MLCache: A Multi-Armed Bandit Policy for an Operating System Page Cache

Renato Costa
University of British Columbia
renatomc@cs.ubc.ca

Jose Carlos Pazos
University of British Columbia
jpazos@cs.ubc.ca

ABSTRACT

Despite significant speed improvements in persistent storage media over the last decades, access to disk remains orders of magnitude slower than memory access. Operating systems typically employ page caches to provide applications with the abstraction of a fast persistence layer. However, memory is limited and the kernel needs to make decisions as to which blocks of data to evict and which to keep in memory. From an end-to-end perspective, application-controlled policies are thought to be a more efficient technique than kernel-based, global policies. However, it introduces a burden for developers, who need to profile each application individually and make low level decisions that might impact other applications. With the recent rise of machine learning, it seems natural to offload the work to learning algorithms, so that application developers are freed from the responsibility of dealing with such issues while getting the performance benefits of having specialized policies for their systems. In this paper, we propose MLCache, a page cache policy that learns access patterns for different applications in order to improve cache hit ratios for subsequent runs. Our results show that MLCache adds low overhead on most use cases and, even with limited training, yields better runtime performance for read-intensive workflows.

1 INTRODUCTION

The page cache (sometimes called buffer cache) is a standard component in an operating system: its main purpose is to provide user space applications with the abstraction of a fast persistence layer. That is achieved by keeping blocks of data read from the file system in main memory. When applications subsequently request to read the same block of data, the kernel can immediately provide that data to the application without issuing a request to the considerably slower disk device. When memory allocated for the page cache is exceeded, the kernel needs to decide which pages to free; in other words, the operating system implements an *eviction policy*. This policy allows the kernel to determine the relative importance of every page currently in the cache. Typically, eviction policies — both in operating systems and in other systems that make use of cached content — employ LRU (Least Recently Used), due to its simple implementation and good empirical results [5].

Application-controlled strategies have been proposed in the past as an alternative to global caching policies that might not be appropriate for different types of applications, each with different caching needs. In particular, monolithic operating systems such as Linux, FreeBSD and Windows lack the flexibility to meet specific application requirements. Application-centered approaches often show beneficial results, especially when incorporated with different techniques such as disk scheduling, and prefetching [4]. This flexibility is also aligned with the end-to-end argument [7], allowing developers to choose different policies based on their specific needs. However, it also introduces two undesired consequences: firstly, applications need to be changed in order to benefit from the improvements; secondly, it requires application developers to think in terms of low level policies that should be abstracted away by the kernel. This problem becomes especially hard when taking into account the interplay between several concurrent processes, which are constantly competing for resources. Additionally, these policies should be adaptable to different workloads, possibly changing on the fly. Thus, this design is not practical. In this work, we present MLCache as an attempt to bridge the gap between the potential performance gains of an application specific policy and the ease of use of a global, general purpose kernel policy. A potential solution to this complex problem is to employ machine learning strategies, through the use of unsupervised techniques or supervised techniques that require minimal input from a developer. We believe that this approach provides similar enhancements to those previously seen from application-controlled caching policies, while being substantially easier to maintain, and ideally hiding the complexities from the developer by pushing the work to the kernel.

2 LEARNING MODEL

We apply a simple linear model to keep track of our scores. This provides us with two benefits: the implementation is simple, and the overhead introduced by the constant processing of the scores is minimized while maintaining relevance. We employ a multi-armed bandit approach for our model, using the UCB1 algorithm [3]. The possible actions to choose from are the current `k` pages in the cache, and we maximize according to the following formula at time step $t > 0$:

$$\max_{1 \le j \le k} \bar{x}_j + \sqrt{\frac{2\log(t)}{n_j}}$$

where $\bar{x}_j$ is the average reward obtained from action j, n_j is the number of times action `j` has been played.

