What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage?

Hyojun Kim
Georgia Institute of Technology
Altanta, GA
hyojun.kim@cc.gatech.edu

Moonkyung Ryu
Georgia Institute of Technology
Altanta, GA
mkryu@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
Altanta, GA
rama@cc.gatech.edu

ABSTRACT

Smartphones are becoming ubiquitous and powerful. The Achilles' heel in such devices that limits performance is the storage. Low-end flash memory is the storage technology of choice in such devices due to energy, size, and cost considerations.

In this paper, we take a critical look at the performance of flash on smartphones for mobile applications. Specifically, we ask the question whether the state-of-the-art buffer cache replacement schemes proposed thus far (both flash-agnostic and flash-aware ones) are the right ones for mobile flash storage. To answer this question, we first expose the limitations of current buffer cache performance evaluation methods, and propose a novel evaluation framework that is a hybrid between trace-driven simulation and real implementation of such schemes inside an operating system. Such an evaluation reveals some unexpected and surprising insights on the performance of buffer management schemes that contradicts conventional wisdom. Armed with this knowledge, we propose a new buffer cache replacement scheme called *SpatialClock*.

Using our evaluation framework, we show the superior performance of SpatialClock relative to the state-of-the-art for mobile flash storage.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*; D.4.2 [**Software**]: Operating Systems—*Storage Management*

Keywords

buffer management, page cache, flash storage

1. INTRODUCTION

Over the past decade, mobile computing devices, particularly smartphones, are finding increasing use in our daily lives. According to a recent Gartner report, within the next three years, mobile platforms will surpass the PC as the most common web access device worldwide [10]. By 2013, over 40% of the enhanced phone installed-base will be equipped with advanced browsers [28].

Although mobile systems have become unbelievably popular today, only few studies have been conducted for deep understanding of mobile systems. Mobile systems are not just miniatures of personal computer systems, and thus, the previous research insights from desktop and server systems should not be simply applied to mobile systems without careful reexamination. For example, a recent study reveals that the storage subsystem has a much bigger performance effect on application performance on smartphones than it does on conventional computer systems [21]. Considering the rapid growth of mobile systems, it is time to move our focus to the mobile system components.

CPU power and main memory capacity are increasing very fast; the latest smartphone has a quad core 1.4 GHz processor (Tegra3 [26]) as well as 1 GB of main memory capacity. On the other hand, the technology used for storage on mobile platforms lags significantly behind that used on regular computers. Flash storage is the norm for smartphones because of the limitations of size, cost, and power consumption. Flash storage exhibits very different performance characteristics relative to the traditional Hard Disk Drive (HDD); plus current operating systems (owing to their legacy of assuming HDD as the primary storage technology) are not engineered to support flash storage adequately. Consequently, flash storage is the Achilles' heel when it comes to performance of mobile platforms [21]. While high-end flash based Solid-State Drives (SSDs) are available and used in regular and enterprise class machines, adoption of such storage for mobile platforms is infeasible for reasons of cost, size, and energy consumption. Write buffering can be used to solve the small write problem both in enterprise storage servers using disk arrays [11] and inside flash storage devices [22]. However, this solution approach has its limitations, especially for a smartphone environment for a variety of reasons, most notably, loss of reliability due to unexpected power losses (e.g., battery getting unhitched from the phone due to accidental dropping of the phone). Therefore, we argue that operating system level software support is critically needed for low-end flash storage to achieve high performance on mobile platforms, and thus, we focus our attention on inexpensive flash storage in this study. Specifically, we are con-

SIGMETRICS'12, June 11–15, 2012, London, England, UK.

cerned with the buffer cache replacement schemes for mobile platforms using inexpensive flash storage.

OS buffer cache is the focal point for actions regarding how to enhance the performance for OS generated write operations to the storage device. Therefore, we are interested in revisiting the page replacement algorithm used by the OS buffer cache. The primary goal of the OS buffer cache is ensuring a good hit-ratio for the subsystems that sit on top of it. It is well-known that Least Recently Used (LRU) or some variant thereof that preserves temporal locality is a good choice for a page replacement algorithm from the point of ensuring a good hit-ratio. However, such algorithms tend to be agnostic about the performance characteristics of the primary storage backing the buffer cache.

The first step is to take stock of the state-of-the-art in buffer cache management schemes proposed and or used in current operating systems and evaluate their efficacy for flash storage used in smartphones. LRU , Clock [5], Linux2Q [6] are three well-known *flash-agnostic* buffer cache replacement schemes. Clean First LRU (CFLRU) [27], LRU Write Sequence Reordering (LRUWSR) [19], Flash based Operation aware Replacement (FOR) [24], Flash-Aware Buffer management (FAB) [17] are previously proposed four *flash-aware* buffer cache replacement schemes. Even though most of these proposed schemes are aiming for general flash storage rather than the inexpensive ones found in mobile platforms, they are a step in the right direction. We would like to understand the performance potential of these schemes (both flash-agnostic and flash-aware ones) for mobile flash storage.

What is the best evaluation strategy for answering this question? Analytical modeling, simulation, and real implementation are the traditional approaches to performance evaluation of computer systems. Specifically, in the context of OS buffer cache replacement schemes, two techniques have been extensively used: *real implementation* in an operating system, and *trace-driven simulation*. Clearly, real implementation inside an operating system would reveal the true performance potential of any buffer cache replacement scheme. But such an approach is fraught with a number of difficulties and downsides. A real implementation of even a single scheme would require a huge amount of effort. This is because changing the core functionality of an operating system such as the buffer cache replacement scheme is non-trivial since it affects all the subsystems that live on top of it (e.g., VM, file systems). Also, to have a side by side comparison, such an approach would require the implementation of all the competing schemes in the operating system. Besides these difficulties, there are also downsides to this approach. It may be difficult to assess the true performance benefit of the scheme being evaluated due to performance noise from other parts of the operating system. Further, it would be difficult to ask a variety of "what if" questions with a real implementation, without re-engineering the implementation to answer such questions. Perhaps, most importantly the barrier to trying out new ideas will be too high if it has to be implemented first in an operating system to get an estimate of its performance potential. It would stifle creativity.

Trace-driven simulation has been extensively used for buffer cache related studies sometimes together with real implementation [14, 15, 27], and many other times just by itself [8, 12, 16, 17, 18, 23, 24, 25]. Typically, storage access traces are collected first from real applications on an existing system or synthesized from a workload model of applications. These traces are then used as inputs to a buffer cache simulator to gather metrics of interest for performance evaluation. Hit-ratio is the most popular metric, but some studies also measure the I/O operation completion time. The virtues of trace-driven simulation include time to getting useful results compared to real implementation, repeatability of results, isolation of performance benefits from other noises, and the ability to have useful "what if" knobs (additional input parameters) in addition to the traces serving as the workload for the evaluation. However, the main drawback of trace-driven simulation is that the results may not accurately capture all the metrics of interest pertinent to the real system. This is especially true with flash storage due to the complicated and often opaque mapping layer inside such devices. Device manufacturers do not publish such internal details; thus these devices have to necessarily be treated as black boxes. Therefore, any simulator can only make a best effort guess as to what is happening inside the device making the veracity of trace-driven simulation results for flash storage questionable.

In this paper, we propose a novel buffer cache evaluation framework, which is a hybrid between trace-driven simulation and real implementation. Basically, our method expands the existing trace-driven simulation by adding one more step with a *real* storage device. It allows us to see the real performance effect of cache replacement schemes without actually implementing the algorithm into an operating system.

