Analysis of Cache Replacement Policies

Aoran Wu
University of Wisconsin-Madison

Brian Chang
University of Wisconsin-Madison

Yuhao Zhang
University of Wisconsin-Madison

Abstract

In this project, we aim to analyze the performance of different cache replacement policies under various workloads. We leverage the functionality of eBPF-based tracing tools, namely bpftrace, to collect page cache access information from the Linux kernel. We implement a set of simulators that take the collected traces as input and mimic different replacement policies. We demonstrate the key results of the simulation and show the effectiveness of different policies.

1 Introduction

Cache has been playing an important role in today's computer systems. A cache hit is typically several orders faster than a cache miss for data access, so an appropriately designed cache replacement policy can significantly improve the system's performance. A lot of studies [8, 10, 12] have proposed effective cache replacement policies.

In order to understand how different policies work, it is crucial to understand how they operate under various workloads. We do so by collecting page cache information with eBPF [6]-based tracing technologies. eBPF is an event-driven framework that allows developers to modify and observe kernel functionality. With eBPF, we can collect kernel events, namely page cache accesses, under different workloads. We use these collected traces and run simulations of the traces under different replacement policies. In this report, we evaluate the performance of LRU, ARC, and 2Q on traces collected from real-world workloads. We also compare their performance with that of an optimal policy OPT [3], aiming to see how effective these cache replacement policies can be in real-world systems.

The rest of this report is organized as follows. We first introduce the background of our project in Section 2. We describe our experimental design in Section 3 and environment setups in Section 4. We report our experimental findings in Section 5. Related work is described in Section 6, and we conclude this project in Section 7.

2 Background

This section introduces the background. We first introduce different cache replacement policies, including the optimal policy, LRU, LFU, 2Q, and ARC. Then we give a brief introduction to eBPF, which we used to collect the traces. At last, we introduce the page cache in Linux.

2.1 Different Cache Replacement Policies

Cache replacement algorithms have always been an active research area over the years. Different replacement algorithms utilize different data structures and policies that try to minimize cache misses. We briefly describe several replacement policies, including the optimal strategy, and point out their key features.

Optimal Policy (OPT) The optimal cache replacement policy *looks into future* and replaces the entry that will not be used for the longest period in the future. This policy is optimal because if we evicted any other entries, this evicted entry would be needed relatively earlier (i.e., not the longest) in the future, thus triggering an earlier cache miss. The optimal policy is also known as the Belady optimal algorithm [3].

Least Recently Used (LRU) The LRU replacement policy removes the least recently used entry first. It keeps track of the time accessed of each entry in the cache, and when a new entry is to be inserted, LRU scans the cache and finds the entry with the furthest access time to remove.

Least Frequently Used (LFU) The LFU replacement policy removes the least frequently used entry first. It keeps track of the access frequency of each entry in the cache. When a new entry is to be inserted, LFU scans the cache and finds the entry least used to remove.

2Q 2Q [8] is one of the variants of LRU. It improves LRU/k by admitting only hot pages to the buffer instead of cleaning cold pages. 2Q maintains one LRU queue Am and two FIFO queues $A1_{in}$ with size K_{in} and $A1_{out}$ with size K_{out}. Pages in Am and $A1_{in}$ are also in the cache, while $A1_{out}$ is a ghost cache storing only page addresses. $A1_{in}$ stores the recently accessed cold pages. $A1_{out}$ stores the older cold pages, which will be promoted to Am once hit. Am stores hot pages, which have been hit at least twice.

ARC ARC [10], short for Adaptive Replacement Cache, can be seen as a combination of the Least Recently Used and the Least Frequently Used policies. ARC maintains both a LRU (t_1) and a LFU (t_2) cache, and a *ghost cache* for both of these caches (b_1, b_2). t_1 and t_2 have a combined capacity of c, and an adjustable parameter p that sets the capacity of t_1 to p and t_2 to $c-p$. t_1 stores entries that have been accessed exactly once, and t_2 stores entries that have been accessed more than once. When an entry is evicted from t_1, it is placed in b_1 for future reference, the same operation goes for t_2. When an entry stored in the ghost cache is requested, ARC adapts p to adjust the size of t_1 and t_2. In short, the ghost caches are indicators of the current access pattern observed. When the cache is accessed in a more random or sequential way, ARC makes t_1 larger, and when the accesses are focused on several heavy hitters, ARC adapts and makes t_2 larger. Full details of the ARC replacement algorithm can be found in their original paper [10].