Due to constraints for numerical calculations in kernel space and the fact that we want to evict the worst page in the cache, we have modified the algorithm to better fit our scenario. Thus, our implementation evicts according to the following formula:

$$\max_{1 \le j \le k} -\bar{x}_j - isqrt(\frac{2S^2 ilog(S^2 t)}{n_j})$$

where `isqrt` and `ilog` are the integer versions of the `sqrt` and `log` functions. For example, $isqrt(x)$ is the biggest integer `k` such that $k^2 \le x$, and similarly with $ilog(x)$. This is due to floating point operations not being readily available in the Linux kernel. S is a scaling factor chosen so as to not lose too much precision in our operations. In fact, we scale every single operation by this factor: time steps are multiplied by the scaling factor, as well as rewards and the number of times an action has been chosen. In our implementation we chose $S = 100$, as this gives us about 4 decimal points of precision in the `sqrt` operation. Unfortunately, the nature of `log` makes even this factor relatively low and thus precision is lost in this operation. The final modification we make is to take the negative of the previous sum, as we represent better pages by lower scores, evicting the maximum scores.

3 IMPLEMENTATION

Our MLCache prototype was built on top of the Linux kernel version 4.14.0-rc7. In this section, we provide a brief background to the page cache implementation on Linux and describe how we modified it in order to add the learning model described in Section 2.

3.1 The Linux Page Cache

Linux employs a modified version of the traditional LRU strategy called *two-list LRU*. As the name suggests, two LRU lists are maintained in this approach: the *active* and *inactive* list. The active list contains pages that have been accessed more than once in the past, whereas the inactive list holds the set of pages that have been accessed only once. In other words, the active list attempts to capture *frequency* and the inactive list captures *recency*.

The following subsections introduce the main concepts that are relevant to our work on MLCache. Figure 1 describes the two-list approach used in Linux.

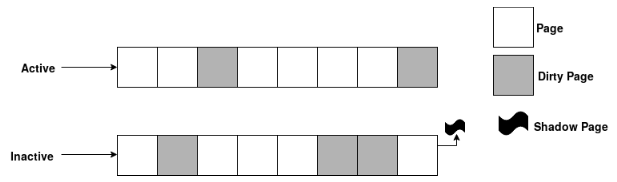


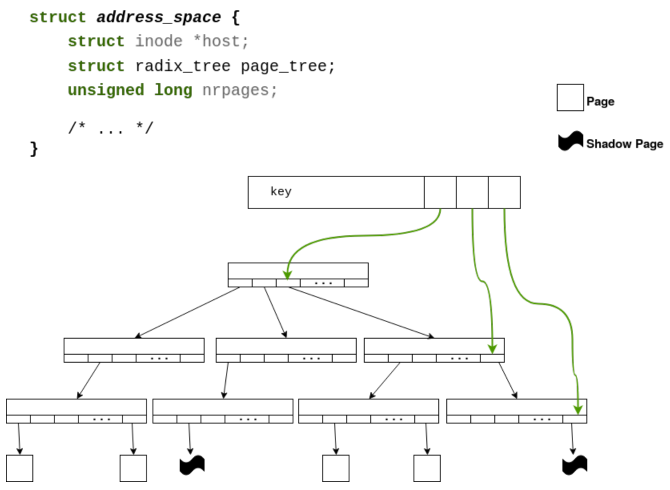
Figure 1: The two-list LRU page cache on Linux. When a page is evicted (last page of the inactive list), it is replaced by a shadow page.

3.1.1 Dirty Pages. As mentioned previously, the page cache holds blocks of data stored in the disk device. When user space applications perform write operations to certain blocks of data, the corresponding pages are marked *dirty*. Dirty pages contain data that has not yet been flushed to the persistence device. In Linux, dirty pages are not immediately flushed to the disk when the write operation occurs; instead, the kernel periodically schedules a *writeback thread* that scans the LRU lists and writes dirty pages to the disk.

3.1.2 Cache Eviction. When pages are read from or written to the disk, new entries are added to the page cache. As long as there is enough memory on the system, the number of pages will keep growing, up to a certain threshold[1]. When the system is under memory pressure, the kernel will create separate threads that will *shrink* the LRU lists. In particular, pages at the tail of the inactive LRU list are removed (pages are always added to the head of an LRU list.) For this reason, Linux also periodically ensures that the active and inactive lists are balanced: for example, if too many pages are in the active list, the ones at the tail will be demoted to the inactive list.

[1]Even a special webpage was [6] created in order to clarify this behavior, which probably scares unfamiliar users.