We collect *before-cache* storage access traces from a real Android smartphone while running popular applications such as Web browser and YouTube player. By using the traces and our proposed evaluation framework, we evaluate seven state-of-the-art buffer cache replacement schemes, and report very surprising results. The most previously proposed flash-aware schemes are not better (sometimes much worse) than flash-agnostic schemes at least with the smartphone workloads that we have evaluated them with. A careful analysis of the results using our new framework reveals the source of this disconnect between previous studies and our surprising new results, namely, not respecting *spatial adjacency* for write operations to inexpensive flash storage. Armed with this new insight, we propose a new buffer cache replacement scheme called *SpatialClock* for mobile flash storage. By comparing SpatialClock to the state-of-the-art buffer cache replacement schemes using our evaluation framework, we show that SpatialClock delivers superior storage performance on real mobile flash storage while not degrading the cache hit-ratio.

We make the following three contributions through this work. First, we propose a new buffer cache evaluation framework. Second, we collect *before-cache* storage access traces from an Android platform, and make them available for other researchers[1] The third and final contribution is the SpatialClock buffer cache replacement algorithm.

The rest of the paper is structured as follows. Section 2 explains previously proposed flash-aware cache replacement schemes. Section 3 introduce the new cache evaluation framework, and Section 4 describes SpatialClock algorithm. Section 5 and Section 6 provide evaluation results and our conclusions.

[1]https://wiki.cc.gatech.edu/epl/index.php/S-Clock

2. FLASH-AWARE CACHE SCHEMES

Ever since the appearance of NAND flash memory based solid state storage devices, multiple flash-aware buffer cache replacement schemes have been proposed. One of the earliest schemes is CFLRU [27]. Due to its very nature, flash memory incurs less time for servicing a read operation in comparison to a write operation. In other words, to reduce the total I/O operation time of flash, it is desirable to reduce the total number of write operations. To this end, CFLRU tries to evict a clean page rather than a dirty page because a dirty page must be written back to the storage during the eviction process. However, such a biased policy could fill the cache with mostly dirty pages at the expense of not holding frequently accessed clean pages. This is clearly undesirable as it will bring down the overall cache hit-ratio. To mitigate this problem, CFLRU divides the LRU cache list into two parts (as done in the 2Q algorithm [18]), and applies the clean-first policy only to the lower part of LRU list. The authors claim that choosing the partition size intelligently will result in shorter total operation time for flash storage.

LRUWSR [19] shares the same motivation with CFLRU. It also tries to give higher priority to dirty pages, but it uses a different method. Instead of partitioning the LRU list into two parts, LRUWSR adds a *cold bit* to each cache frame, and gives a second chance to a dirty page frame to remain in the cache. When a page is selected as a potential victim, its dirty and cold bits are checked first. If the page is dirty and its cold bit is zero, then the algorithm decides not to choose it as a victim. Instead it sets the cold bit for this page to indicate that the page has gotten its second chance to stay in the cache. The authors argue that LRUWSR is better than CFLRU both in terms of usability (because it does not require any workload dependent parameters) and performance.

One of the latest flash-aware cache replacement scheme is FOR [24]. It also focuses on the asymmetric read and write operation time of flash storage. In addition, it combines *inter operation distance (IOD)* (the core idea of the Low Inter Reference Recency [16]) together with the recency attribute of the LRU algorithm. IOD is the number of distinct operations performed between two consecutive "identical operations" on one page including the operation itself. Further, FOR calculates the IOD and recency values separately for read and write operations. By considering all these factors, FOR calculates the weight of each page, and evicts the page having the minimal weight value. The results reported by the authors show 20% improvement for database workloads for FOR over other schemes.

Flash-Aware Buffer (FAB) management scheme [17] was proposed especially for portable media players to generate more sequential writes. It clusters cache pages in blocks and maintains an LRU list in units of blocks. By choosing and flushing the victim block having the highest number of pages, FAB produces large sequential writes, which are desirable for flash storage. To verify the performance effect of existing cache replacement schemes on mobile systems, we decided to evaluate these four flash-aware schemes as well as three flash-agnostic schemes: LRU, Clock, and Linux2Q. Linux2Q is the page replacement scheme used in Linux kernel. Compared to the original 2Q algorithm [18], it introduces the *active bit*, which makes the transition between the active and inactive queues more strict.

3. A NOVEL CACHE EVALUATION FRAMEWORK

Flash memory is different from conventional magnetic storage. To overcome the physical limitations of flash storage, every flash storage device includes a Flash Translation Layer (FTL) [13, 20] in addition to the storage elements. FTL is a special software layer emulating sector read and write functionalities of an HDD to allow conventional disk file systems to be used with flash memory without any modifications. FTLs employ a *remapping technique* to use the storage cells more judiciously. When FTL receives a request to overwrite a sector, it redirects the new content to an empty page, which is already erased, and modifies the mapping table to indicate the new physical page address where the logical sector has been written.

The FTL algorithm employed by a flash storage device is usually opaque, making it difficult to simulate a flash storage device. The performance of a flash storage device critically depends on the algorithms used internally by the FTL. Thus it is pretty much impossible to accurately simulate the internal architecture of a flash device for performance evaluation purposes. While it is always possible to simulate a given FTL algorithm (assuming the details are known) for a specific flash storage, it is impossible to generalize and use it for other flash devices that we want to compare it against, since the performance characteristics of a flash device is so intimately tied to its specific FTL algorithm. Trace-driven simulation may still be good enough to understand the performance potential of a buffer cache management scheme with respect to certain metrics of interest (e.g., hit-ratio). However, we believe, it is not good enough for evaluating all the metrics of interest (e.g., completion time of I/O operations) in understanding the performance potential of a buffer cache scheme on a specific flash storage.

3.1 Evaluation Framework

The new buffer cache evaluation framework (Figure 1) is a hybrid between trace-driven simulation and real implementation. First, we give the big picture and then drill down to the details. We collect traces of read/write accesses to the buffer cache from an Android platform running popular Apps. We call this the *before-cache* traces. We use these traces as the workload on a simulator that implements the seven different buffer cache management policies that we alluded to in the previous section. The output of this simulator is two-fold: *hit-ratio* for the chosen cache scheme; and *after-cache* storage access trace for each cache scheme. The latter is the sequence of read/write requests that would be issued to the actual storage since the buffer cache does not have these pages. We have developed a tool called *Workload Player* that takes the *after-cache* trace as its input, sends the read/write requests in the trace to a real storage device, and reports the total elapsed time for performing the read/write operations. Since the times gathered by the Workload Player are the actual elapsed times for the requests, they account for the internal architecture of the flash storage on which the trace is being played. Thus, this hybrid evaluation framework is a faithful reproduction of the combined performance characteristic of the buffer cache algorithm and the real flash storage.