2.2 Introduction to eBPF

The extended Berkeley Packet Filter (eBPF) is a Linux subsystem that enables custom programs to run in a sandboxed environment in the kernel without the need of modifying/recompiling the kernel source code or loading the kernel module. An eBPF program is attached to a kernel event, which invokes the eBPF program when triggered. Once running, the eBPF program has access to the context of the event and other kernel data structures, which gives eBPF unprecedented observability and modifiability to the Linux kernel behaviors. The events which eBPF programs can hook to are all over the kernel, from `kprobe` to the eXpress Data Path (XDP). Such flexibility has led to the development of many widely used eBPF-based frameworks and applications. For instance, Cilium [1] has leveraged eBPF to implement networking and security functions for containerized services.

eBPF programs have added modifiability and functionality to the Linux kernel with it is event-driven, sandboxed execution nature. Various forms of applications have emerged along with the maturing of eBPF. One of the main categories of eBPF applications is various **tracing** applications. Tools such as bcc and bpftrace make probing the kernel much more straightforward for developers and enable deep inspection and benchmarking of complicated systems. In this project, we use bpftrace to probe the kernel of page access events. We first go over the Linux kernel source tree to find appropriate functions that provide page access information (e.g., physical page address, page index), then we use `kprobes` to hook onto the discovered function and collect page access traces on various workloads.

2.3 Page Caches in Linux

The page caches in Linux are memory areas used to keep some disk data in memory so that further access to them will not require disk I/Os and can be served directly by the memory. As disk access is much slower than that of memory access and repeated accesses to the same disk data are quite common, page cache can significantly improve the system performance. However, the page cache size is limited by the main memory size of Linux, and when the page cache is used up and new disk data are accessed, some pages in the page cache have to be evicted so other new pages can be put in. The *page cache replacement policy* is used to determine which page to evict when the page cache is full.

In Linux the page caches are shared across all processes, and are accessed with the kernel function called pagecache_get_page(). The signature of this function is:

```
  struct page * pagecache_get_page (struct
address_space *mapping,pgoff_t offset,int
fgp_flags,gfp_t gfp_mask),
```

where the first argument is the address of an address_space object, which indicates the owner of this page in the pagecache, or more specifically, to which **file** on disk this page belongs. The second argument shows the index of the page in this file. We can combine these two arguments to uniquely determine which physical page of a disk is accessed by this function call. In Section 3, we conduct some experiments to verify our understanding of this function call's semantics.

3 Design and Implementation

We first design some small experiments to see whether we correctly understand the semantics of the pagecache_get_page() kernel function. Then we use eBPF to collect the page access information of three real-world workloads of Linux. Finally, we implement different cache replacement policies to see how they perform on these workload traces collected.

3.1 Experiment with `pagecache_get_page()`

Before we collect the traces, we have to testify that the semantics of the kernel function `pagecache_get_page()` do satisfy our requirements. We have to show that the accesses to the same page of the same specific disk file do correspond to a unique combination of an `address_space` object and page index (i.e., the first and second argument of the `pagecache_get_page()` function). To testify that, we create a file and write three program to access this file. Specifically,

- The first program uses the `read` system call to read the contents of the first page of this file.
- The second program uses the `write` system call to write to the first page of this file.
- The third program uses `mmap` to map the first page of the file into the address space of this program.

To clearly show the trace, we put the call of `read`/`write`/`mmap` into an infinite loop so that the trace entry corresponding to the access of this file will appear constantly. Then we launch pairs of these three programs and kill them one by one. The result shows that these three programs, though they are different processes and thus have different address spaces, do show the same address of the `address_space` object and index when they access the same file on disk. This experiment verifies that each of the combinations of the `address_space` object and page index does represent access to a unique disk page, and thus the trace we collect with this `pagecache_get_page` function does represent the disk page access patterns and thus can be used to evaluate different cache replacement policies.

3.2 Trace Collection with eBPF

We use a tool called *bpftrace* provided in eBPF to probe the function pagecache_get_page(). As pagecache_get_page() is a kernel function, we use *kprobe* to dynamically trace its first and second argument in each invocation. The command that we use is as follows:

Listing 1: Kprobe Command

```
sudo bpftrace -e
'kprobe:pagecache_get_page
{
  if ((arg0 >> 32) != 0xDEADBEEF
    || (arg0 & 0xFFFFFFFF) != 0xCAFEBABE)
      printf("%p %d\n", arg0, arg1);
}'
```

In the command above, `arg0` is the physical page address and `arg1` is the page offset.

