# Block Storage Virtualization with Commodity Secure Digital Cards

Harvey Tuch    Cyprien Laplace    Kenneth C. Barr    Bi Wu *

VMware, Inc.

{htuch, claplace, kbarr}@vmware.com, bi.wu@duke.edu

## Abstract

Smartphones, tablets and other mobile platforms typically accommodate bulk data storage with low-cost, FAT-formatted Secure Digital cards. When one uses a mobile device to run a full-system virtual machine (VM), there can be a mismatch between 1) the VM's I/O mixture, security and reliability requirements and 2) the properties of the storage media available for VM block storage and checkpoint images. To resolve this mismatch, this paper presents a new VM disk image format called the Logging Block Store (LBS). After motivating the need for a new format, LBS is described in detail with experimental results demonstrating its efficacy. As a result of this work, recommendations are made for future optimizations throughout the stack that may simplify and improve the performance of storage virtualization systems on mobile platforms.

***Categories and Subject Descriptors*** D.4.2 [*Operating Systems*]: Storage Management

***General Terms*** Design, Performance, Reliability, Security

***Keywords*** Virtualization, Log Structured File System, Secure Digital card (SD card)

## 1. Introduction

We are presently at a crossover point, at which the volume of smartphones sold each year has surpassed PC sales [31]. Mobile devices will, within a brief period of time, become the dominant end user computing platform. As a result, mobile devices are fast becoming a first-class concern amongst enterprises who face the task of managing a heterogeneous fleet of phones and tablets.

VMware's Mobile Virtualization Platform (MVP) provides an enterprise mobile device management solution, multiplexing two phone personas, a work and a home phone, on a single device via system virtualization. A Bring Your Own Device (BYOD) model is facilitated, in which an employee is given the freedom to select his own device and provision it with a virtual machine (VM) containing the work environment. The hypervisor and on-device management components enable the VM to be managed remotely by the enterprise, while the home environment remains under the complete control of the employee.

The MVP hypervisor is based on a hosted architecture, similar to that of VMware Workstation [28] and Fusion, in which the host provides the home environment and the guest provides the enterprise environment. The virtual machine monitor (VMM) is provisioned from an app store prior to the installation and launch of the enterprise VM. Further motivation for and details of this system structure are provided by [3].

This hypervisor deployment scenario introduces several constraints on storage virtualization:

- *Hardware diversity*. Since devices are selected by employees, MVP must support a wide array of phones, with different storage devices and filesystems. We focus on the variety of Android devices in this paper, but many of the details at the hardware and hypervisor level are common across mobile platforms.
- *Non-perturbation*. Introducing the hypervisor and VM should not cause the user to lose any existing data or require the technical sophistication involved in reformatting or repartitioning storage media.
- *Reliability*. Not all storage is journaled or logged, yet power on the phone may be lost at any time due to battery depletion. VM corruption should not be possible in the event of power loss or a host crash.
- *Security*. The backing store and checkpoint files for the work VM, residing on host filesystems, must be protected from malicious host applications.

Android devices typically offer two types of storage to applications:

- *Internal*. Internal NAND flash memory or embedded MultiMediaCard/Secure Digital (eMMC/eSD) chips are constrained in size due to cost and power consumption. Today, capacities typically range between 256 MB and 16 GB. The system kernel, middleware and libraries reside on internal storage as well as application code and some application data.
- *External*. Small form factor Secure Digital (microSD) cards are standard and provide removable mass storage (up to 32 GB) for application data. Secure Digital Extended Capacity (SDXC) cards will support up to 2 TB capacities in the future. SD card storage benefits from the economies of semiconductor scaling and supply after a smartphone has been shipped and purchased.

The storage footprint for a typical guest, 3 GB or higher, can easily exceed the available space on internal storage for some devices. In order to achieve broad support, the VM backing storage and checkpoint files need to be placed on external microSD cards.

* Bi Wu is a graduate student at Duke University. This research was performed while he was an intern at VMware.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*VEE'12,* March 3–4, 2012, London, England, UK.