To contrast our hybrid approach to the traditional trace-driven simulator, the latter stops with reporting the observed hit-ratio for a given scheme and generating the ac-

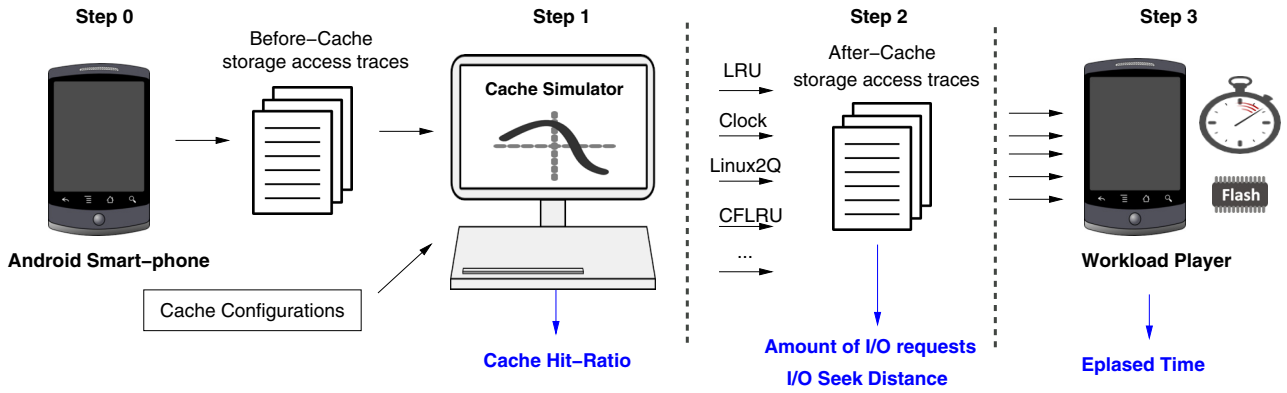

Figure 1: The evaluation framework for buffer cache replacement schemes: *Traditional trace-driven simulations stop with Step 1 or 2. Enhancing the framework with Step 3 allows us to do an accurate evaluation of the performance implications of different cache replacement schemes on real flash storage devices.*

tual storage access traces corresponding to the misses. Some studies may take the actual storage access traces to compute the expected completion of I/O requests based on published static read/write/block-erase times of a given flash storage device. By actually playing the *after-cache* traces on a storage device we are able to get the real I/O completion times.

To collect the *before-cache* traces, we have instrumented the Android OS running on a smartphone. We have used Google Nexus-One smartphone with Android version 2.3.7 Gingerbread. This in itself is a non-trivial piece of engineering. We modified the *init procedure* of Android for our purpose, and also patched Linux kernel page cache related sources. We developed the buffer cache simulator for the seven schemes discussed in Section 2 to run on a standard desktop Linux platform. The Workload Player runs both on the smartphone and the desktop, and collects performance statistics of playing the *after-cache* traces on flash storage. Our experimental setup includes multiple smartphone storage devices to run the *after-cache* traces.

3.2 A Surprising Result

We first present a representative result for the seven buffer cache management schemes using a microSD card on Google Nexus-One phone. The traces are generated from mobile Apps (such as web browsing and video streaming) running on the Android phone. The result is summarized in Table 1. This result is a good sample of the overall trend we observed for most of the test cases (different flash storage devices and different cache sizes), which we elaborate in Section 5 (with more details about the workloads that are used to generate the traces). At this point our goal is to reveal a surprising result that emerged from the evaluation of these seven schemes.

All the schemes show remarkably higher hit-ratios with 64 MB cache size than with 4 MB cache size as expected. However, despite the fact that higher hit ratios implies a reduction in storage activity, the measured *elapsed times* on a real storage device tells a different story. Note for example from Table 1 that the measured *elapsed times* for LRU, Linux2Q, CFLRU, LRUWSR, and FOR are worse than those for Clock and FAB. This is surprising because hit-ratio is the overarching metric used to evaluate the effectiveness of buffer cache replacement schemes.

Besides, all four flash-aware schemes fail to reduce the number write operations. Recall that the main focus of flash-aware schemes (except FAB) is to reduce the number of write requests to the storage device. Given this focus, it is interesting that the amount of write operations generated by the flash-aware schemes is not that different from the flash-agnostic ones.

What is interesting from this representative result is that we cannot differentiate the relative merits of these cache replacement strategies using a conventional metric such as hit-ratio. Incidentally, this result also highlights the limitation of a pure trace-driven simulator since hit-ratio and read/write traffic to the storage device are the metrics that can be generated by such a simulator.

Our hybrid approach helps understand the performance potential of the schemes better by letting us measure the elapsed time for playing the *after-cache* traces on a real flash storage device. Surprisingly, three of the four flash-aware schemes show slower performance than the Clock scheme. This is interesting because Clock is not designed for flash storage, and the observed performance differences cannot be explained either by hit-ratios or by the number of generated read/write operations. The surprising result is the fact that the latest flash-aware schemes are not performing as well as one would have expected on a mobile flash storage (Section 5 shows this is true for a variety of flash storage devices). More importantly, this result establishes our first contribution in this paper, namely, the power of the hybrid evaluation framework for performance analysis of buffer cache replacement strategies for flash storage

3.3 Explaining the surprising result

A more intriguing question is why the three out of the four flash-aware schemes are not performing as well as one would expect. A careful investigation reveals the source of this puzzling anomaly. The short answer to this puzzle is *not respecting spatial adjacency* for the write requests that are generated to the flash storage. To fully appreciate this phenomenon, we need to understand some basics of flash storage.

Flash storage is based on semiconductor technology, and hence shows very different performance characteristics when compared to the traditional magnetic disk. A number of studies have reported on the special performance character-

Table 1: Evaluation results for seven cache replacement schemes: Mixed workload on Patriot 16GB microSDHC card / Nexus One with 4 / 64MB cache size

		LRU	Clock	Linux2Q	CFLRU	LRUWSR	FOR	FAB
4MB	Hit-Ratio	0.7592	0.7583	0.7665	0.7584	0.7580	0.7529	0.7497
	Generated Read Operation Count	13,735	13,917	12,709	14,067	14,125	15,418	14,549
	Generated Write Operation Count	44,600	44,650	44,096	44,413	44,450	44,261	46,131
	Measured Elapsed Time (second)	**234.69**	**234.78**	**261.69**	**241.68**	**229.78**	**236.62**	**292.48**
64MB	Hit-Ratio	0.8863	0.8860	0.8876	0.8860	0.8857	0.8829	0.8860
	Generated Read Operation Count	1,861	2,745	1,605	1,909	1,927	2,523	1,681
	Generated Write Operation Count	26,064	25,833	25,998	26,062	26,060	26,120	26,327
	Measured Elapsed Time (second)	**236.86**	**183.57**	**277.44**	**237.87**	**231.43**	**291.22**	**129.30**

Table 2: Comparison of an HDD (3.5" 7200 RPM HDD) vs. three flash storage devices (Patriot 16GB microSD, two eMMC devices used in Nokia N900 and Google Nexus-S smartphones (KB/sec): *write ordering is important but read ordering is not important on flash storage.*

Storage	Read(KB/sec)		Write(KB/sec)	
	Sorted	Scattered	Sorted	Scattered
HDD	6,498.4	537.6	4,836.6	1,004.0
microSD	4,852.7	4,836.6	545.2	8.3
eMMC-1	5,100.1	4,444.6	470.9	16.1
eMMC-2	3,124.5	2,551.5	566.4	259.1

istics of flash storage [4, 7, 9]. It is well known that flash storage devices show a relatively low write-throughput for small, scattered (random) requests and a higher throughput for large, sequential write requests. At the same time, they are insensitive to the order of read requests, showing almost unchanging performance for sequential and random read requests. We ourselves have performed simple measurements to identify these differences in performance.

Table 2 compares the measured read and write throughput of an HDD and three flash storage devices. We use four synthetic workloads. All four workloads use the same number of requests (32,768), with request sizes of 4KB (typical page size in most virtual memory systems) within a 1 GB address space (typical process virtual address space). Two of these workloads use random read and write requests, respectively. The remaining two workloads use, respectively, read and write requests that are sorted by the sector number (thus resulting in accessing sequentially ordered sectors on the storage device).