The results of tracing are redirected into a file on disk. However, as the file containing the tracing results is also read into the pagecache, writing to it will result in extra invocations of `pagecache_get_page`, and these invocations are not part of the workloads we are interested in and thus should not be part of the tracing results. This problem is not hard to solve as the trace result file only corresponds to one specific address space object. To solve this problem, we add an if statement to filter out the specific address space object. Since it is awkward to write a 64-bit constant for the specific address space object, we compare it by splitting into two 32-bit constant.

We collect traces under three different workloads: building Linux kernel, video format conversion, and daily use of Linux. The following is a brief introduction to each of these three workloads.

- **Linux kernel building.** In this workload, we build the latest version of the Linux kernel (v5.10) and collect traces when the kernel building is ongoing. As Linux is a large project and kernel building involves many small source file reading, this workload stresses the disk I/O and may present typical file access patterns in real-world workloads. The **kernel** trace contains 6,222,343 page accesses.

- **Video format conversion.** In this workload, we use the command `ffmpeg` to convert an MP4 file to different formats with a new resolution $1,728 \times 1,080$. The original MP4 file has resolution $1,680 \times 1,050$, duration 17:34, and size 18.2MB. We convert it to different formats MP4 (same format, but a new resolution), MKV, AVI, MOV, WMV, WebM. The **video** trace contains 391,747 page accesses.

- **Linux daily use.** In this workload, we use Ubuntu Desktop to perform some daily use cases of typical desktop users, such as web browsing, document editing, music playing, etc. The resulting traces can represent typical page access patterns for normal users in the desktop environment. The **daily** trace contains 665,985 page accesses.

Besides, we artificially create another workload **mixed** by concatenating the **video**, **kernel**, and **daily** traces.

3.3 Simulating Different Cache Policies

After collecting page traces of the aforementioned workloads, we built simulators in C++ that mimic the behaviour of different replacement policies. Our simulator first generates key via the trace information probed from the function described in Section . We form the key as a string by appending the file address of the page (`address_space *mapping`) with the page index (`pgoff_t offset`). The key is used to indicate which exact page is being accessed.

We then implement the OPT, LRU, 2Q, and ARC replacement policies. Each implemented class has a `put(key)` method that attempts to place the entry with a specific key in the cache. Since the objective of this project is to collect hit/miss rates of different policies, our `put` method does not need to log the values of the entries. For each trace file, the simulator keeps track of the total number of hits and misses, and reports the hit/miss ratio at the end.

4 Methodology

We perform all our experiments in a virtual machine in Oracle VM VirtualBox [13]. The host machine has a 1.9 GHz Intel Core i7-8650U Quad-Core processor with 16GB main memory and 256GB SSD. The virtual machine is assigned 8GB main memory, 30GB disk space, and three processor cores. The operating system of the virtual machine is Ubuntu 20.04.1 LTS with desktop support. The starting state of the system does not influence the traces we collect, so we do not take any special efforts to set the system to a specific starting state. The version of `bpftrace` we use is v0.9.4, and the traces are collected under workloads described in Section 3.2.

5 Results & Discussion

In this section, we analyze different cache-replacement algorithms by answering the following research questions:

- **RQ1:** How effective are LRU, 2Q, and ARC when using different cache sizes and running on different workloads?
- **RQ2:** How do K_{in} and K_{out} affect the effectiveness of 2Q?
- **RQ3:** How does the adaptation parameter p in ARC change during each workload?

For **RQ1**, we evaluate the hit rates of different cache replacement policies when using different cache sizes and running on four different workloads. For **RQ2**, we evaluate the hit rates of 2Q when varying K_{in} and K_{out} for three different workload and cache size combinations: **daily** + 8M, **daily** + 16M, **kernel** + 8M. For **RQ3**, we evaluate how the adaptation parameter p changes during each workload for 8M and 16M cache size.

