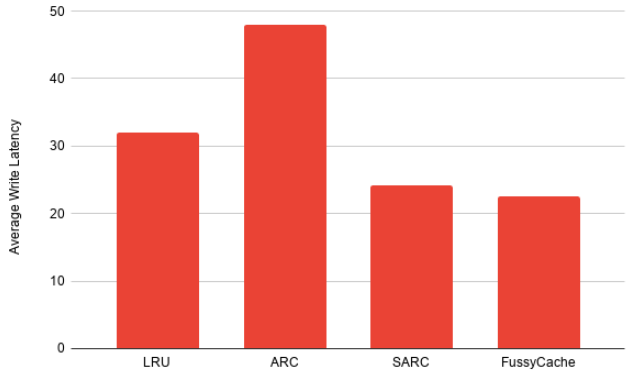


Fig. 8. Average Read/Write Latency for VDA Workload on ENVM

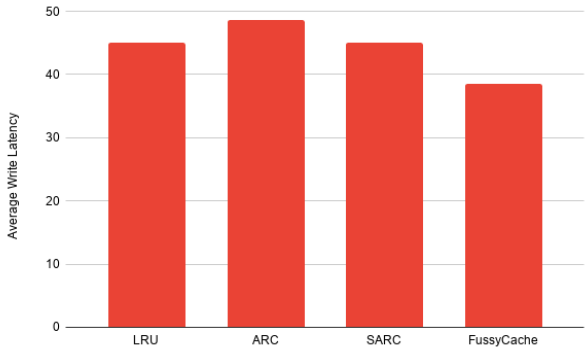


Fig. 9. Average Write Latency for DB Log Workload on ENVM

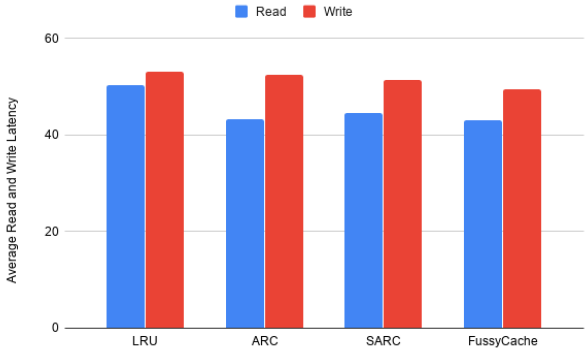


Fig. 10. Average Read/Write Latency for DB Table Workload on ENVM

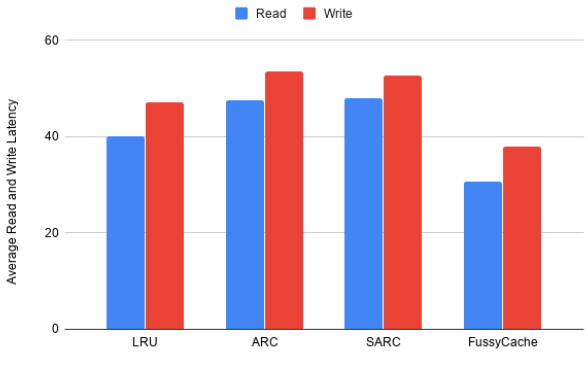


Fig. 11. Average Read/Write Latency for Software Build Workload on ENVM

For the Database Table (a read dominant workload) and Log (write only workload) workloads, FussyCache does perform better than LRU, ARC and SARC by about 5-10% for reads and 5-7% for writes. This is due to the fact that it detects that for the given workload, it cannot make as much gains as it could in other workloads and hence it occasionally shifts to a traditional LRU. This is carried out by the *Cache Change* operation in the DMC. Hence depending on the workload, FussyCache decides whether the trade-off between the "fuss" in caching versus the serving of requests from the low latency backend device makes sense. In cases where it doesn't, it shifts to the background allowing a traditional mechanism to take over, which in this case is LRU.

In the Database Log workload, which is a write only workload, FussyCache once again outperforms the other three mechanisms with LRU performing the best among them. It does better than LRU by about 13% which itself does better ARC and SARC (though it beats SARC marginally). The latter two are outperformed by FussyCache by almost 20% and 15% respectively.

*2) Low Latency SSD - L2S2D:* In this setup, our backend device is slower than Intel Optane but its read and write latency falls within the range of 90-200 microsecond [25]compared to Intel Optane where the read write latency range is between 20-50 microseconds.

We can see that even in such a setup FussyCache performs well in most cases. For example, in the VDA workload, as seen in 16, FussyCache performs better than LRU and ARC and reports latency values comparable to SARC. It makes gains over ARC by almost 15% and over LRU by 3%.