On an HDD, both read and write throughputs are highly influenced by the sequence of the requests because it has seek-delays of the mechanically moving magnetic head. In contrast, the read throughput of flash storage is not much influenced by request ordering because there is no seek delay. For the write requests, flash devices show uniformly lower throughput than the HDD, and their scattered write throughputs are lower than the sorted write throughputs even though there is no moving parts inside flash storage devices. The reason for the lower throughput for scattered writes is due to the way data updating happens internally in a NAND flash memory chip. In other words, not respecting the spatial adjacency for consecutive write requests can result in a huge performance penalty in flash storage. This result suggests that write request ordering can make a huge performance difference, and the disparity between sorted and scattered writes demonstrates that the *ordering* is perhaps more important than the *number* of write requests.

Of the flash-aware schemes evaluated, only FAB respects spatial adjacency while the others (CFLRU, LRUWSR, and FOR) are focused on reducing the total number of write requests to the flash storage. As Table 2 shows, the penalty for ignoring spatial adjacency (i.e., sending scattered write requests to the storage) is huge. This is the reason we see elapsed time differences among the cache replacement schemes even though they all generate roughly the same amount of read/write requests (see Table 1). As is evident from Table 1, FAB has the least elapsed time compared to the other schemes (for 64 MB cache size) since it is the only scheme that explicitly cares about spatial adjacency. However, FAB, owing to its focus on supporting media player workload, is biased too much towards write performance optimization for flash storage to the detriment of overall buffer hit-ratio, which is an important figure of merit for a general-purpose OS buffer cache. This becomes apparent especially at smaller cache sizes and other diverse workloads. For example, it can be seen in Table 1 that FAB has the worst performance (both elapsed time and hit ratio) with a 4 MB cache size compared to the other schemes. Further, it has some inherent complexities for implementation as a general-purpose OS buffer cache scheme, which we discuss in Section 5.5.

4. SPATIALCLOCK

We have seen that even the flash-aware general-purpose OS buffer cache replacement algorithms proposed thus far do not pay attention to the spatial adjacency (or lack thereof) of the pages being evicted from the OS buffer cache (FAB is an exception but as we noted earlier it does this at the expense of cache hit-ratio and so it is not general-purpose enough for OS buffer cache). Given the discussion in Section 3.3, this is a missed opportunity that hurts the performance of flash storage. Therefore, we propose a new algorithm *SpatialClock* that respects the spatial adjacency of the pages being evicted from the OS buffer cache without loosing cache hit-ratio remarkably.

There are two important points we want to address head on before we delve into describing SpatialClock:

1. Page replacement algorithms are an age-old topic. However, from the point of view of the storage technology that is currently being used and will be used for the foreseeable future in mobile platforms, we believe it is time to revisit this topic. From our discussion Section 3.3, we can distill a couple of observations regarding

flash storage that strengthen our belief: (a) respecting spatial adjacency for writes is very important, and (b) read and write operations are independent of each other due to the nature of the flash technology, in contrast to traditional storage devices such as an HDD (due to the absence of the mechanical head movement).

2. The OS buffer cache is deeply entrenched in the software stack of the operating system. Therefore it is not prudent to overburden this layer with device-specific optimizations (such as write reordering), which would require a significant architectural change of the entire OS software stack. What we are proposing in this section is a localized change only to the page replacement algorithm of the OS buffer cache, which does not affect the other functionalities of this layer. Further, as we will see shortly, the proposed algorithm is not storage specific; it merely respects the *logical* spatial adjacency of the pages being evicted from the buffer in addition to temporal locality

4.1 Main Idea

The key question is how to design a new page replacement scheme to achieve the two different objectives simultaneously: high cache hit-ratio and sequentially ordered write requests. One commonly used approach is dividing cache memory space into multiple partitions, and applying different cache management policies to the distinct cache partitions [15, 25, 27]. However, this partitioning approach introduces another difficult problem: how to adaptively adjust the partition sizes for various workloads. Further, such a partitioning approach is too major a change to this critical layer of the operating system. Therefore in designing SpatialClock, we take a different approach rather than partitioning the OS buffer.

Before we describe SpatialClock, let us briefly review LRU and Clock algorithms. In LRU, page references are kept in a sorted temporal order by the OS buffer cache. When a page frame is accessed, the frame needs to be moved to the Most Recently Used (MRU) position. The operation may require obtaining a global lock to protect the data structure from concurrent accesses. Because page references are very common, such frequent rearrangement of the data structure is expensive. Further, true LRU is difficult to implement in practice since it requires hardware assistance at individual memory reference granularity to track page frame accesses from the VM subsystem. Nevertheless, true LRU is used in memory system studies as a *standard* to compare other practical page replacement algorithms.

Clock is an approximation to the true LRU algorithm and is often referred to as *second-chance* replacement algorithm. Clock relies on a simple hardware assist common to all processor architectures supporting virtual memory, namely, a per-page *reference bit* (usually part of the page table entry for that page). The reference bit can be set by the hardware and cleared by the software. The hardware sets the associated reference bit when a page frame is accessed unbeknownst to the software (i.e., the operating system). The Clock algorithm keeps the page frames as a circular list in FIFO order of their arrival into the OS buffer cache from the storage device. The victim selection works as follows. The algorithm sweeps the circular list of page frames skipping over the frames whose reference bits are set (clearing the reference bits as it sweeps) and stops at the page frame

```
1  /* Victim selection */
2  pageframe*
3  spatialclock_choose_victim()
4  {
5    /* sweeping until find a victim */
6    while (1)
7    {
8      /* circular movement */
9      if (cur_pointer == NULL)
10       cur_pointer =
11         avltree_move_to_first();
12
13     victim      = current_pointer;
14     cur_pointer =
15       avltree_move_to_next(cur_pointer);
16
17     if (victim->referenced == 0) break;
18     victim->referenced = 0;
19   }
20   return victim;
21 }
```

Figure 2: SpatialClock victim selection algorithm

whose reference bit is not set. This page frame is chosen as the victim for eviction from the OS buffer cache. Clock does not keep the precise reference order like LRU; hence it is simpler and does not require global locks for maintaining its data structure. Despite its impreciseness in maintaining the page reference history, the good news is that the performance of Clock approximates LRU in most cases. Therefore, Clock has been widely used especially for virtual memory systems, which require low overhead lookup of the buffer cache.

In SpatialClock, we follow the basic rules of the Clock algorithm with only one difference. Page frames are arranged by the *logical* sector number of the storage system that contains the page frame. Consequently, page frames are chosen as victims for eviction in the sequential order of the sectors that contain these frames, and thus, frames are implicitly chosen sequentially with respect to the storage device. This results in preserving/respecting spatial locality during victim selection when the chosen frames happen to be dirty as well. Figure 2 shows the victim selection algorithm of SpatialClock.

Figure 3 shows an example of the victim selection in the SpatialClock algorithm. Each row represents a page frame, and the left cell in the row represents the containing sector number for that page frame, the right cell indicates the reference bit value. The page frames are pre-arranged in sorted order with respect to the containing sector numbers for the page frames. To choose a victim page, page frames are scanned from the current pointer position to find a page frame, which has a '0' for the reference bit value. In the given example, the sweep stops at page frame whose containing sector number is 80, while clearing the reference bits of the page frames having 40 and 72 as the containing sector numbers, respectively.