5.1 Results

RQ1: How effective are LRU, 2Q, and ARC when using different cache sizes and running on different workloads? From the results shown in Figure 1, we can see that OPT always outperforms other cache replacement policies, and it does serve as an upper bound in terms of cache hit rate. For the other three cache replacement policies, we can see that most of the time (60.41%), ARC performs better than other policies and can adapt to different workloads. In general, 2Q performs best when the cache size is small, and LRU performs better when the cache size is large. As there are tunable hyper-parameters in 2Q, and we fix them as the settings recommended by the 2Q paper in this research question, the results may not represent the best offline performance of 2Q. Another thing worth noting is that when the cache size is large enough, all pages accessed can be held in the cache, and there will only be cold misses. In this case, LRU and ARC will reach the upper bounder as OPT, but as 2Q splits the cache capacity into multiple queues, they may not reach the upper bounder in these cases.

RQ2: How do K_{in} and K_{out} affect the effectiveness of 2Q? The effectiveness of 2Q varies a lot when tuning K_{in} and K_{out}, as shown in Figure 2. For **daily** trace and 8M cache size (Figure 2(a)), the maximum hit rate 44.50% is obtained when $K_{in} = 0.1$ and $K_{out} = 0.7$, while the minimum is about 38%. For **daily** trace and 16M cache size (Figure 2(b)), the maximum hit rate 49.06%, which even outperforms the best policy ARC's 48.89% hit rate, is obtained when $K_{in} = 0.7$ and $K_{out} = 0.3$, while the minimum is about 40%. For **kernel** trace and 8M cache size (Figure 2(c)), the maximum hit rate 83.18%, which even outperforms the best policy ARC's 79.65% hit rate, is obtained when $K_{in} = 0.1$ and $K_{out} = 0.1$, while the minimum is about 57%.

Three surface plots and their projections are quite different, indicating that it is difficult to tune the hyper-parameters K_{in} and K_{out} except for brute-force enumeration.

RQ3: How does the adaptation parameter p in ARC change during one workload? Figure 3 shows how the adaptation parameter p changes during different workloads. The track the changes of workloads described in Secion 3.2 individually (Figure 3(a), Figure 3(b), Figure 3(c)), and a synthetic workload that concatenates the three above workloads in the order of **video**→**kernel**→**daily**(Figure 3(d)).

We observe a spike in p during the video encoding workload, the converted target for that range of trace numbers was the WebM encoding format. While we are not familiar with these encoding algorithms, we can make a coarse observation that WebM encoding behaves very differently from other encoding formats and is very likely to access entries in a scanned manner since the size of the t_1 cache is extremely high. The p value of the kernel trace, on the other hand, is relatively low throughout the entire workload, which suggests that the cache access of this trace is concentrated on several hot items.

Figure 3(d) shows the change in p under the concatenated workload. By referencing Figure 3(a) through Figure 3(c), we can see that the overall trend of p matches the trend if we directly combine Figure 3(a) through Figure 3(c). This match verifies ARC's adaptive nature to different workloads, as it finds the appropriate p value under various access patterns.

5.2 Interesting Observations

We find that sometimes 2Q's hit rates do not increase when the cache size increases. For example, in **video** workload (Table 2), when the cache size increases from 2M to 4M, the hit rate decreases from 67.2429% to 67.2194%. This phenomenon may be caused by the FIFO inside 2Q and known as the *Belady's Anomaly* [4].

Table 1: **Kernel** Trace: Detailed Hit Rates according to Different Cache Sizes

Cache Size	LRU	2Q ($K_{in} = 0.2c$, $K_{out} = 0.5c$)	2Q ($K_{in} = 0.3c$, $K_{out} = 0.5c$)	ARC	OPT
1M	0.227637	**0.258965**	0.256962	0.248761	0.415670
2M	0.293489	**0.349724**	0.341951	0.325871	0.533562
4M	0.423832	**0.524968**	0.506229	0.462779	0.700427
8M	0.749505	0.788861	0.770774	**0.796509**	0.871050
16M	0.889932	**0.906478**	0.905170	0.905083	0.912094
32M	0.901160	0.910751	0.910817	**0.910975**	0.916462
64M	0.907797	0.911046	0.911195	**0.913743**	0.924361
128M	0.915153	0.911314	0.911410	**0.919152**	0.938652
256M	0.928789	0.932479	0.932345	**0.933251**	0.949184
512M	**0.956054**	0.943319	0.944347	0.954355	0.956085
1024M	**0.956085**	0.946638	0.950922	**0.956085**	0.956085
2048M	**0.956085**	0.951819	0.955673	**0.956085**	0.956085