3.1.3 Page Lookup. Every page I/O operation on Linux goes through the page cache — if the an entry is found (*cache hit*), an expensive call to the disk driver is saved. Since these page lookups happen very often, they need to take a very short time to produce a result. Linux 2.6 introduced a new page cache lookup algorithm using *radix trees* (as opposed to the old, global hash table in previous kernels.) Each file contains its own radix tree of pages, and lookups happen by providing only a file offset. Figure 2 illustrates the radix tree approach, as defined in Linux 4.14.0-rc7.



```
struct address_space {
    struct inode *host;
    struct radix_tree page_tree;
    unsigned long nrpages;

    /* ... */
}
```

Figure 2: The Linux page tree. Each file is represented by a (poorly named) `struct address_space`, and contains a radix tree with pointers to locations where pages are kept in memory. When a page is evicted, a shadow page is placed in its previous spot. In the example lookup of this figure, key resolves to an entry with a shadow page. The corresponding page will likely be loaded directly into the *active* LRU list.

Another important aspect of the Linux implementation of radix trees is that nodes can be marked with *tags* — the page cache uses this functionality to indicate whether pages are dirty, unevictable, or a series of other attributes. Nodes may also be marked *exceptional*, a feature that is used to detect shadow pages in the radix tree.

3.1.4 Shadow Pages. Another important optimization employed by the Linux page cache implementation is the use of *shadow pages*. Whenever a page is evicted, a special entry — called shadow page — is created and replaces the spot previously occupied by the recently evicted page in the radix tree. When a page lookup occurs and a shadow page is found at the requested offset, the new page is added to the page cache and inserted directly into the active LRU list. The use of shadow pages is a cheap optimization (the shadow page is 8 bytes long, whereas a page is 4 KB in x86) that can potentially avoid workflows that would cause thrashing.

3.2 Page Scoring

MLCache automatically calculates the score for each page based on the algorithm described in Section 2. Specifically, for every page definition (`struct page`), MLCache adds two new fields: the current page score, and the number of times the page was previously evicted. In order for the algorithm to work, the page scores need to remain accurate at all times.

Fortunately, Linux supports a *tracepoint* API. More precisely, events can be defined and, once included in a source file, be triggered with certain arguments. MLCache introduces a new tracepoint, `mlcache_event`, which is supposed to be used whenever a page is requested. The event accepts a series of parameters including a pointer to the `struct address_space` involved, a pointer to the page itself, the offset requested, and whether it was a hit or a miss. The newly added event is triggered from within the `find_get_page` function in Linux.

3.3 The `mlcache` Driver

MLCache also introduces a new Linux driver, which coordinates its execution. The driver's main tasks are:

Update page scores When this driver is loaded, it registers a function to be executed whenever the `mlcache_event` tracepoint is triggered. This function rewards the page being requested and penalizes the other pages in the cache by traversing the associated radix tree.

Allow configuration by system operators While MLCache can potentially improve the performance of a large class of applications, it is convenient to allow the system operator to selectively choose which applications should use MLCache's algorithms. That is done by writing to the `/proc/mlcache/filter` file that MLCache creates when it is loaded. Currently, MLCache is able to monitor either a list of processes or the process tree under a certain ID. For instance, the command:

```
echo tree:$$ >/proc/mlcache/filter
```

causes MLCache to monitor every process under the running shell.

Read collected information When monitoring a process, MLCache will create a file under the `/proc/mlcache` directory named after the identifier of the monitored process. By reading that file, system operators can analyze statistics about page cache hits/misses.

3.4 Leveraging Shadow Pages

As described in 3.1.4, Linux uses shadow pages to optimize access to frequently accessed pages even if they happen to be evicted due to high memory pressure. In our work, we extend the information contained in the shadow page to include MLCache's scoring data. More specifically, whenever a page is evicted from the page cache, we include both the current page score, as well as the number of times the page was previously evicted in the created shadow page. Thus, when a page request finds a shadow page, it can be brought back to the cache with the score it used to have before evicition. Figure 3 illustrates how the shadow page is structured when MLCache is enabled.