Respecting and checking the reference bits gives SpatialClock the advantage of an approximate LRU for victim selection. Arranging the page frames in a spatially adjacent manner gives an opportunity to enforce write ordering for the evicted page frames. Giving more importance to phys-

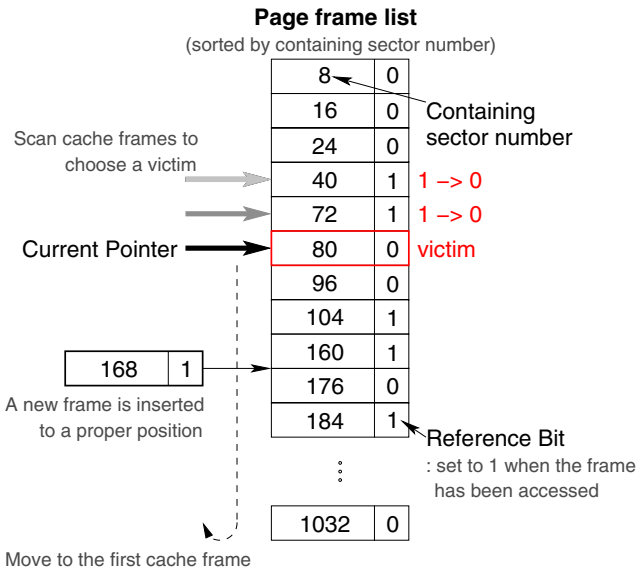

Figure 3: Victim Selection in SpatialClock: *SpatialClock maintains and scans page frames in the order of the containing sector numbers to generate ordered write requests.*

ical adjacency than the recency of access could affect the hit-ratio. However, our evaluation results show that this is not the case at least for the traces we studied. More importantly, we argue that paying attention to the elapsed time for storage access is crucial for achieving good performance on flash storage.

Compared to the original Clock scheme, SpatialClock requires maintaining page frames in a sorted manner, and we use an AVL tree [3] for the purpose. However, the burden is only for a page frame insertion operation, which is a relatively rare operation as long as the hit-ratio is high. The more common reference operation of the OS buffer cache remains exactly the same as in the original Clock algorithm. Besides, the AVL tree can be used for page look up purpose, which is mandatory for buffer cache maintenance.

We have implemented SpatialClock using an AVL tree to bound the page frame insertion time to be $log\ N$, where N is the number of page frames. We associate logical sector numbers with each frame (obtained from the storage map maintained by the OS buffer cache) to organize the circular list respecting spatial adjacency.

5. EVALUATION

We focus our evaluation on the following two key points:

- Hit-Ratio Comparison: SpatialClock is designed to produce sequentially ordered write requests, and often disobeys the philosophy of Clock and LRU policies. Will it degrade cache hit-ratio remarkably? We will answer this question by comparing cache hit-ratios with multiple traces and cache sizes.
- Performance effect on flash storage: We will verify the performance effect of SpatialClock on real flash storage devices.

We compare SpatialClock head to head with the seven schemes we introduced already: (1) LRU, (2) Clock, (3) Linux2Q, (4) CFLRU, (5) LRUWSR, (6) FOR, and (7) FAB.

5.1 Experimental Setup

5.1.1 Trace Collection from Android Smartphone

We collected *before-cache* storage access traces from a real Android smartphone. Even though many disk access traces are available in the public domain, most of them are *after-cache* traces, and some traces used in previous studies (for example, LIRS [16]) do not separate read and write accesses. More importantly, we want to use the traces that came from a real smartphone while running popular mobile Apps.

We used a Google's Android reference phone, Nexus-One [2] with Android Open Source Project (AOSP) [1] 2.3.7, Gingerbread version. We modified the init procedure of Android to use the partitions on an external microSD card with EXT3 file system instead of the internal NAND flash memory with YAFFS2 file system because it is not possible to collect general block level traces from YAFFS2 file system. We also had to modify the page cache related parts of Linux kernel (version 2.6.35.7) to collect the *before-cache* accesses.

Since we do not have standard workloads for smartphones, we have tried to use typical workloads for data usage on smartphones. Of the three workloads used in this study, the first two, namely, W1 (web browsing) and W2 (video streaming), are very typical but with vastly different characteristics in terms of storage access. The third workload is a mix of several popular applications that are run today on smartphones.

- **W1: Web Browsing.** We collected storage access traces while doing web browsing for multiple hours. Web browsing may be the most common activity on today's mobile platforms such as smartphones and Internet tablets. When we visit web pages, web browser downloads web resources like image files into local storage to reduce network traffic. Therefore, while doing web browsing, small files are continually read and written, and storage performance influences user's web browsing experience. The collected amount of traces is smaller than our expectation because Android web browser is directed to the mobile web pages, which are optimized to minimize network and I/O traffic.
- **W2: Video Streaming.** We collected storage access traces while watching various YouTube video clips for multiple hours. When we watch Internet streaming video like YouTube, video data are buffered into local storage to provide stable video watching quality. This is another very popular activity on mobile platforms, and generates very different storage access pattern compared to web browsing. This workload has the highest amount of write traffic among the three workloads studied.
- **W3: Mixed App Workload.** In this workload, we collected storage access traces for several hours while running multiple Apps sometimes together and sometimes separately. Following Apps were used: Facebook, Twitter, Maps, Pandora, Angry Birds (game), Fruit Ninja (game), OfficeSuite, Camera, Internet Browser, YouTube, Gallery, Android Market, etc. We believe this workload is the best one to reflect a realistic usage of the smartphone.

Table 3 shows the number of read and write operations in the collected mobile traces. Note that there are only few read requests in the video streaming trace (W2). This could

Table 3: Trace Information

	Read Operation		Write Operation	
	Count	Amount (MB)	Count	Amount (MB)
W1	23,666	92.4	47,350	185.0
W2	67	0.3	387,701	1,514.5
W3	134,910	527.0	105,796	413.3

Table 4: Flash Storage Devices

Smartphone	Type	Chip Maker	Size
Nexus One	microSDHC	Patriot(Class 10)	16 GB
N900	eMMC	Samsung	30 GB
Nexus S	eMMC	SanDisk	15 GB

very well be due to the limitation of our trace collection method since it is not possible to collect in-memory accesses for a memory mapped file without hardware support[2]. Even though the collected traces may not be a perfectly faithful reproduction of the I/O activity in these Apps (since they are missing the accesses to memory mapped files), we note that this situation is unfortunately unavoidable and will happen even if we profile a real operating system. Thus, we believe that the traces are valid and proper for our evaluation. Besides, since all the cache replacement schemes are compared with the same set of traces, the comparison is fair.

5.1.2 Parameters for Cache Replacement Schemes

Some of the buffer cache replacement schemes require setting some algorithm specific parameters. For Linux2Q, we set the active vs. inactive queue ratio to be 3:1 (this is similar to the setting in the Linux kernel). For CFLRU, the Clean-First window size is set to be 25% of total cache size. For FOR, we use an alpha value of 0.5 as recommended by the authors of the paper, and read and write operation cost as 100us and 800us, respectively. Lastly for FAB, we set the number of pages per block as 64, which is the same as in the author's own evaluation of their scheme.

5.1.3 Flash Storage Devices

Two different types of flash storage are popularly used in smartphones today: microSD cards and eMMC devices. Due to space limitation, we choose to present the result of three devices. Table 4 shows the list of the chosen flash devices: one microSD card (with Google Nexus-one phone) and two (one slower and the other faster) eMMC devices (internal to each of a Nokia N900 and Google Nexus-S phones, respectively).

5.1.4 Workload Player

Workload Player simply receives a trace file, performs the I/O operations specified in the trace file on a real storage device, and reports the elapsed time for performing the operations. We run Workload Player on real smartphones. Google Nexus-One is used to evaluate the microSD card, and a Nokia N900 and a Google Nexus-S are used respectively, to evaluate their internal eMMC devices. The Workload Player is written as a regular C program running on Linux, and the buffer cache is bypassed by using `O_DIRECT` option.