Table 2: **Video** Trace: Detailed Hit Rates according to Different Cache Sizes

Cache Size	LRU	2Q ($K_{in} = 0.2c$, $K_{out} = 0.5c$)	2Q ($K_{in} = 0.3c$, $K_{out} = 0.5c$)	ARC	OPT
1M	0.657001	0.632380	0.646627	**0.657266**	0.675454
2M	0.666384	0.664197	**0.672429**	0.667347	0.680878
4M	0.668587	0.672158	**0.672194**	0.671227	0.688720
8M	**0.675216**	0.674098	0.673695	0.673841	0.703163
16M	0.676633	0.675576	**0.676888**	0.677057	0.729302
32M	0.677897	0.678254	0.678619	**0.715326**	0.752853
64M	0.715367	0.722446	0.723102	**0.725300**	0.752853
128M	**0.738027**	0.726977	0.728194	0.738007	0.752853
256M	**0.752805**	0.737175	0.737249	**0.752805**	0.752853
512M	**0.752853**	0.737313	0.737397	**0.752853**	0.752853
1024M	**0.752853**	0.737744	0.745800	**0.752853**	0.752853
2048M	**0.752853**	**0.752853**	**0.752853**	**0.752853**	0.752853

Another interesting phenomenon that we observed is that in Figure 3(c), spikes happen at different time points in the 8M simulation and the 16M simulation. We think that this is because the cache takes longer to fill up for a bigger cache. Therefore spike would also take longer to appear. We observe that if we shift the 8M-plot a little to the right, the shifted trend of the 8M-plot matches that with the 16M-plot.

6 Related Work

Different cache replacement policies. Among all the different cache replacement policies, Belady's algorithm [3] has been proved to be optimal when considering cache hit rates. In MIN, the page which will be accessed farthest away in the future will be selected for replacement. The problem with this policy is that in real-world systems, it is always hard (if not impossible) to get page access information about the future. However, the optimality of this policy naturally provides an upper bound of cache hit rates against which other cache replacement policies can be evaluated.

While it is hard to get complete information about the future, information about the past is always available. A popular policy called LRU [5, 9] is proposed based on the assumption that the recent past is a good approximation of the near future. With this assumption, in LRU, the least recently used page will be selected for eviction. LRU and some of its variants have proved to be effective in many real-world systems [11, 12].

2Q [8] is one of the variants of LRU, and it is a low-overhead high-performance buffer management replacement algorithm. 2Q improves LRU/k by admitting only hot pages to the buffer instead of cleaning cold pages. 2Q is evaluated on both artificially generated traces and real data collected from a commercial DB2 database application. However, in our project, we plan to use traces of various workloads collected by eBPF. In the 2Q paper, the evaluations fix K_{out} to $0.5c$ and vary K_{in} from $0.2c$ to $0.3c$, where c is the size of the cache. Besides the fixed settings, we explore different hyper-parameter settings to show how K_{out} and K_{in} affect the effectiveness of 2Q in our project.

ARC [10], short for Adaptive Replacement Cache, is a dy-

Table 3: **Daily** Trace: Detailed Hit Rates according to Different Cache Sizes

Cache Size	LRU	2Q ($K_{in} = 0.2c$, $K_{out} = 0.5c$)	2Q ($K_{in} = 0.3c$, $K_{out} = 0.5c$)	ARC	OPT
1M	0.261252	0.272002	0.270916	**0.312453**	0.377941
2M	0.330902	0.360370	**0.362468**	0.356041	0.413002
4M	0.356088	**0.386623**	0.385249	0.380243	0.457788
8M	0.421371	**0.442328**	0.438187	0.441951	0.506999
16M	0.484473	0.486239	0.486003	**0.488927**	0.548578
32M	0.518307	0.509731	0.514363	**0.528706**	0.577941
64M	0.534335	0.526201	0.526321	**0.548169**	0.606764
128M	0.554233	0.544568	0.547140	**0.566297**	0.621517
256M	0.580357	0.553027	0.555107	**0.594284**	0.621517
512M	0.615666	0.563502	0.572498	**0.618948**	0.621517
1024M	**0.621517**	0.584222	0.593827	**0.621517**	0.621517
2048M	**0.621517**	0.605096	0.613460	**0.621517**	0.621517