Similarly, looking at the Software Build workload in 15, FussyCache performs almost as well as LRU and even better than both ARC and SARC by about 15% in writes and close to 18% in writes. LRU, ARC and SARC mirror the behavior they exhibited between themselves for *ENVM* in this case too.

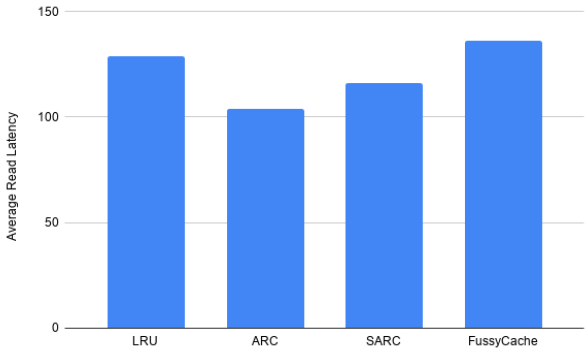


Fig. 12. Average Read Latency for SNIA Web Search Workload on L2S2D

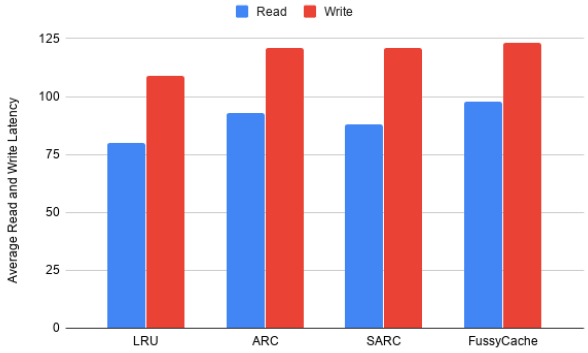


Fig. 13. Average Read/Write Latency for DB Table Workload on L2S2D

Looking at the Database Table and Database Log workloads, we can observe that similar to the reported performances for *ENVM* devices, all four of the caching mechanisms behave in a similar fashion. For Database Table, the read performance between all four caching mechanisms only differs by a maximum of 6% and for writes it is close to 8%. For Database Log however, LRU outperforms all the other mechanisms by almost 20% but the difference between ARC, SARC and FussyCache is around 5%. Thus showing that FussyCache is comparable to the other solutions at hand.

For the SNIA Web Search workload, FussyCache does almost as well as LRU with a difference of only 4% between the two. This proves that using FussyCache on lower latency SSDs can be an effective solution too as we can see that across different workloads FussyCache performs almost as well as widely used caching mechanisms and even does better in some cases. This is a significant result since we expect that future SSD offerings will generally provide even lower latencies than the current ones.

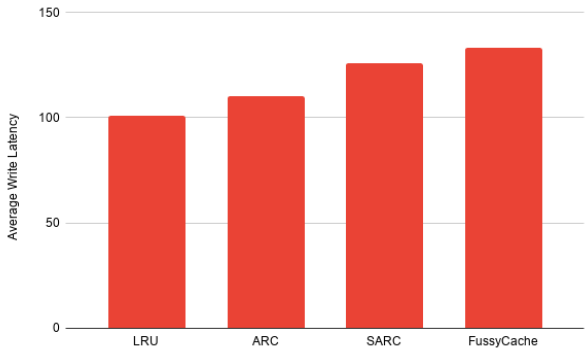


Fig. 14. Average Write Latency for DB Log Workload on L2S2D

*3) High Latency SSD - HLS2D:* This setup has a high latency SSD as its backend storage device. The Toshiba XG5 reports read and write latencies in the range of 250-400 microseconds [26], which is almost double of what we saw in *L2S2D* and close to ten times the average read write latencies reported in *ENVM*. The motivation to include such a SSD (which is rather dated) is to explore the limits of FussyCache.