[2]ARM processor in the Nexus-One phone does not provide this functionality.

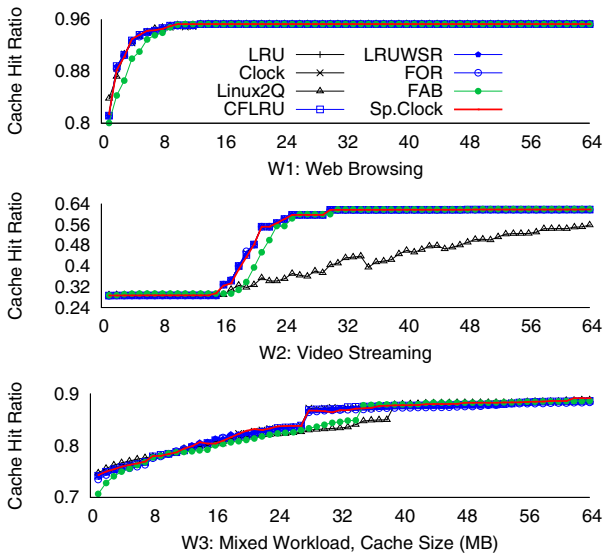

Figure 4: Hit-Ratio Comparison: *SpatialClock shows comparable hit-ratios to other schemes.*

5.2 Hit-ratio Comparison

Figure 4 shows simulated buffer cache hit-ratios[3] with 4-64 MB cache sizes[4] for the eight cache replacement schemes using the three traces. Except Linux2Q and FAB, other six schemes are not much different from one another in terms of hit-ratio. Linux2Q shows lower hit-ratio than the others for the video streaming workload (middle graph) while showing relatively higher hit-ratios when cache size is small. Meanwhile, FAB shows remarkably lower hit-ratios when the cache size is small. It will be very interesting to analyze the reason for the poor performance of Linux2Q and FAB but it is not the focus of this study. We can clearly verify that SpatialClock and other flash-aware schemes except FAB show comparable (not remarkably low, at least) hit-ratios to non-flash-aware schemes even though they sometimes disobey LRU philosophy for the sake of accommodating the performance quirks of flash storage.

5.3 I/O Operation Cost Analysis

Existing flash-aware buffer replacement schemes are mainly focusing on the asymmetric read and write operation costs. Prior cache replacement performance studies have calculated the total cost of the I/O operation by applying a simple mathematical equation using the differential read/write times. We have done a similar calculation. To this end,

[3]Cold-miss ratios are 0.05, 0.38, 0.11 for W1, W2, and W3, respectively; since most of the evictions stemming from misses are due to on-demand paging, there is significant opportunity for SpatialClock to improve the performance of the storage system.

[4]Even though modern smartphones have more than 512 MB main memory, a significant portion of the memory is used for other purposes such as kernel heap / stacks and application heap, and only a limited portion of the memory is available for page / buffer cache. Besides, our study shows that the hit-ratio saturates with 64MB cache size (except in one case: Linux2Q, W2) irrespective of the caching scheme or the workload. This is the reason for choosing 64 MB as the maximum cache size in our real storage evaluation.

we count the number of read and write operations for each buffer management scheme, and calculate the total cost by using a simple cost model for a flash chip as is done in the FOR paper [24] (100us and 800us for read and write I/O operations, respectively).

Figure 5 shows the calculated times. Similar to the hit-ratio comparison results, no obvious differences are seen except for the Linux2Q and FAB cases. Based on this calculated result, it would appear that none of the flash-aware algorithms (including SpatialClock) are any better in reducing the total I/O cost compared to the flash-agnostic ones. We show in the next subsection that such a conclusion would be erroneous.

5.4 Performance Effect on Real Flash Storage

It is hard to show write ordering effect of buffer cache replacement algorithms without using real flash storage devices. Therefore, we collect *after-cache* traces generated by the cache simulator (Figure 1), and play them on real smartphones.

Figure 6-8 show the elapsed time on real flash storage devices. The measured time is represented by the bars in the graph, and shorter bars imply better performance. Unlike the mathematically calculated performance result shown in Figure 5, we can see clear differences among the seven buffer replacement schemes through this exercise of running the *after cache* traces on real flash storage devices[5].

In each cluster of the graphs, the right most red bar represents the elapsed time of SpatialClock. In general across all the mobile flash storage devices used in the evaluation, SpatialClock shows remarkable performance gains with the Web Browsing workload (W1) and Mixed workload (W3). Linux2Q and FOR algorithms show very poor performances with the Video Stream workload (W2). SpatialClock is significantly better than Linux2Q and FOR but does not show any significant performance advantage over the other cache replacement algorithms for the Video Streaming workload (W2).

As already shown in Table 2, eMMC-2 in the Nexus-S smartphone is less sensitive to write ordering, and thus, SpatialClock shows the smallest performance gains for this flash storage (Figure 8) while it shows huge performance gains for eMMC-1 (Figure 7). This is because this eMMC-2 chip is specially designed to provide good random write performance similar to a high-end SSD, and hence the limited performance gain with SpatialClock. This result also implies that the interface of the eMMC devices to the host is not a differential point in the observed results. It is an interesting point to note that SpatialClock consistently performs better with the inexpensive microSD card than on the other the two eMMC chips, which appear to be high-end ones given their superior performance for dealing with scattered writes. One way of interpreting this result is that if the flash storage already is well positioned to handle scattered writes, then the additional performance advantage due to SpatialClock is small. However, SpatialClock does better than the other cache replacement schemes even on the eMMC-2 (24% re-

[5]Please note that the absolute calculated numbers in Figure 5 are very different from what we obtained from measurements shown in Figure 6-8. This is because the parameters used for the calculation (directly obtained from [24]) are most certainly different from the actual internal chip-level timing values to which we have no access.

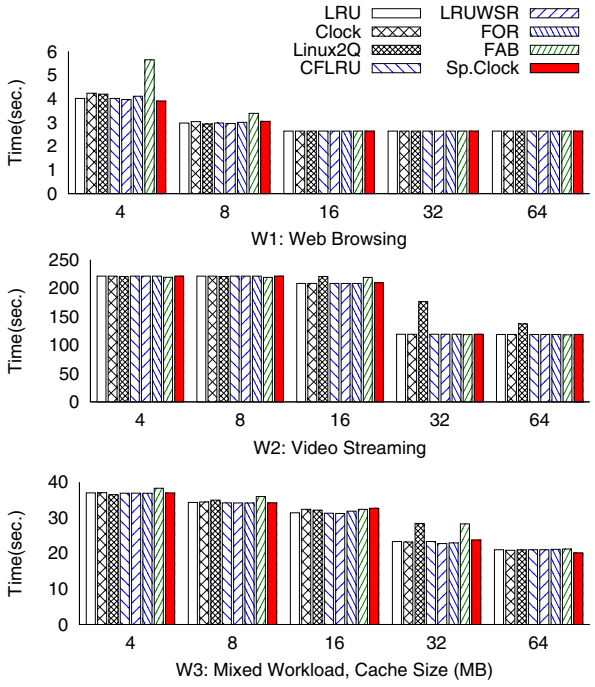

Figure 5: Calculated elapsed time based on the number of read and write operations in after-cache traces: *the results are almost indistinguishable for the different schemes.*

