Log-Structured Cache: Trading Hit-Rate for Storage Performance (and winning) in Mobile Devices

Abutalib Aghayev
College of Computer and Information Science
Northeastern University
Boston, MA
aghayev@ccs.neu.edu

Peter Desnoyers
College of Computer and Information Science
Northeastern University
Boston, MA
pjd@ccs.neu.edu

ABSTRACT

Browser caches are typically designed to maximize hit rates—if all other factors are held equal, then the highest hit rate will result in the highest performance. However, if the performance of the underlying cache storage (i.e. the file system) varies with differing workloads, then these other factors may in fact not be equal when comparing different cache strategies.

Mobile systems such as smart phones are typically equipped with low-speed flash storage, and suffer severe degradation in file system performance under sufficiently random write workloads. A cache implementation which performs random writes will thus spend more time reading and writing its cache, possibly resulting in lower overall system performance than a lower-hit-rate implementation which achieves higher storage performance.

We present a log-structured browser cache, generating almost purely sequential writes, and in which cleaning is efficiently performed by cache eviction. An implementation of this cache for the Chromium browser on Android was developed; using captured user browsing traces we test the log-structured cache and compare its performance to the existing Chromium implementation.

We achieve a ten-fold performance improvement in basic cache operations (as measured on a Nexus 7 tablet), while in the worst case increasing miss rate by less than 3% (from 65% to 68%). For network bandwidths of 1 Mb/s or higher the increased cache performance more than makes up for the decrease in hit rate; the effect is more pronounced when examining 95^{th} percentile delays.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*

General Terms

Performance, Design, Experimentation

Keywords

Log-structured, cache, mobile browser, NAND flash, Chromium, Android

1. INTRODUCTION

The use of mobile computing devices like smartphones and tablets is growing at a remarkable speed, and analysts are predicting mobile users surpassing desktop users within a year [6]. At the same time, performance of these mobile devices lags far behind that of desktops. While some of this disparity in performance is due to slower networks and CPU, recent research [8] has shown that the flash storage may be the primary bottleneck in many mobile applications.

Although this research primarily measured the performance of removable MicroSD cards on mobile devices, it is also applicable to on-board flash storage in most smart phones. At present, most of these devices rely on eMMC (Embedded Multi-Media Card) devices for flash storage; these are essentially MicroSD cards in a soldered package; like a MicroSD card, they typically rely on an 8-bit micro-controller for internal processing. While devices of this sort have performed well for simple applications like music and photography, their performance degrades, sometimes catastrophically, with the random access patterns generated by complex applications.

One such complex application is a web browser. Browsers are by far the most used application on desktops, and are becoming so on tablets and smartphones. The primary use of storage within a browser is in turn its persistent cache, which stores copies of most resources retrieved from the network for possible reuse. An average web page contains about 50 resources [12], therefore, even moderate browsing activity results in large amounts of storage I/O.

Caches of modern browsers like Chromium [1] and Firefox [2] employ some variant of LFU or LRU to maximize hit rate. When paired with a storage system with very poor random write performance, however, this hit rate may come at the expense of slower cache operations. Performing in-place LFU, as is done in Chromium for instance, results in many random writes with correspondingly poor file system performance on mobile devices which we have tested. The eMMC storage on these devices lacks command queuing, and therefore slow writes block reads; thus, although writes may be performed asynchronously in the background, they still can adversely affect user performance.

We present an implementation and analysis of a cache designed to maximize write throughput to flash storage, at the cost of a small decrease in hit rate. The cache is structured much like a log-structured file system [13], dividing storage into large *segments* which are written sequentially. Once filled, a segment is immutable and cannot be reused until it is *cleaned* of any remaining valid data. Although segment cleaning in a log-structured file system is complex and often time-consuming, in a cache it may be performed simply by evicting any remaining data in a segment selected for cleaning. Also, unlike log-structured file system, we do not buffer segments in memory until they are full, but rather write data as it becomes available in order to reduce the memory footprint. Finally, due to sequential nature of write traffic, write amplification is minimized, resulting in optimal wear leveling.