Copyright © 2012 ACM 978-1-4503-1175-5/12/03. . . $10.00

SD cards are optimized for cost, compatibility and the I/O mixture expected from the transfer of large sequential files such as MP3s, photos and videos. As a result, they are formatted with the FAT filesystem and have simple flash translation layer (FTL) controllers (utilizing a minimum of costly SRAM) that perform extremely poorly with small random writes [1, 6]. The I/O mixture from the guest is far less sequential than that of the media workloads that SD cards are intended for. In addition, FAT does not support Unix permissions and does not provide robustness guarantees in the event of a host upset. Users cannot be expected to reformat the SD card due to the non-perturbation requirement. These challenges motivate a new VM backing store and checkpoint storage system capable of meeting the constraints outlined above while performing the bulk of storage on FAT-formatted microSD cards.

Our contributions in this paper are as follows:

- An empirical characterization of the unique storage characteristics of SD cards and Android VM workloads.
- A storage architecture and block storage format, which we refer to as the *logging block store* (LBS), capable of providing the desired impedance matching between our enterprise VMs and low cost consumer-grade SD card storage.
- Experimental evaluation of LBS and a performance characterization.
- Potential optimizations at other levels of the I/O stack capable of improving VM performance if adopted in mobile platforms.

While the individual techniques we employ in LBS are familiar, to the best of our knowledge this is the first system to combine them to bridge the gap between the high performance/reliability/security requirements of a VM and the characteristics of the low-cost solid state storage on mobile devices.

In the rest of the paper, we first show how device performance characteristics (Section 2) and virtual machine workload characteristics (Section 3) motivate the design of LBS. This is followed by the design and implementation details for LBS in Section 4 and evaluation in Section 5. The paper concludes with suggested optimizations in Section 6, related work in Section 7 and future directions in Section 8.

## 2. SD card performance characteristics

An SD card is composed of NAND devices, providing the raw storage media, organized by an FTL into a logical block structure that is exported across an SD card bus connector. The FTL performs wear leveling, error detection and the remapping of bad blocks. The limiting storage performance characteristics are hence dictated by the FTL, NAND read/write/erase times and page/erase block organization. For cost reasons, the FTLs are optimized for simplicity of implementation and minimization of SRAM, distinguishing SD cards from their richer cousins, solid state disks, which have more significant resources available for the FTL. The random access property of NAND is as a result constrained by the FTL, with the internal data structures utilized by simple FTLs being optimized for sequential write patterns and coarse block operations [6].

SD cards are rated by speed classes, e.g. Class 2, Class 10, indicating the expected minimum sequential I/O bandwidth (MB/s) in the presence of zero fragmentation [26]. Unfortunately, this rating provides no guarantee of random or fragmented I/O performance. We present some illustrative examples of these characteristics below, gathered on a HTC Nexus One smartphone by a synthetic tool, *sdperf*, designed to characterize SD cards. *sdperf* opens a file or raw block device and performs read or write I/O of specific sizes to the target file. Sequential, strided, partitioned and random patterns are supported.

| Manufacturer | Capacity | Class | Alloc. unit | FAT cluster |
|---|---|---|---|---|
| SanDisk™ | 4GB | 4 | 4MB | 32KB |
| SanDisk™(WP7) | 8GB | 4 | 4MB | 32KB |
| Kingston™ | 4GB | 4 | 4MB | 32KB |
| ADATA™ | 8GB | 6 | 4MB | 32KB |
| PNY™ | 16GB | 10 | 4MB | 32KB |