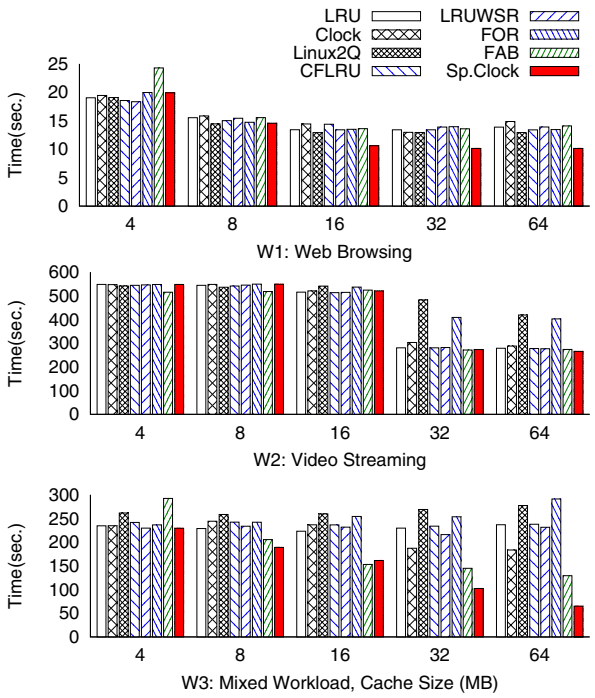

Figure 6: Patriot microSD: *SpatialClock shows -8.6-31.9%(W1), -6.4-43.4% (W2), and -5.4-77.7% (W3) elapsed time reduction compared to the other schemes.*

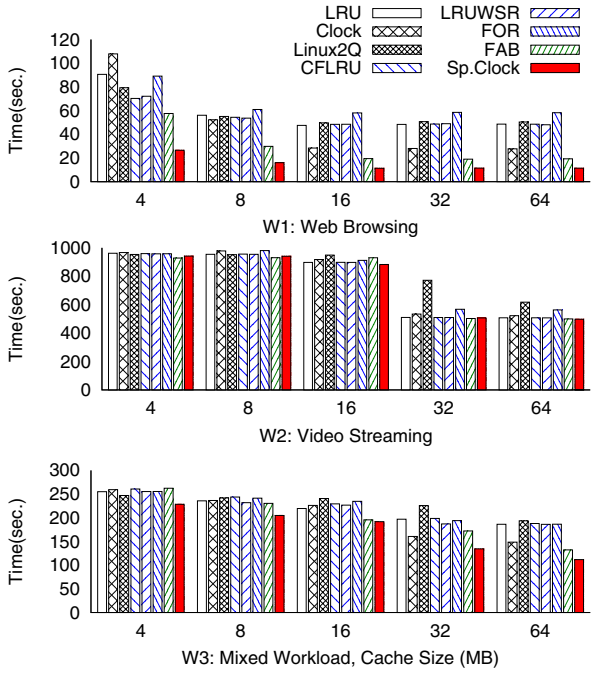

Figure 7: eMMC-1 (N900): *SpatialClock shows shows 0-80.5% (W1), -1.5-34.4% (W2), and 0-42.3% (W3) elapsed time reduction compared to the other schemes.*

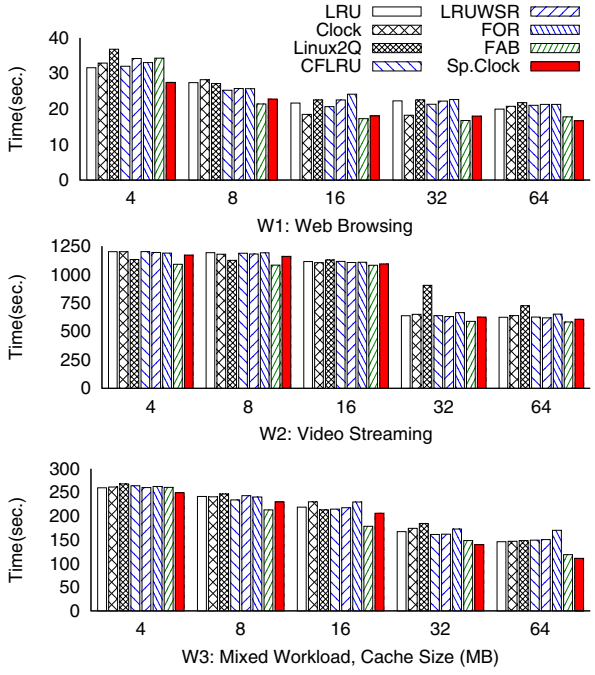

Figure 8: eMMC-2 (Nexus-S): *SpatialClock shows -7.4-25.4% (W1), -7.4-30.9% (W2), and -15.4-34.9% (W3) elapsed time reduction compared to the other schemes.*

duction in elapsed time compared to LRU for W3 workload and 64 MB cache).

There is another very surprising and interesting insight stemming from this performance result. With 64 MB cache and W3 workload the elapsed times for LRU are: microSD: 236.9, eMMC-1: 186.3, eMMC-2: 146.0 seconds. For the same configuration, the SpatialClock elapsed times are: microSD: 64.9, eMMC-1: 111.8, eMMC-2: 110.8 seconds. That is, the best absolute elapsed time (64.9 seconds) for this workload is achieved by using SpatialClock on the cheaper microSD card! Compare this with using the standard cache replacement scheme available in commercial operating systems running on a souped up flash storage such as eMMC and still being nearly 2.2 times slower than SpatialClock on an inexpensive flash. In other words, with the right OS support (SpatialClock), we can achieve better performance than using a hardware solution (eMMC) to circumvent the performance issues of mobile flash storage. SpatialClock is the third contribution of this paper, presenting a compelling case for revisiting the replacement scheme used for the buffer cache in smartphones.

5.5 Discussion

By using three different workload traces and three different flash storage devices, we show the superior performance of our proposed SpatialClock replacement scheme. We look more into some of the finer details concerning SpatialClock.

5.5.1 FAB vs. SpatialClock

According to the evaluation results, FAB shows comparable performances to our SpatialClock. FAB is also designed to respect spatial adjacency, which we know is very important on low-end mobile storage devices. However, there are important differences between SpatialClock and FAB from the point of view of a viable buffer cache replacement scheme for a general-purpose OS.

FAB clusters cache pages in blocks and maintains an LRU list in units of blocks. By choosing and flushing the victim block having the highest number of pages, FAB produces large sequential writes, which are desirable for flash storage. However, such a policy sacrifices temporal locality to maximize spatial locality, and results in lower cache hit-ratio as shown in Figure 4, and also can cause performance issue when cache size is small (see Figure 6, wherein it can be seen that FAB displays even worse performance than flash-agnostic schemes). Note that SpatialClock does not have hit-ratio issue nor performance issue with small cache size.

Besides, FAB would also incur a significant computation overhead in any reasonable implementation. Inspecting the pseudo-code provided by the authors of FAB [17], the implementation of the algorithm needs to check all blocks to find a victim block that has the most number of sectors to be written in it. This would undoubtedly result in a significant performance penalty especially with large caches. In other words, for small cache sizes FAB suffers from poor hit-ratio; and for large cache sizes FAB will result in significant performance penalties in terms of computation overhead. Interestingly, FAB reveals a limitation of our cache evaluation framework. As we pointed out, FAB has CPU computation overhead issue, but unfortunately our evaluation framework does not measure the computation overhead. Consequently in a real implementation, the performance results of FAB will be worse than what we have reported in this paper.