The contributions described in this paper are as follows:

- We present a design and rationale for a log-structured cache optimized for write performance on low-end flash media.
- We present experimental results showing a nearly tenfold improvement in read and write latency for cache operations.
- Using real browsing traces collected over a period of two months, we compare our cache with simple FIFO eviction to the existing cache. The worst case reduction in hit rate is no more than 3% (35% to 32%) across the traces and cache sizes we evaluated.

2. OVERVIEW

Although today's smart phones run general purpose operating systems on CPUs as powerful as desktop machines of several years ago, storage systems on these devices are far less powerful than those found on typical desktops or laptops. Almost all Android devices on the market appear to use eMMC [3] storage, essentially a soldered down version of ubiquitous MicroSD card. Where desktop and laptop SSDs use 32-bit microcontrollers with megabytes of memory to run the algorithms which translate operating system requests to flash operations, these low-end devices may rely on a 8-bit microcontroller with less than 10 KiB of RAM.

As an example, in Figure 1 we see sequential read and write throughput for direct I/O (i.e. un-cached) requests to a 16 MiB file on a Galaxy Nexus phone running the Android operating system. Full write bandwidth is not achieved for the write sizes less than 1 MiB and performance falls off drastically for write sizes less than 64 KiB.

This performance for sequential I/O is comparable to what was observed in the recent work of Kim et al. [8] which measured the performance of external storage (MicroSD cards) as well as on-board flash (eMMC) on a Nexus One phone. Not surprisingly, we found the random I/O performance to be at an unusable level (0.02 MiB/s). This is a known artifact of the flash translation algorithms used on controllers with limited memory [9], which with modern flash chips may suffer 128 times or worse decrease in throughput for random writes [5].

As can be seen in the Figure 1, to achieve the best write performance with these devices, large contiguous writes must be used. This is similar to one of the constraints that shaped the design of the Log-Structured File System. Rosenblum and Ousterhout observed that sequential I/O is much faster than random I/O on modern disks due to seek overhead and argued that read performance would decrease in importance

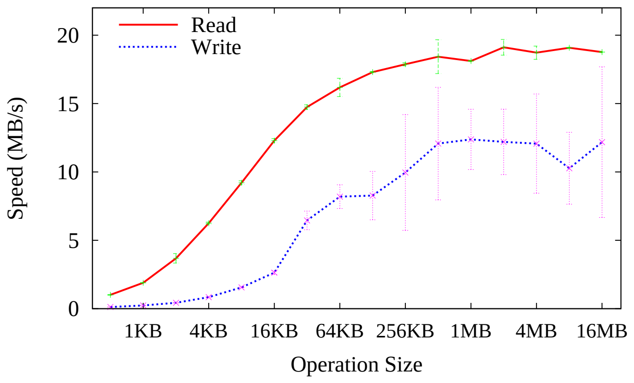

Figure 1: Sequential I/O throughput of the eMMC found on Galaxy Nexus.

with growing memory sizes as more read requests would be served from memory. In the case of flash, we note that the read performance is inherently faster than write and does not degrade with random I/O, and large sequential writes are needed to achieve high performance.

Log-structured file system divides the storage into large fixed sized segments, only one of which (the *write frontier*) is written to at a time. Writes are buffered in memory until a full segment can be written to disk. Updates to existing data are written to the write frontier and cause the overwritten data (living in some previous segment) to become obsolete. When the number of free segments drops below a certain threshold, a cleaning process chooses segments to be compacted. Any remaining live data from these segments is copied to the write frontier making these segments available for reuse.

This process of compacting live data in a log-structured file system is known as *segment cleaning*. Finding an efficient segment cleaning algorithm has been the most challenging part of the log-structured file systems. Much research has been devoted to it with the best implementations causing serious performance degradation under certain scenarios [14].