**Table 1.** SD card details.

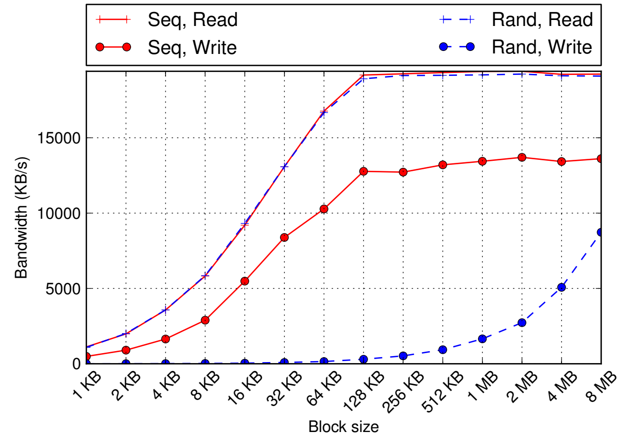



**Figure 1.** 8 GB ADATA Class 6 SD card I/O bandwidth as a function of block size and I/O ordering.

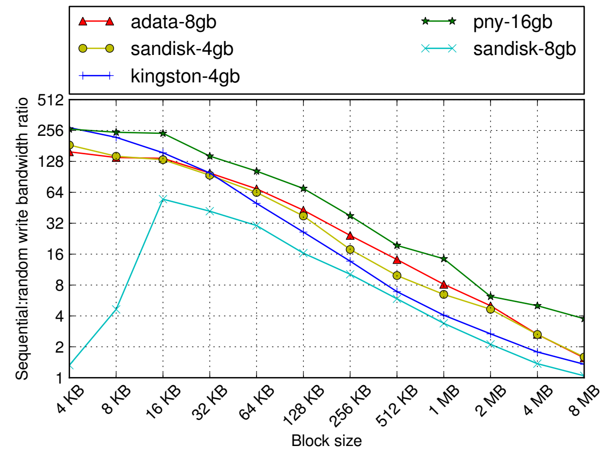



**Figure 2.** Sequential:random write bandwidth ratio as a function of block size.

Below we describe the results of various I/O read/write patterns within a preallocated 128 MB file, intended to be representative of a VM disk image file. The page cache layer in the Linux kernel was bypassed with `O_DIRECT` to avoid interference. Five SD cards from different manufacturers and with different speed class ratings were analyzed; card specific details are provided in Table 1. The 8 GB SanDisk card packaging was labeled as being Windows Phone 7 compliant, indicating potential improved support for random read/write operations [29].

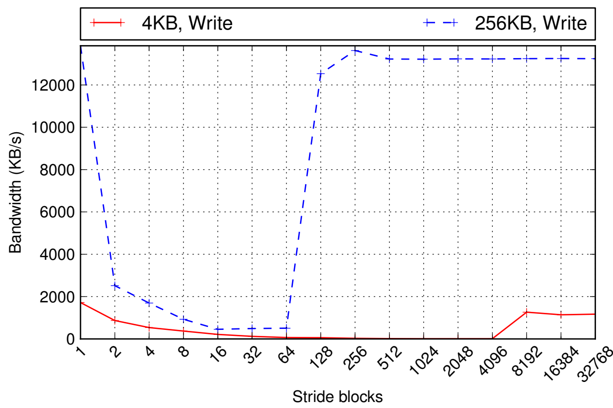



**Figure 3.** 8 GB ADATA Class 6 SD card write bandwidth as a function of inter-write stride distance and block size.

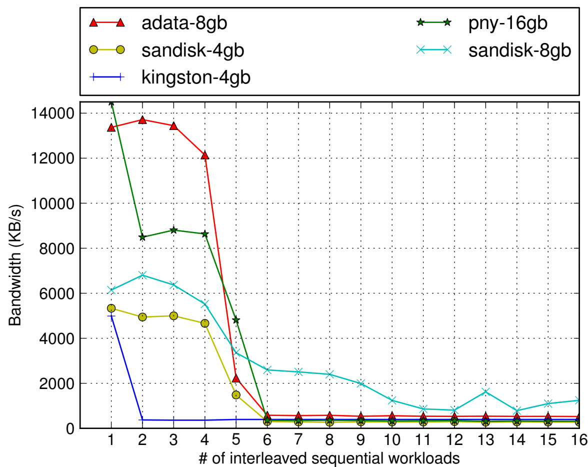



**Figure 4.** Write bandwidth as a function of the number of interleaved sequential workloads, separated by 2 AU, at 256 KB block size.

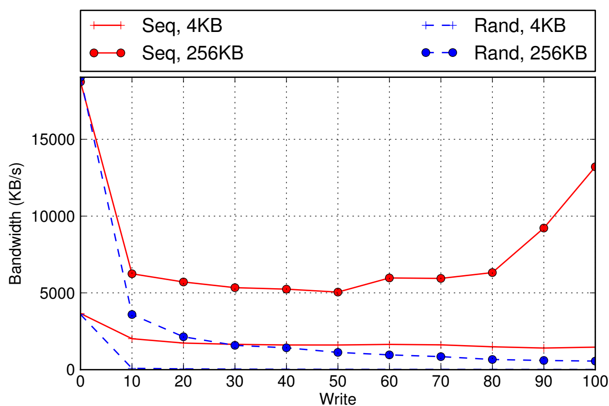



**Figure 5.** 8 GB ADATA Class 6 SD card I/O bandwidth as a function of write percentage in I/O mixture and I/O ordering.