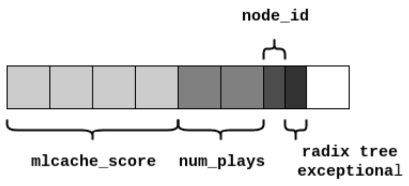


Figure 3: The Linux shadow page when MLCache is enabled. Each square represents one byte of information. The shadow page is 8 bytes long and contains a series of data — CPU node identifier, a radix tree entry indicating this is an exceptional node, among others. MLCache adds a 4 bytes long page score, as well as a 2 bytes long counter, indicating the number of times that page was previously evicted.

3.5 Cache Eviction

The final component of MLCache's implementation is the modified cache eviction policy. Cache eviction on Linux happens when the amount of memory consumed by the page cache is above a certain threshold. When pages need to be evicted, kernel threads will be awaken and both the active and inactive LRU lists may be *shrinked*. The shrinking process happens by evicting the last `n` pages in an LRU list (where the value of `n` is determined at runtime depending on the system's memory pressure.)

MLCache changes the LRU shrinking process by choosing the pages with the worst score. Unfortunately, LRU lists are maintained as a linked list of pages, and choosing the pages to evict in LRU is a constant time operation. Since MLCache needs to determine which pages have the worst score overall, a linear scan over the LRU list is introduced in this step. Even though better data structures could be leveraged in order to optimize the eviction process, we see positive results for some workloads in our benchmarks, despite this unfavorable situation.

4 EVALUATION

We ran our tests on the QEMU [1] hypervisor on top of Linux 4.11.8, with hardware virtualization enabled (using KVM.) In the benchmarks described below, all virtual machines were launched with two processors, and a varying amount of RAM. Since MLCache's main goal is to make smarter cache eviction decisions, most of our interest lies in benchmarks where the size of the page cache can potentially outgrow the amount of memory available in the system.

We developed two different benchmarks to test MLCache:

- **`cscope`.** This benchmark consists of running `make cscope` inside the Linux kernel source tree. `cscope` is a source code indexing tool which allows developers to quickly find C symbol definitions, function call sites, pattern matching, among others. The purpose of this benchmark is to evaluate how well MLCache performs in a workload that is primarily sequential (scanning of source files.)

- **pgbench.** This benchmark consists of running the pgbench utility on a database that is a lot larger than available memory. pgbench is a bechmarking tool for the PostgreSQL database. It is able to perform a variety of tests, some of which are based on the TPC-B database benchmark [2]. The goal of this test is to understand how MLCache can potentially help a common scenario in commercial applications (a dedicated database server.)

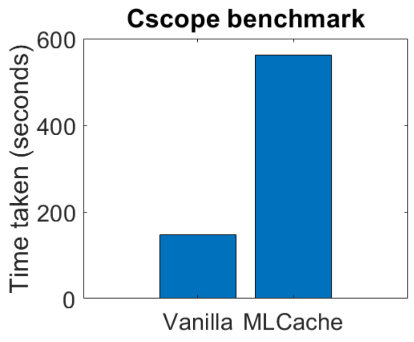


Figure 4: Cscope benchmark results for the unmodified Linux kernel versus our MLCache prototype, ran on a VM with 256MB of RAM.

The results from the **cscope** benchmark give us some sense of the overhead caused by our implementation, as the current Linux implementation does not benefit very much from caching for this benchmark. We can see that the time taken was about 3x larger on MLCache. This is not surprising, as we have to update each page's scores in the cache on every reference, as well as doing a linear scan over Linux's linked list on eviction. Additionally, Linux is already very optimized for the linear scan scenario, reading blocks ahead of time when a page is requested. Thus, MLCache only adds overhead for score maintenance in this case and performance is greatly affected.

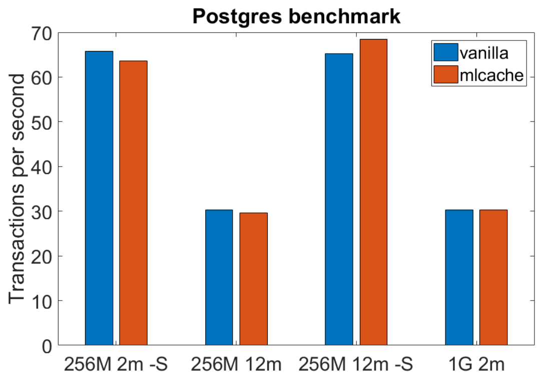


Figure 5: Postgres benchmark results for the unmodified Linux kernel versus our MLCache prototype, with different RAM sizes. The -S command means just reads, no writes.