Table 4: **Mixed** Trace: Detailed Hit Rates according to Different Cache Sizes

Cache Size	LRU	2Q ($K_{in} = 0.2c$, $K_{out} = 0.5c$)	2Q ($K_{in} = 0.3c$, $K_{out} = 0.5c$)	ARC	OPT
1M	0.253817	**0.280251**	0.279207	0.276554	0.426198
2M	0.316977	**0.367620**	0.361611	0.346980	0.530460
4M	0.430805	**0.520232**	0.504092	0.466351	0.677600
8M	0.715489	0.750985	0.735125	**0.757278**	0.828712
16M	0.841363	**0.855610**	0.854540	0.854364	0.869003
32M	0.854123	0.861555	0.862054	**0.865476**	0.876690
64M	0.863278	0.865692	0.865865	**0.870092**	0.886078
128M	0.872605	0.867845	0.868227	**0.875753**	0.899642
256M	0.887445	0.887257	0.887337	**0.889620**	0.908644
512M	**0.913981**	0.897487	0.899194	0.906998	0.914543
1024M	**0.914543**	0.902243	0.907217	0.912867	0.914543
2048M	**0.914543**	0.909394	**0.913453**	**0.914543**	0.914543

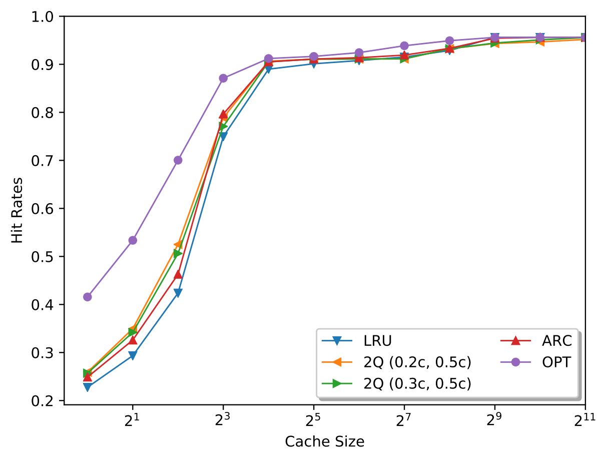
namic replacement method combining concepts from LRU and LFU. It maintains two caches, denoted t_1 and t_2, which stores entries that appear once or more than once. ARC also uses the concept of *ghost caches* that dynamically adjusts the size of t_1 and t_2, which allows it to adjust to different workloads over time (e.g., scan-pattern workloads and workloads with hot items).

Evaluations on cache replacement policies. Al-Zoubi et al. [2] present a performance evaluation of CPU cache replacement policies: FIFO, Random, OPT, LRU, and PLRU for the SPEC CPU2000 Benchmark Suite [7]. However, our project differs from theirs in that we evaluate the replacement policies for file system cache. The paper finds that the optimal policy performance is near the performance of the next best policy of a cache twice as big. Our project will also evaluate the optimal policy, which provides an upper bound for the performance of replacement policies.

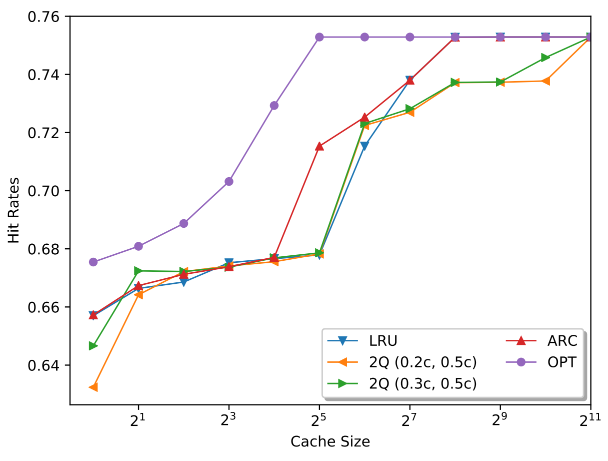
Trace-driven analysis Ousterhout et al. [14] presented a trace-driven analysis of a UNIX BSD file system. A simulator that uses the traces to analyze the performance of disk cache is also provided. Their cache simulator uses the LRU replacement policy [12]. The simulator finds out which blocks are accessed based on `read`/`writes` and checks if the accessed blocks are in the cache from the collected traces. Their simulation also allows configuration of cache size, write policy, and block size. The aforementioned paper shows that trace-driven analysis is a valid, accurate approach to analyze cache behavior when given traces that are accurate enough, and eBPF makes it possible to collect such traces with tools like *bpftrace*. Our project can follow a similar approach to using traces as simulator input to figure out the performance of replacement policies.