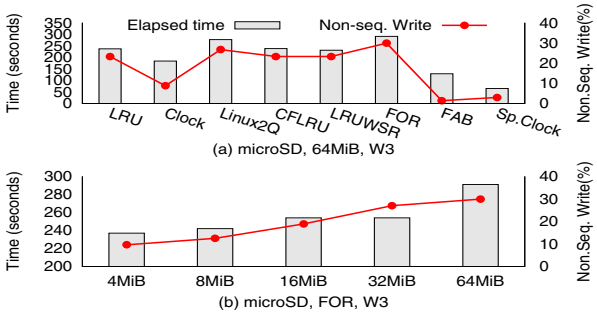

Figure 9: Correlation between the elapsed time and portion of non-sequential write operations in after-cache traces (microSD card)

5.5.2 *Result with Video Streaming*

SpatialClock shows significant performance gains with W1 (Web Browsing) and W3 (Mixed) workloads. However, for W2 (Video Streaming) workload we see almost no gain with SpatialClock. This is easily explained by understanding the video streaming workload. Recall that SpatialClock is geared to respect spatial locality for write requests to the flash storage. Most of the write requests generated by the video streaming workload are sequential in nature (except for a few metadata accesses). The traces are already flash-friendly, thus obviating the need for any special handling by the cache replacement algorithms.

5.5.3 *Verification of Sequentiality*

We claim that SpatialClock generates more sequentially ordered write requests than the other schemes, and that is the reason why it achieves better performance. How can we verify it is actually doing that? To answer this question, we analyze the generated traces for each cache replacement scheme for one chosen test-case (64 MB, W3, microSD). We count the number of non-sequential write operations from the generated trace, and show the portion of non-sequential writes as a percent of all the write requests together with the measured elapsed times in Figure 9 for the different buffer management schemes (Figure 9-(a)), and FOR (Figure 9-(b)). Figure 9-(a) shows that SpatialClock meets its design goal of respecting spatial adjacency for write requests.

5.5.4 *Performance Anomalies*

It is interesting to note the performance anomaly in Figure 9-(b) for the FOR cache replacement scheme. The figure shows that the performance (in terms of elapsed time for the I/O operations) actually becomes worse with larger cache size. This defies conventional wisdom! The reason once again can be linked to the increase in the amount of non-sequential writes to the storage, which as can be seen in Figure 9-(b), increases with the cache size. FOR chooses a victim page using a complex weighted function that takes into account operation cost, recency, and inter-operation distance. With such a choice, it is conceivable that the chosen victim page might be the best one for statistical reasons, but quite a bit distant spatially from the previously chosen victim, thus affecting the performance on a mobile flash memory.

In our studies, we noticed similar anomalies for the other flash-aware cache replacement schemes as well. In general, such schemes that give preferential treatment to dirty pages shielding them from eviction to reduce the I/O cost seem to work well when there is memory pressure. With large buffer caches, the memory pressure goes away and the lack of attention to spatial adjacency for writes in these schemes starts dominating the total I/O cost making them perform worse.

5.5.5 *Limitation of Our Evaluation Framework*

Although our evaluation framework allows us to see the performance effects of application workloads on real storage devices, it has its limitations. As we already mentioned in Section 5.5.1, it does not measure the computation overhead of the cache schemes being compared. Besides, the elapsed time for storage completion reported by our framework may not necessarily translate to comparable gains or losses at the application level, since performance at the application level is influenced by additional factors such as CPU computation overhead and user input.

6. CONCLUSION

Recent studies have shown that flash storage may be the performance bottleneck for the performance of common Apps on mobile devices. Due to size, power, and cost considerations, smartphones will continue to deploy low-end flash memories as the primary storage. Therefore, it is important to consider what can be done in the OS to enhance the performance of flash based storage systems. In particular, since the buffer cache is the point of contact between the upper layers of the OS software stack and the I/O subsystem, this paper re-examines the buffer cache replacement schemes with respect to their suitability for mobile flash storage. We make three contributions through this work. First, we develop a novel performance evaluation framework that is a hybrid between trace-driven simulation and real implementation. Second, we gather *before cache* storage traces for popular Apps running on an Android phone that can be used in the study of cache replacement schemes. We use this and the evaluation framework to study seven different cache replacement strategies. We made some surprising findings through this study, and the insight drawn from the study paved the way for our new buffer cache replacement scheme, SpatialClock. The key insight is the need to pay attention to spatial locality for writes to the flash storage to reduce the overall I/O time, a crucial metric to enhance the storage performance, and hence the application performance on smartphones.

Our future work includes considering the actual implementation of SpatialClock in the Android OS and fully understanding its performance potential in the presence of other OS entities such as the dirty page flusher.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments, which helped us tremendously in revising the manuscript. Our shepherd, Mustafa Uysal, has been very helpful in guiding us with the revision adding his own thoughtful comments augmenting those of the reviewers. We also thank the members of our Embedded Pervasive Lab for their support and feedback. This work was supported in part by the NSF award: NSF-CSR-0834545.

8. REFERENCES

[1] Android Open Source Project. http://source.android.com/index.html.

[2] Google Nexus One. http://en.wikipedia.org/wiki/Nexus_One.

[3] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proc. of the USSR Academy of Sciences*, 1962.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of the USENIX Annual Technical Conf.*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[5] A. Bensoussan, C. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15:308–318, 1972.

[6] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of the international joint conf. on Measurement and modeling of computer systems*, SIGMETRICS'09, pages 181–192, New York, NY, USA, 2009. ACM.

[8] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. of the international conf. on Measurement and modeling of computer systems*, SIGMETRICS'00, pages 286–295, New York, NY, USA, 2000. ACM.

[9] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. of the international symposium on Computer architecture*, ISCA'09, pages 279–289, New York, NY, USA, 2009. ACM.

[10] Gartner. Gartner Highlights Key Predictions for IT Organizations and Users in 2010 and Beyond. http://www.gartner.com/it/page.jsp?id=1278413.

[11] B. S. Gill and D. S. Modha. WOW: wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proc. of the USENIX conf. on File and stroage technologies*, FAST'05, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[12] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. of the international conf. on Measurement and modeling of computer systems*, SIGMETRICS'97, pages 115–126, New York, NY, USA, 1997. ACM.

[13] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proc. of the USENIX Annual Technical Conf.*, ATC'05, pages 35–35, Berkeley, CA, USA, 2005. USENIX Association.

[15] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proc. of the USENIX Conf. on File and Storage Technologies*, FAST'05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[16] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of the international conf. on Measurement and modeling of computer systems*, SIGMETRICS'02, pages 31–42, New York, NY, USA, 2002. ACM.

[17] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[18] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the International Conf. on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[19] H. Jung, H. Sim, P. Sungmin, S. Kang, and J. Cha. LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. *IEEE Transactions on Consumer Electronics*, 54(3):1215–1223, 2008.

[20] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proc. of the USENIX Technical Conference*, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[21] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of the USENIX conf. on File and stroage technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association.

[22] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proc. of the 6th USENIX conf. on File and Storage Technologies*, 2008.

[23] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. of the conf. on Symposium on Operating System Design & Implementation*, OSDI'00, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[24] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *Proc. of the international conf. on Management of data*, SIGMOD '11, pages 13–24, New York, NY, USA, 2011. ACM.

[25] N. Megiddo and D. S. Modha. ARC: a self-tuning, low overhead replacement cache. In *Proc. of the USENIX conf. on File and stroage technologies*, FAST'13, pages 115–130, Berkeley, CA, USA. USENIX Association.

[26] Nvidia. Tegra 2 and Tegra 3 Super Professors. http://www.nvidia.com/object/tegra-3-processor.html.

[27] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proc. of the international conf. on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[28] Richard Pentin (Summary). Gartner's Mobile Predictions. http://ifonlyblog.wordpress.com/2010/01/14/gartners-mobile-predictions/.