Our results from the **pgbench** benchmark show us that even with the extra machinery we have built on top of Linux, we achieve similar performance, especially as memory decreases and on longer runs. This is probably due to the current implementation having to develop new scores from scratch on every run, and so reaching accurate scores takes some time. After 12 minutes, however, our design starts to outperform Linux. This is relatively fast, since training would ideally be done only once, and the rest is online learning. We should note that our implementation takes advantage of smaller memory sizes due to the increased amount of evictions done: without any evictions, our design is entirely overhead.

As a small experiment, we also tested how our implementation fared when evicting the lowest score (best) pages every time. For the **cscope** benchmark, it took about 3 hours to complete. This gave us some reassurance that our algorithm, although not ideal, is in the right direction compared to a completely naive implementation.

5 LIMITATIONS AND FUTURE WORK

Our current prototype has several limitations:

Lack of accuracy in the calculations There is no easy way to perform floating point operations on kernel space. However, we expect this not to be a prohibitive issue, as the `isqrt` operation is sufficiently precise given our scaling factor.

Improper data structures for our purposes Although we explicitly chose a lightweight model to avoid bringing unnecessary overhead to the kernel, our prototype is built on top of the Linux LRU implementation. Since that implementation is extremely optimized for that policy, MLCache introduces significant overhead to the current system during eviction: for each page to evict, we scan through a linked list to find the worst scoring page. Ideally, we would have a different data structure to keep track of scores, such as a priority queue. This would greatly decrease the runtime of the system overall.

No training phase Ideally, we would have a persistent score per page throughout several executions of a program. This would provide greater accuracy. Unfortunately, due to the early state of the project, we have not been able to achieve this. Due to this limitation, every time a new page is introduced to the cache, we assign an average score across all pages, which may make the page look better or worse than it actually is to the algorithm. Memory limitations also add to this issue, as the pages referenced by a file are in a constant state of flux, and we experienced heavy memory usage when keeping track of all scores.

Memory overhead When enabled, MLCache adds two new `unsigned long` fields to the definition of a page on Linux (`struct page`.) This incurs an overhead of approximately 15% on a default x86 kernel configuration. We could potentially reduce that overhead by keeping scores only for pages that belong to processes that are currently monitored by MLCache.

We would like to evaluate our implementation against the optimal eviction strategy, as this would give us a better idea of how close we are from getting to it — after all, our main goal is to allow applications to achieve better caching without having to be altered manually.

Possible enhancements to this approach would be to integrate other techniques, like disk scheduling, and prefetching. This is rather complex in practice, given the possible interplay between many different types of applications. It is not clear whether a machine learning approach would provide a performance enhancement for such a design, as the modelling complexity might incur too much overhead in the system.

6 CONCLUSION

Our prototype MLCache has proven to be on par with the Linux implementation for read-intensive workflows such as the the **pgbench** benchmark. This shows great promise, since a number of improvements are possible and yet, we had comparable results when compared to the standard Linux LRU implementation. For other results, like the **cscope** benchmark, we are still in the process of improving the implementation so, even for these cases, the performance gap will diminish as our design matures. Overall, with a very simple design, we have shown that an adaptive approach to caching can be beneficial to potentially different kinds of applications. This is only the first step; further work is needed in order to use optimized data structures for our eviction policy, which could bring even greater performance gains. Most importantly, we believe that machine learning techniques can be leveraged in order to provide increased performance to existing, unchanged applications.

REFERENCES

[1] 2017. QEMU. (2017). https://www.qemu.org/

[2] 2017. TPC-B Homepage. (2017). http://www.tpc.org/tpcb/default.asp

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* 47, 2-3 (May 2002), 235–256. https://doi.org/10.1023/A:1013689704352

[4] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. 1996. Implementation and Performance of Integrated Application-controlled File Caching, Prefetching, and Disk Scheduling. *ACM Trans. Comput. Syst.* 14, 4 (Nov. 1996), 311–343. https://doi.org/10.1145/235543.235544

[5] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. https://doi.org/10.1145/363095.363141

[6] Vidar Holen. 2009. Help! Linux ate my RAM. (2009). https://www.linuxatemyram.com/

[7] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402