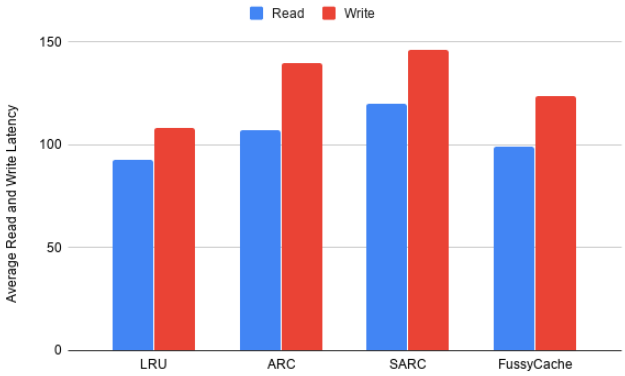


Fig. 15. Average Read/Write Latency for Software Build Workload on L2S2D

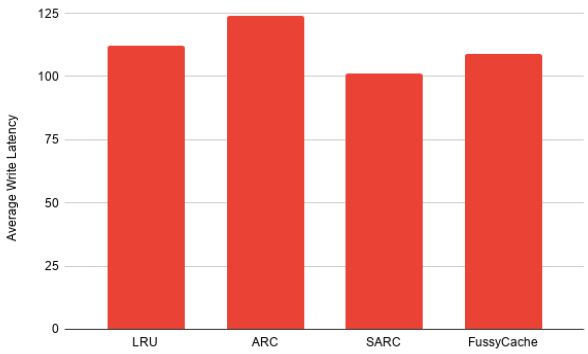


Fig. 16. Average Write Latency for VDA Workload on L2S2D

Obviously, we expect FussyCache to perform worse with such high latencies since it is designed expressed for the emerging low latency devices.

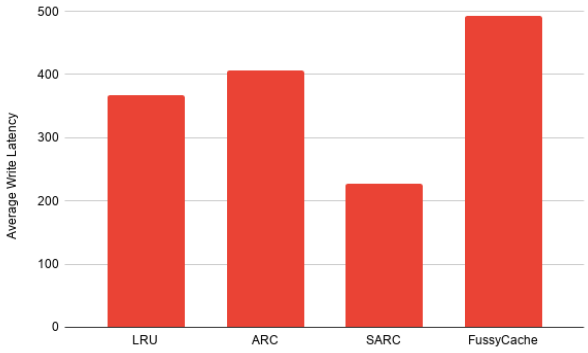


Fig. 17. Average Write Latency for VDA Workload on HLS2D

Fig. **??** confirms that in this case FussyCache is outperformed by the other mechanisms. However, it still manages to perform comparably to LRU, ARC and SARC. There is only a difference of 10% with ARC while ARC itself performs better than LRU and SARC by 6% and 4% respectively.

Similarly, if we look at the Database Log workload in Fig. 19, we can see that FussyCache performs better than SARC by about 12%. LRU, however, performs the best in this case,

even better than ARC by about 10%. In the Software Build workload we can see a change in behavior where FussyCache performs better than SARC and ARC when it comes to writes (by about 6% and 12% respectively). The read latency reported is comparable to that of SARC and ARC. In this case too we can see that LRU performs better than the other mechanisms, though marginally. The reason that LRU seems to perform decently across systems and workloads is because of its low computational overhead even though it cannot leverage the frequency and sequentiality.