The design of our cache is similar to the design of the log-structured file system. A large preallocated file is used instead of a block device and is divided into fixed sized segments. Cached objects are written to the segment which happens to be the write frontier and once the segment is full, a new segment is chosen as the write frontier. We manage to avoid the problem of segment cleaning by choosing to evict entries that live in the segment that we chose to clean. In addition, due to memory constraints on mobile devices, the segments are not buffered in memory; data is written to segment as it becomes available.

3. LOG-STRUCTURED CACHE DESIGN

Our log-structured cache was implemented and evaluated in Chromium, the open-source browser from Google. This decision placed certain constraints on our design, due to the cache API and overall architecture of Chromium.

3.1 Chromium's cache architecture

Chromium's cache is implemented in two layers. At the lower layer there is a generic object cache *backend*, that can store *entries* consisting of a fixed number of *streams*, each of arbitrary size. C++ definitions for the key portions of the API for this layer are shown In Figure 2; this interface is used for storage and retrieval by higher-layer caching components in Chromium.

The primary user of this interface is the *HTTP cache*, used to store most webpage objects. The HTTP cache uses two streams for each object cached; one stream holds the HTTP metadata such as content-type and last-modified time, while the other holds the object itself. This two-layer design allows the same lower-layer cache backend and replacement policy implementation to be used for multiple higher-layer caches, each with different semantics. Examples of higher-layer caches are the *media cache* and the *HTML5 AppCache*, both with different caching semantics; we do not consider them in this work.

In order to achieve high performance and fast user response, almost every internal API in Chromium is asynchronous, through the use of callback functions as shown in Figure 2. Function calls return immediately and the return value determines whether the operation has completed; if not, a task is posted to a dedicated *cache thread* and a callback invoked at some time in the future to indicate a completed task. The log-structured cache described below implements this API, replacing the current Chromium cache.

```
// Backend class methods
int CreateEntry(const std::string& key,
                Entry** entry,
                const CompletionCallback& cb) = 0;
int OpenEntry(const std::string& key,
              Entry** entry,
              const CompletionCallback& cb) = 0;

// Entry class methods
int ReadData(int stream_index,
             int stream_offset,
             IOBuffer* buf,
             int buf_len,
             const CompletionCallback& cb) = 0;
int WriteData(int stream_index,
              int stream_offset,
              IOBuffer* buf,
              int buf_len,
              const CompletionCallback& cb) = 0;
void Doom() = 0;  // Mark entry for deletion.
void Close() = 0;  // Release a reference to entry.
```

Figure 2: Chromium Cache Backend API.

3.2 The new backend

The log-structured cache implementation stores data in a single large file, which as shown in Figure 3 is divided into segments of large fixed size. The segment size is chosen to be much larger than the average cache entry, and to be larger than the cleaning unit of the flash device (typically 4 MiB for eMMC devices we have seen). Since the cache operates above the file system it is not possible to align segments with underlying storage boundaries; however by creating the cache file at once it is typically possible to obtain sequential block allocation.

Each segment holds multiple entries, followed by a footer or *segment summary* holding offsets to each entry in the

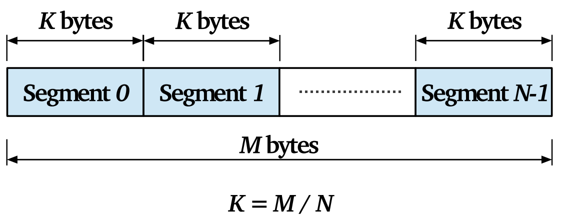

Figure 3: Log-structured cache view of the storage.

