FussyCache: A Caching Mechanism for Emerging Storage Hierarchies[1]

Abstract

With the emerging high speed storage devices, it is no longer desirable to adopt a DRAM caching scheme that carefully discriminate what should be brought into the cache and what should be accessed directly from the device as needed. In this paper we propose such a mechanism and 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

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.

The recently released Intel Optane, and particularly the 2nd generation Optane to be released in early 2020, has a latency of only 10-20us [1], which is becoming close to memory latency ([2]). 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.

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 [3] 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.

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 IV then discusses the experimental evaluation. Finally, section VI concludes the paper.

II. Related Work

Caching is an extremely rich area with numerous algorithms; the most popular ones being LRU and its variants such as LFU, ARC (Adaptive Replacement Cache) ([4]), ACME ([5]), and Sequential Adaptive Replacement Cache (SARC) [6]. Other related algorithms include Adaptive Multistream Prefetching (AMP) [7], Domino [8], Sampled Temporal Memory Streaming (STMS) [9], Tombolo [10], etc.

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,

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 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. Benchmarks and Parameters

A. Workloads and Parameters

For evaluating our proposed mechanism, we used the workloads provided in SPECSFS 2014 benchmark suite [11](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.

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.

TABLE I
AVERAGE REQUEST SIZE IN KB

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

V. Results and Discussion

We compare FussyCache against three existing caching mechanisms for comparison: (a) Sequential Adaptive Replacement Cache (SARC) [6], (b) Adaptive Replacement Cache(ARC) [12], and (c) the Least Recently Used (LRU) [3] 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 [13], 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.

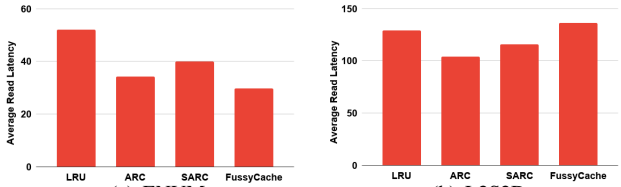
For the mechanisms being considered, it is expected that SARC performs well in workloads that exhibit sequentiality. ARC is expected to achieve a higher hit rate than LRU because of the fact that it considers both recency and frequency unlike LRU. However, dealing with these other factors makes ARC a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

FussyCache was compared with the mentioned policies across two 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.

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 on ENVM(which is a read only workload), we can see in Fig. 1 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.

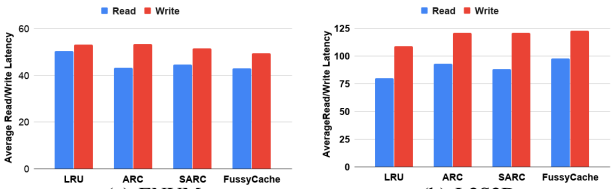


(a) ENVM (b) L2S2D

Fig. 1. Average Read Latency for SNIA Web Search Workload

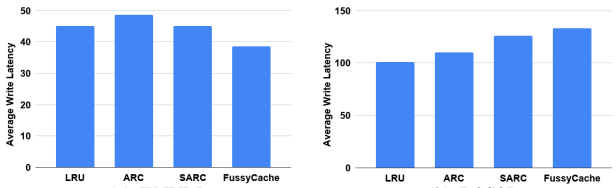
The VDA workload (a write only workload) is interesting due to its highly sequential nature. The large request size seen in Table I also shows that a prefetching algorithm that accounts for sequentiality in workloads may do well here. And that is mirrored in the results observed for ENVM in Fig. 5. SARC 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



(a) ENVM (b) L2S2D

Fig. 2. Average Read/Write Latency for DB Table Workload



(a) ENVM (b) L2S2D

Fig. 3. Average Write Latency for DB Log Workload

to ARC and SARC, FussyCache does better by 28% for reads and as much as 30% for writes.

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.

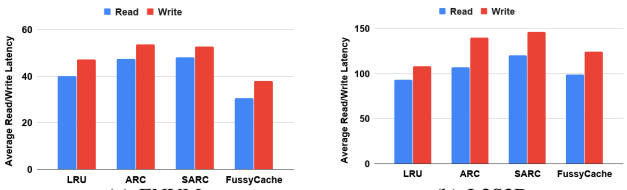
2) Low Latency SSD - L2S2D: In this setup, our backend device is slower than Intel Optane but its read and write latency falls within the range of 90-200 microsecond [14]compared to Intel Optane where the read write latency range is between 20-50 microseconds.

We can see that even in such a setup FussyCache performs well in most cases. For example, in the VDA workload for L2S2D, as seen in 5, 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 for L2S2D in 4, 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.

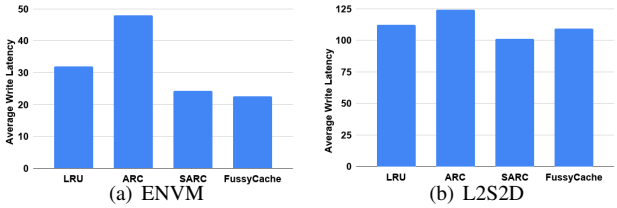
Looking at the Database Table and Database Log workloads, we can observe that similar to the reported performances for *ENVM* devices, all four of the caching mechanisms behave in a similar fashion. For Database Table, the read performance between all four caching mechanisms only differs by a maximum 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.



(a) ENVM (b) L2S2D

Fig. 4. Average Read/Write Latency for SWBUILD workload



(a) ENVM (b) L2S2D

Fig. 5. Average Write Latency for VDA Workload on ENVM and L2S2D

VI. Discussion

In this paper we argue that as the newer storage technologies with latencies in a few tens of microsecond become commonplace, it becomes more desirable and efficient to serve semi-popular cache blocks can be served directly from the device rather than paying for the overhead of cache

management. 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. The mechanism does similar or better job compared with traditional algorithms even for fast SSDs.

It is important to explore the endurance impact of FussyCache even though our assumption is that 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.

FussyCache has several parameters whose suitable setting can be tricky. Thus, to be useful, it is important to make FussyCache entirely auto-tunable. This requires automatted learning of the appropriate parameters and adaptation to phase change in the workload.

In reference [15], the authors describe a caching mechanism called BeliefCache that not only learns its hyper-parameters initially, but also adapts them if the workload characteristics change so drastically that the original parameters are no longer valid. Given the need for keeping the mechanism lightweight, it is still necessary to examine whether 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.

We would like to know what could be lightweight and effective solutions to the two problems we have discussed above as the current values of the parameters are assumed to be static. Also, the switching between mechanisms so as to adopt the "no-harm policy" is another aspect to which we are open to ideas. We would like to get ideas on how to make the mechanism automatically revert to LRU if deployed on a slow device.

[2] 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/

References

[1] 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.

[3] 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.

[4] 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.

[5] İ. Arı *et al.*, "Acme: Adaptive caching using multiple experts," 2002, proceedings in Informatics 14.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] S. SPEC, "Benchmark, 2014."

[12] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[13] 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.

[14] 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

[15] 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.