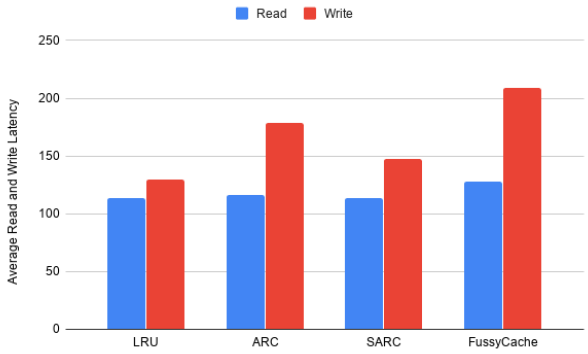


Fig. 18. Average Read/Write Latency for DB Table Workload on HLS2D

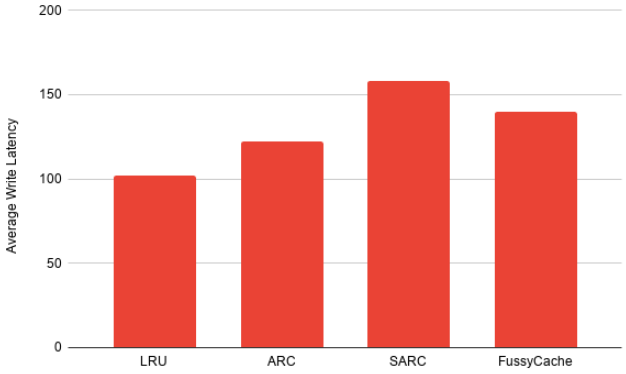


Fig. 19. Average Write Latency for DB Log Workload on HLS2D

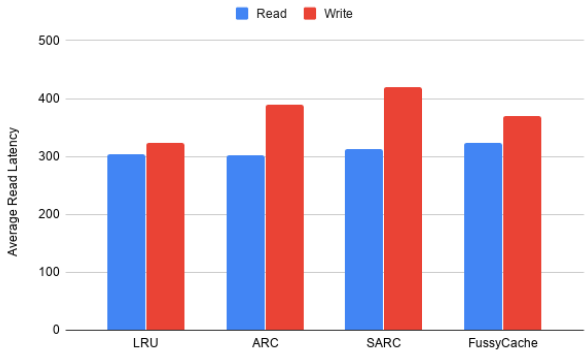


Fig. 20. Average Read/Write Latency for Software Build Workload on HLS2D

For the Database Table and VDA workloads we can see that FussyCache's performance drops but it does so to a level that is still comparable to the other mechanisms. SARC again performs the best in the VDA workload.

Throughout the three setups, we can see that SARC does not particularly work well except for workloads that exhibit sequential characteristics. The added complexities that it introduces in order to detect sequentiality and to prefetch does not provide major gains in other workloads. It should also be noted that for a mechanism that is built for systems using newer and faster storage devices (ex. Intel Optane Memory), FussyCache performs admirably when tested on other slower systems.

With HLS2D being the slowest setup and ENVM the fastest, we can see how L2S2D acts as a bridge to this enormous gap in average request latencies. As the storage device gets faster, FussyCache's performance improves too and it can be seen that for ENVM technologies, caching needs to be rethought with an eye towards the storage device and its capabilities. For example, the recently released second generation Optane is reported to have a latency of 10 microseconds, and we expect that FussyCache will perform even better for it.

## VII. Conclusions and Future Work

Previous work in caching has resulted in the introduction of numerous mechanisms that prioritize added capabilities while keeping the core functionalities of the cache same. We have argued that as the newer storage technologies with latencies in a few tens of microsecond become commonplace, we need to rethink traditional caching mechanisms themselves. In particular, cache blocks that are not very popular can be increasingly served directly from the device instead of wasting cache space on them and going through the latency of managing those entries in the cache. The key challenge here is to separate the popular from from semi-popular blocks in a light-weight manner, which is precisely what our FussyCache solution does.

We have compared FussyCache with three widely used existing caching solutions - LRU, ARC and SARC over five different workloads and shown that it performs better in all the cases for ENVM technology with almost 25-30% gains being observed. To further consolidate our hypothesis, we have tested FussyCache over two other setups, involving low and high latency SSDs to show that even though the performance does not match that of the ENVM setup (due to the increase in the average access latency), it still does a decent job in competing with existing caching solutions.

Although we did not explicitly study the endurance impact of FussyCache in this paper, the endurance should not be adversely affected by much due to FussyCache mechanism. The reason is that mostly the less unpopular (e.g., isolated) writes will go directly to the device; the popular items that receive many writes will still be updated in DRAM and then written back eventually to the device.

In the future, we plan to make FussyCache entirely autotunable by automatically learning the appropriate parameters, much like what we have done in the past for our BeliefCache [27]. Incidentally, our Belief Cache was motivated by the exact opposite consideration than FussyCache – in BeliefCache, we maintain correlations across blocks, which is rather heavy-duty but pays off in environments where the access latencies are large (e.g., accessing data from the cloud). BeliefCache has built-in mechanisms to not only learn its hyperparameters initially, but also adapt them if the workload characteristics change so drastically that the original parameters are no longer valid. There is abundant literature that points out to the existence and importance of such phase changes. [28], [29] and [30] try to identify workload phases from live storage traces. We shall examine how these mechanisms can be adapted for FussyCache in order to make it truly autonomic.

Cache Change (i.e., switching from partial caching to full caching and back) is an important component of FussyCache to ensure that it "does no harm". We will examine how to make this aspect more sophisticated without adding significant overhead due to switching or due to maintenance of relevant statistics.

**Algorithm 2** Read Request ( R , D )

$D \leftarrow \{D_1, D_2 \ldots D_N\}$ {set of devices}
$D_W \leftarrow$ *Set of warning devices*
$D_C \leftarrow$ *Set of considered devices*
**for** $e$ in $D$ **do**
  check warning status
  **if** *warning status* **is** *set* **then**
    $D_W \leftarrow D_W \cup e$
  **end if**
**end for**
$D_C \leftarrow D - D_C$
**if** $D_C == \varnothing$ **then**
  **for** $e$ in $D$ **do**
    Call *NonDeterministic procedure* **for** $e$
  **end for**
**else**
  **for** $e$ in $D_C$ **do**
    $availability_e \leftarrow \frac{DTWINReadCount_e - UsedReadCount_e}{DTWINReadCount_e}$
    $SelectDevice \leftarrow indexofmax(availability)$
    **if** $D_W \neq \varnothing$ **then**
      **for** $e$ in $D_W$ **do**
        Call *NonDeterministic procedure* **for** $e$
      **end for**
    **end if**
  **end for**
**end if**
**return** Select Device

## ACKNOWLEDGMENT

This research was supported by NSF grant IIP-1439672 and partly by HP Enterprise. The authors would like to thank Tanaya Roy and Madhurima Ray, both Ph.D. students, for their help in the experimental setup and discussions.