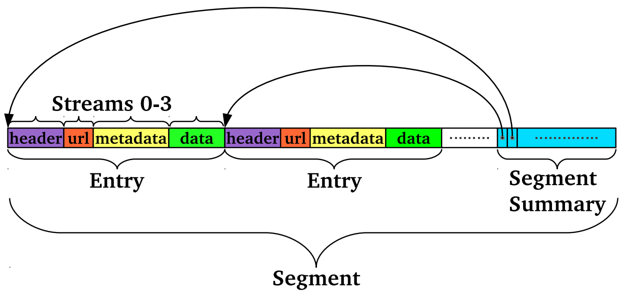

Figure 4: Contents of a Segment.

segment, for more efficient scanning at cleaning time. An entry consists of a header, followed by a key (e.g. URL), followed by one or more streams. Figure 4 shows the entry layout when the log-structured cache backend is used by an HTTP cache. Internally, we do not distinguish the key from the other streams and consider it to be just another stream. The header is of fixed size and holds the size of each stream.

The segment summary has a fixed number of slots and stores the offsets of every entry that was written to the segment; entries do not span multiple segments. The segment size thus determines the maximum size of an entry that can be inserted into the cache, and the number of entries that can be stored in a segment is determined by the number of slots in the segment summary. Entries larger than a segment are stored as separate files on disk, just like the current cache does. The segment summary is sized assuming a mean entry size of 4 KiB, based on trace data as shown in Figure 6.

3.2.1 The Index

In addition to storing entries, we also store the cache index, which maps the hash of the key (for the HTTP cache, the URL) to the corresponding entry's location in the file, as shown in Figure 5. The index is very small in comparison to the cache storage, and is updated on every new entry insertion. It is thus kept in memory as a memory-mapped file, which would only be written back to storage under memory pressure or on browser exit. (If the index is corrupted by a power failure, the entire cache can be emptied). In our prototype implementation the in-memory hash structure is not mapped to a file.

3.2.2 Insert

Inserting an entry into the cache backend involves a few

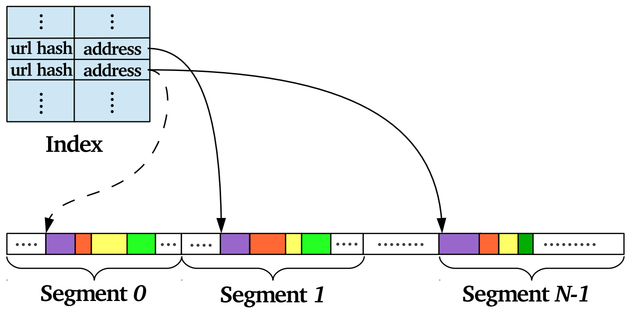

Figure 5: Cache index and effect of updating an entry.

steps. Once the key of the entry to be cached is known, the high-level cache calls *CreateEntry* member function of the *Backend* object, which upon success returns an *Entry* handle. As the data is received from the network, the streams of the entry are populated via *WriteData* call. The entry is buffered in memory until *Close* is called. After *Close* is called, the entry is written to the current segment and its location in file is written to index as well as the segment summary. Segment summary of the current segment is buffered in memory and is not written until the segment is full, resulting in sequential fill of the segment.

3.2.3 Open

A cache lookup from the upper layer results in a call to *OpenEntry*, which attempts to locate an existing entry with a specified key and open it for reading. Given the key, it calculates the hash and uses the index to find the location of the entry within the file. Once found, the header is read and the offsets of the streams can be calculated and streams can be read.

3.2.4 Delete

If the upper layer determines after creating an entry that it should not actually be cached, it may mark it for deletion by calling *Doom*. When *Close* is called, the in-memory entry is destroyed rather than being written to storage. To delete a previously inserted entry, it must be opened (*OpenEntry*), and then may be marked for deletion (*Doom*), and closed (*Close*). In this case the index is updated, and the space occupied by the entry will become usable when the segment is cleaned.

Note that deleting an entry is performed by the upper layer, and is used when a particular object *must* be removed from cache; in contrast, eviction is performed by the cache backend, and selects objects which *may* be removed from cache.