7 Conclusion

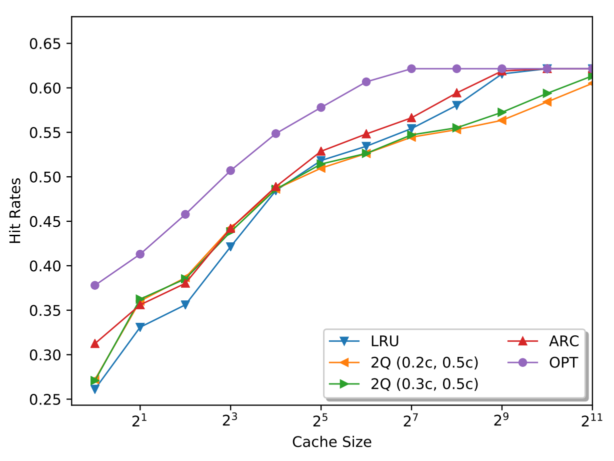
We implemented and analyzed the performance of different cache replacement policies, LRU, 2Q, ARC under various



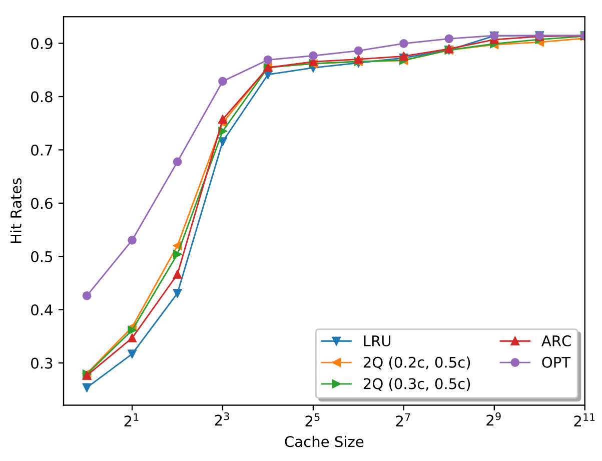
(a) Hit Rates of **Kernel** Trace



(b) Hit Rates of **Video** Trace



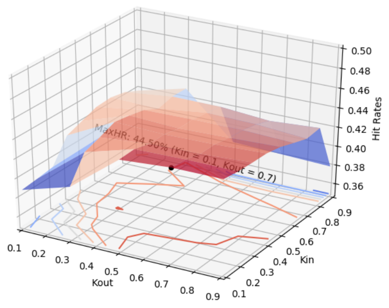
(c) Hit Rates of **Daily** Trace



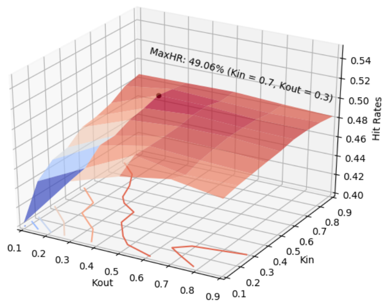
(d) Hit Rates of **Mix** Trace

Figure 1: Hit Rates of Different Workloads

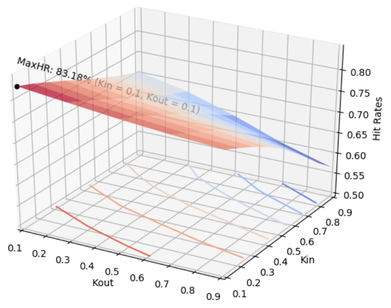
workloads collected by eBPF-based tracing tools, namely bpftrace. We have three main findings: First, the overall performance of ARC is better than 2Q and LRU in our experiments. Second, the effectiveness of 2Q is sensitive to K_{in} and K_{out}, which are hard to tune except for brute-force enumeration. Third, ARC can adapt to different workloads, as it finds the appropriate adaptation parameters value under various access patterns.