## REFERENCES

[1] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, no. 12, pp. 7–21, 1978.
[2] M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 54–65.
[3] A. L. Shimpi, "Western digital's new velociraptor vr200m: 10k rpm at 450gb and 600gb." [Online]. Available: https://www.anandtech.com/show/3636/western-digitals-new-velociraptor-vr200m-10k-rpm-at-450gb-and-600gb
[4] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.
[5] C. Mellor, "Intel gen 2 optane dc ssds are at least 50 per cent faster and keep latency low." [Online]. Available: https://blocksandfiles.com/2019/09/26/intel-alder-stream-optane-ssd-performance/
[6] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.
[7] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.
[8] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3. Washington, D.C.: USENIX, 2003, pp. 115–130.
[9] İ. Arı *et al.*, "Acme: Adaptive caching using multiple experts," 2002, proceedings in Informatics 14.
[10] B. S. Gill and D. S. Modha, "Sarc: Sequential prefetching in adaptive replacement cache." Washington, D.C., 2005, p. 293–308, Proceedings of USENIX ATC.
[11] B. S. Gill and L. A. D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache." Washington, D.C., 2007, p. 185–198, Proceedings of FAST.
[12] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 131–142.
[13] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 79–90.
[14] R. H. Patterson *et al.*, "Informed prefetching and caching," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. Copper Mountain, Colorado, USA: SOSP '95, 1995, pp. 79–95.
[15] S. F. Kaplan *et al.*, "Adaptive caching for demand prepaging," in *ACM SIGPLAN Notices*, vol. 38, ACM. New York, NY: ACM, 2002, pp. 114–126.
[16] S. Yang *et al.*, "Tombolo: Performance enhancements for cloud storage gateways," in *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST 2016)*, 2016.
[17] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel i/o prefetching using mpi file caching and i/o signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press. Hoes Lane Piscataway, NJ: IEEE, 2008, p. 44.
[18] J. He *et al.*, "I/o acceleration with pattern detection," in *Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2013, pp. 25–36.
[19] T. M. Madhyastha and D. A. Reed, "Input/output access pattern classification using hidden markov models," in *Proceedings of the fifth workshop on I/O in parallel and distributed systems*. ACM, 1997, pp. 57–67.
[20] ——, "Learning to classify parallel input/output access patterns," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 8, pp. 802–813, 2002.

[21] T. M. Madhyastha, "Automatic classification of input/output access patterns," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.
[22] S. SPEC, "Benchmark, 2014."
[23] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Algorithmica*, vol. 1, no. 1-4, pp. 311–336, 1986.
[24] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.
[25] B. Tallis, "The samsung 970 evo plus (250gb, 1tb) nvme ssd review: 92-layer 3d nand." [Online]. Available: https://www.anandtech.com/show/13761/the-samsung-970-evo-plus-ssd-review/3
[26] ——, "The toshiba xg5 (1tb) ssd review." [Online]. Available: https://www.anandtech.com/Show/Index/11663?cPage=4&all=False&sort=0&page=2&slug=the-toshiba-xg5-1tb-ssd-review
[27] D. Ramljak, D. Abraham, K. Kant, and D. Voigt, "Modular framework for data prefetching and replacement at the edge," *Proc. of IEEE Edge Computing, Seattle, WA*, June 2018.
[28] D. Gu and C. Verbrugge, "A survey of phase analysis: Techniques, evaluation and applications," Technical Report SABLE-TR-2006–1, Tech. Rep., 2006.
[29] J. Zhang, M. Yousif, R. Carpenter, and R. J. Figueiredo, "Application resource demand phase analysis and prediction in support of dynamic resource provisioning," in *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, IEEE. Hoes Lane Piscataway, NJ: IEEE, 2007, pp. 12–12.
[30] P. Pipada, A. Kundu, K. Gopinath, C. Bhattacharyya, S. Susarla, and P. Nagesh, "Loadiq: Learning to identify workload phases from a live storage trace." in *HotStorage*. Washington, D.C.: USENIX, 2012.