3.2.5 Update

When an entry is updated (*OpenEntry* $\rightarrow$ *WriteData*, .., *WriteData* $\rightarrow$ *Close*), since we do not perform in-place update, we write a completely new version of the entry to the write frontier.

Figure 5 shows the effect of updating an entry. An entry that was originally written to *segment 0* is re-opened for an update. Once the update is complete, the new version is written to the current write frontier, *segment n-1*. It is possible that only part of the entry (e.g. one stream) was modified; in this case, the remainder of the information is copied from the old entry.

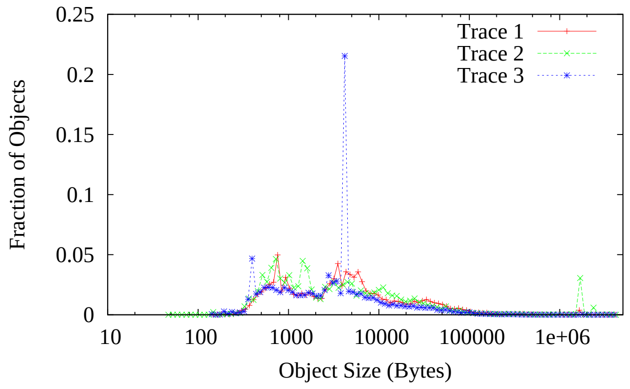

Figure 6: Log graph of object sizes in collected user traces.

An alternative to copying unmodified streams is to store back pointers to the original stream in the updated entry. This strategy was evaluated; the resulting implementation gained very little in space savings at the cost of significant complexity. In addition the loss of spatial locality led to a decrease in read performance.

3.2.6 Segment Life Cycle

A segment starts in the *free* state, goes to the *current* state when it is chosen as the write frontier, and eventually is filled and reaches the *closed* state. A *closed* segment is read-only until it is freed for reuse. Once the number of *free* segments drops below a low watermark, a cleaning process starts and frees segments until their number reach a high watermark.

3.2.7 Cleaning a Segment

Segments are chosen for cleaning in FIFO order, skipping any segments with entries open for reading. Cleaning is done by reading the segment summary, identifying the entries within the segment, and then for each entry retrieving the entry key and checking whether the index for that key hash still points to the entry. If not, the entry has been updated or deleted and no action is required; if the index still points to this entry then it is active, and must be evicted by removing its pointer from the index.

4. EVALUATION

We implemented the log-structured cache for Chromium on Nexus 7, running Android Jelly Bean 4.2.2 based on 3.4.x Linux kernel with ext4 file system. In order to evaluate it, we collected logs from a group of users who used an instrumented version of the stock Chromium browser for two months as their main browser. The instrumentation logged arguments to the cache API in full detail, with URLs hashed with random salts. A short sample from a trace appears in Figure 7.

We note that these traces were collected on desktop and laptop systems (due to Chromium not being completely open source on Android), and thus will differ from mobile browser traces. In particular we expect the mean number of objects per page to be fewer and the sizes of objects to be smaller.

```
4332781373459   WriteData       0x171a0fc0:1:0:1702:1   1702:0
4332781374765   Close   0x171a0fc0      0
4332790701807   OpenEntry       0FD43A89D499A163F2D1173EDE9935A418F4E3A6:77     -2:0
4332790701987   CreateEntry     0FD43A89D499A163F2D1173EDE9935A418F4E3A6:77     0:0x17084950
4332790794450   Doom    0x17084950      0
4332790794491   Close   0x17084950      0
4332797398506   OpenEntry       CC60F344EBFC6D88B2BE1567A4B88E335966D4B5:105    -2:0
4332797398913   OpenEntry       0E7C5EAF3FAD3FA4F4B1FE164DB8303F8762DB1E:217    -2:0
4332797399074   CreateEntry     CC60F344EBFC6D88B2BE1567A4B88E335966D4B5:105    0:0x187a3580
4332797399149   CreateEntry     0E7C5EAF3FAD3FA4F4B1FE164DB8303F8762DB1E:217    0:0x1664b1f0
```

Figure 7: Sample from a trace.