(a) **Daily** Trace with Cache Size 8M

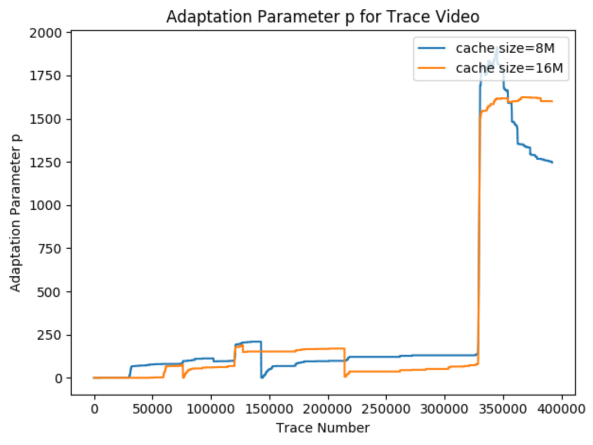


(b) **Daily** Trace with Cache Size 16M

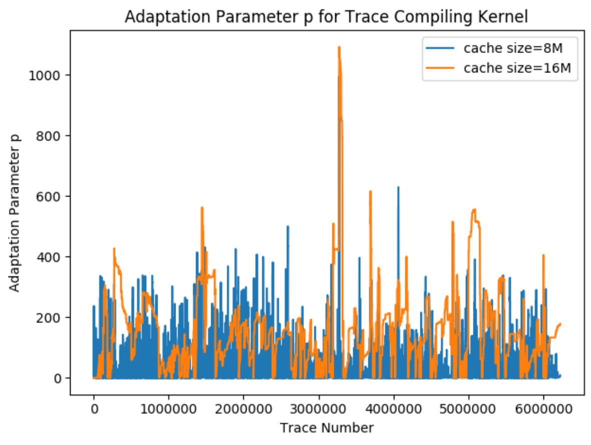


(c) **Kernel** Trace with Cache Size 8M

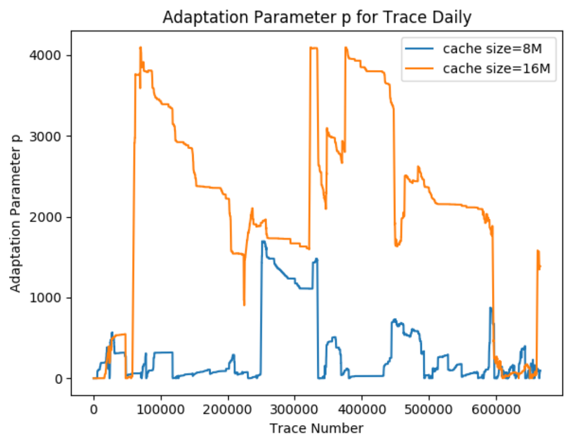
Figure 2: Hit rates for 2Q when tuning K_{in} and K_{out} under different workloads and different cache sizes.



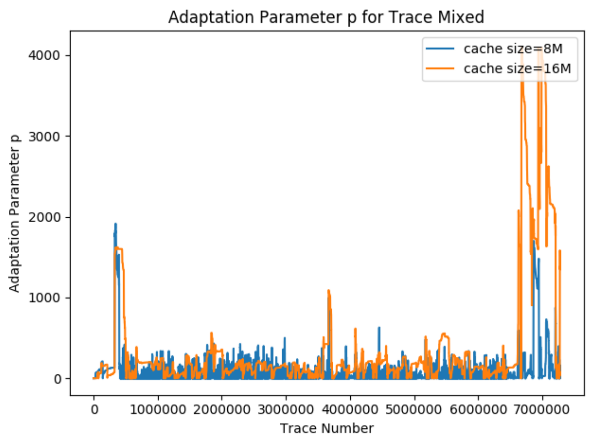
(a) **Video** Trace: Adaptation Parameter p



(b) **Kernel** Trace: Adaptation Parameter p



(c) **Daily** Trace: Adaptation Parameter p



(d) **Mixed** Trace: Adaptation Parameter p

Figure 3: Adaptation parameter p changes in each workload.

References

[1] Cilium - Linux Native, API-Aware Networking and Security for Containers. https://cilium.io/ (Date retrieved: October 16, 2020).

[2] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference, 2004, Huntsville, Alabama, USA, April 2-3, 2004*, pages 267–272, 2004.

[3] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[4] Laszlo A Belady, Robert A Nelson, and Gerald S Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[5] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, (1):64–84, 1980.

[6] eBPF.io. What is eBPF? An Introduction and Deep Dive into the eBPF Technology. https://ebpf.io/what-is-ebpf.

[7] John L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[8] Theodore Johnson and Dennis E. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450, 1994.

[9] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[10] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003.

[11] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *Fast*, volume 3, pages 115–130, 2003.

[12] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[13] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/.

[14] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24, 1985.