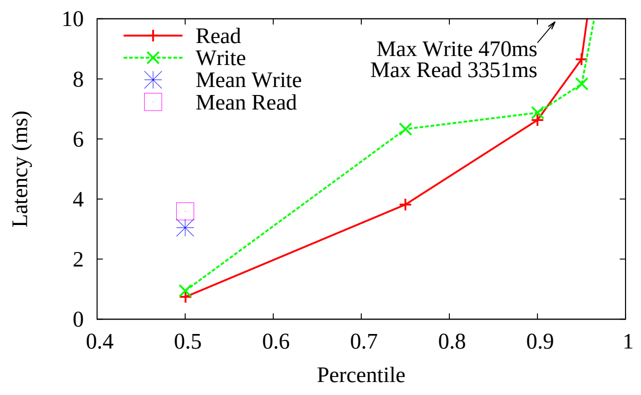

Figure 8: ReadData and WriteData latency percentiles for the existing cache.

4.1 Storage Performance

A test driver was used to replay and measure the latency of the operations. To help ensure accurate reproduction of the application behavior, we preserved inter-operation delays from the trace (truncated to a maximum of 5 seconds), with the exception of inter-session delays. When traces were played back without inter-operation delays, substantial differences in performance were observed, due to additional aggregation of operations at the file system layer.

Figure 8 shows the CDF for *ReadData* and *WriteData* operation latencies for the existing Chromium cache. It can be seen that the 95th percentile *ReadData* and *WriteData* latency is nearly 10ms, with a worst-case read latency of over 3 seconds.

Figure 9 shows the corresponding data for the *log-structured cache*. As can be seen, the 95th percentile *ReadData* and *WriteData* latencies are less than a millisecond, with a worst case of 28 and 10ms.

4.2 Hit Rate

Hit rate was evaluated on the same traces used for testing storage performance. First an upper bound on hit rate was established for each trace, by simulating the operation of an *infinite cache*. That is, we instantiate a cache large enough that would never evict an entry and then replay the trace on it. Since the traces were collected on a cache with bounded capacity, they contain many failing *OpenEntry* calls due to evictions, followed by an immediate *CreateEntry* call to insert the entry. We fix these by preprocessing traces and replacing such *OpenEntry*-*CreateEntry* pairs with succeeding *OpenEntry* calls, thus giving an impression of a trace from an infinite cache. With the *infinite cache*, we never have a failing *OpenEntry* call with the exception of compulsory misses, giving an upper bound on the hit rate. For the collected traces this upper bound ranges from 34% to 45%, which is consistent with previous studies [15].

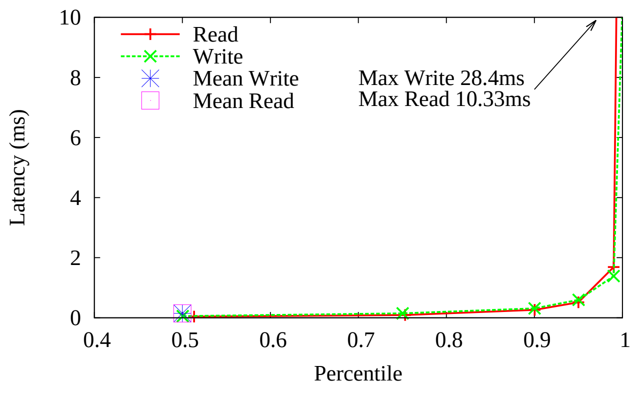

Figure 9: ReadData and WriteData latency percentiles for the log-structured cache.

Due to maximum size limitation of 2 GiB on the current Chromium cache, traces were trimmed to ensure they did not exceed this size before further experiments were performed. In Figure 10 we see the hit rate results for the infinite cache case as well as the cache sizes ranging from 1 GiB to 64 MiB for both the original and the log-structured cache.[1] The impact on the hit rate is seen to be modest; in the worst case there is an increase of 3% in the miss rate, from 65% to 68%.

4.3 Quantifying the Impact

To compare the loss in hit rate to the improvement in storage performance, we calculate the mean per-operation performance

$$T_{op} = T_{read} \cdot R_{hit} + T_{fetch} \cdot (1 - R_{hit}) \qquad (1)$$

where T_{read} is the cache read operation latency and T_{fetch} is the time to retrieve an object—$\frac{8 \cdot S_{obj}}{R}$ where R is the network speed in bits per second and S_{obj} is the mean object size in bytes. Assuming a mean object size of 4000 bytes, at a network speed of 0.94 Mbit/s T_{op} will be 23.16ms for both caches; for higher speeds the log-structured cache will outperform the existing cache.

We can approximate the 95th percentile latency by ignoring object size variance and applying the same formula, giving a lower bound. Here we find that at a network rate of 0.33 Mbit/s both caches will have a 95-percentile per-operation latency of 65.8 ms; for higher network speeds the log-structured cache will be better.

Since 1 Mbit/s is at the low end of reported 3G performance, and smart phone WiFi performance is in the 8-9 Mbit/s range [8], the loss of hit rate for the log-structured cache seems to be more than made up for by increased performance. We note that this analysis omits the overhead of asynchronous writes performed in the background; however although difficult to quantify they have a significant impact as well, by using storage bandwidth and CPU needed by the critical-path portions of the application.

5. RELATED WORK

[1]We note that Chromium's current cache algorithm uses absolute age as one of its criteria for eviction. Tests such as ours which replay weeks worth of activity over a short period will not see the age-based aspects of this algorithm.

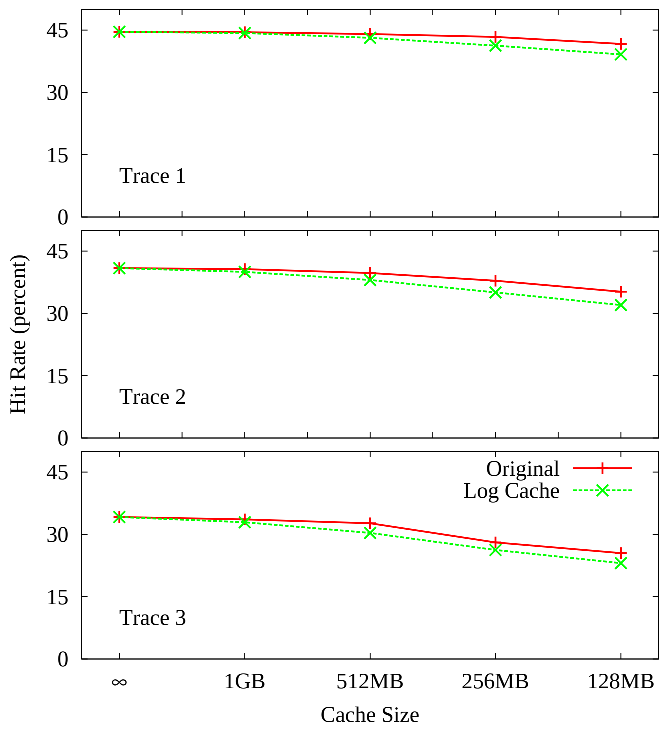

Figure 10: Hit rate for the *log-structured cache* with FIFO eviction policy.

A large body of work has examined the problem of caching web accesses at a proxy providing service for large numbers of client users; an early survey of this area is provided by Wang [16]. Other work has focused more specifically on cache replacement algorithms for web traffic (e.g. Jelenković and Radovanović [7]); a survey by Podlipnig and Böszörmenyi [10] covers many such algorithms which have been proposed. Unlike our work, which focuses on the per-operation cost of the browser cache, web proxy research has in general focused on abstract performance—i.e. byte or object hit rates—and assumed that a sufficiently powerful server would be deployed to handle the workload.

A number of researchers have recently begun to investigate mobile browser cache performance. Qian et al. at Michigan [11] have analyzed data collected by a cellular provider, as well as additional user-contributed traces, measuring cache effectiveness; their results show that as much as 20% of mobile browsing traffic is redundant and could be avoided by better caching. Wang et al. [17] examine per-user traces and analyze top10 websites, finding that revalidation traffic hampers cache performance in high-latency mobile environments. In each case the focus has been on the cacheability of data in the mobile environment, rather than the system performance of cache operations.

Most recently Kim et al. [8] have examined the effect of storage devices on performance of mobile applications. Their work found that in many cases the limiting factor in performance, even for high network traffic applications like browsing, was the poor performance of flash storage devices rather than limited network bandwidth. Our work is motivated by this investigation, and extends their work to examine a specific application, web browsing, where significant changes may be made to optimize storage behavior.

Finally, although the idea of a log-structured cache is not new, we have not found it presented and examined in the academic literature. Badam et al. [4] mention a commercial caching product based on a log-structured design, yet they explicitly mention that the product website has disappeared and there are no citable references.

6. CONCLUSIONS

Analysis of caching implementations all too often focus exclusively on counting hits and misses, ignoring the question of how fast the resulting system runs. In contrast our cache for the Chromium browser is designed to maximize the performance of the hit and miss operations themselves, speeding them up by almost an order of magnitude, at the cost of a modest reduction in hit rate. Further work will focus on quantifying the actual impact on user-visible performance, as well as investigating whether hit rate may be improved without significantly impacting storage performance.

Acknowledgments

We would like to thank Ricardo Vargas of Google Chrome team for answering our questions related to Chromium disk cache and helping us with performing experiments on Chromium's existing cache.

7. REFERENCES

[1] Chromium. http://www.chromium.org/Home.

[2] Firefox. http://www.firefox.com.

[3] J. S. S. T. Association. Embedded multimediacard(e-mmc) emmc/card product standard, high capacity. Technical Report JESD84-A441, Mar. 2010.

[4] A. Badam, K. Park, V. Pai, and L. Peterson. HashCache: Cache Storage for the Next Billion. In *NSDI*, 2009.

[5] S. Boboila and P. Desnoyers. Performance models of flash-based Solid-State drives for real workloads. In *IEEE Symposium on Massive Storage Systems and Technologies*, June 2011.

[6] M. Ingram. Mary meeker: Mobile internet will soon overtake fixed internet. http://gigaom.com/2010/04/12/mary-meeker-mobile-internet-will-soon-overtake-fixed-internet/, 2010.

[7] P. R. Jelenković and A. Radovanović. Optimizing LRU caching for variable document sizes. *Comb. Probab. Comput.*, 13(4-5):627–643, July 2004.

[8] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *FAST*, 2012.

[9] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash sys tems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[10] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, Dec. 2003.

[11] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Web caching on smartphones: ideal vs. reality. In *Proceedings of the*

10th international conference on Mobile systems, applications, and services, MobiSys '12, pages 127–140, New York, NY, USA, 2012. ACM.

[12] S. Ramachandran. Web metrics: Size and number of resources. https://developers.google.com/speed/articles/web-metrics, 2010.

[13] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:26–52, 1992.

[14] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *USENIX*, 1993.

[15] S. Souders. Cache them if you can. http://www.stevesouders.com/blog/2012/03/22/cache-them-if-you-can/, 2012.

[16] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.

[17] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How effective is mobile browser cache? In *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students, for the students*, pages 17–20, New York, NY, USA, 2011. ACM.