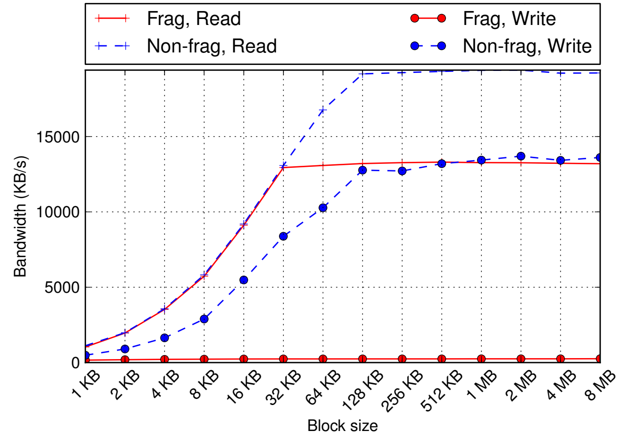



**Figure 6.** 8 GB ADATA Class 6 SD card sequential I/O bandwidth as a function of block size and fragmentation.

Figure 1 provides the observed I/O bandwidth as a function of the block size and access pattern on the 8 GB Class 6 ADATA SD card. There is little difference in these examples between sequential and random read performance, but a marked distinction on writes, in particular at small block sizes. Figure 2 provides the sequential-to-random write performance ratio for all five cards. A similar random write penalty can be observed across the tested cards, with the exception of the 8 GB SanDisk. The card exhibited comparable sequential and random write performance at 4 KB block sizes but behaved similarly otherwise to its peers past 16 KB. We use the 8 GB ADATA card as a running example in the rest of the paper since its key characteristics are similar to other cards we have examined.

The penalty for a non-sequential write is not uniform, it depends to some extent on the location and distance between the two writes, as well as the history of previous writes. Figure 3 provides the write bandwidth at the 4 KB and 256 KB block sizes when a stride takes place between writes. At stride of 1 block, we have the sequential case and performance drops until writes are an *allocation unit* (AU) apart. The AU is a logical unit provided by the SD card at which erase operations are preferred and speed class calculations performed. For a given card it has a fixed size, dictated by the NAND erase block size and card internal organization (4 MB for the card in the figure). Writes at a smaller granularity can involve a read-modify-write operation. When the stride becomes sufficiently large, we might expect to see a change in performance when once again only a single AU is in use, as strides wrap around at the file size, 128 MB. A performance improvement occurs earlier however, at 32 MB, likely due to an FTL implementation that supports efficient interleaving of writes to multiple AUs as long as sequentiality is maintained within each stream [19]. This effect is visible in Figure 4, where we simulate interleaved sequential writers, with the writes occurring at a distance of 2 AU. Several of the cards show good performance with up to four sequential writers. The PNY card is best with a single writer, but supports 2-4 writers with mid-range performance. The Kingston card supports only a single writer. While it may be tempting to exploit these patterns, they are card and distance specific: with only 5 cards we were able to identify 3 behaviors. We assume the non-sequential write penalty to be high for the rest of this paper, since we are aiming to provide a portable solution where the SD card is unknown.

Even a small number of write accesses in an I/O mixture can drive overall performance towards the write performance curve, as indicated in Figure 5, where write accesses were inserted at random

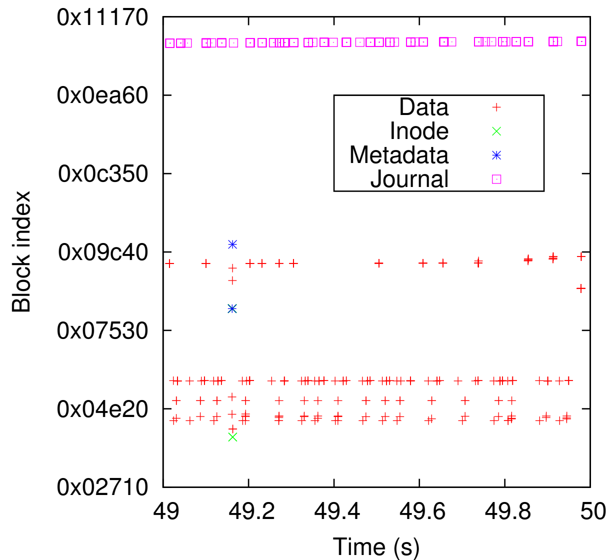



**Figure 7.** 1 second sample of browsing session writes on ext3.

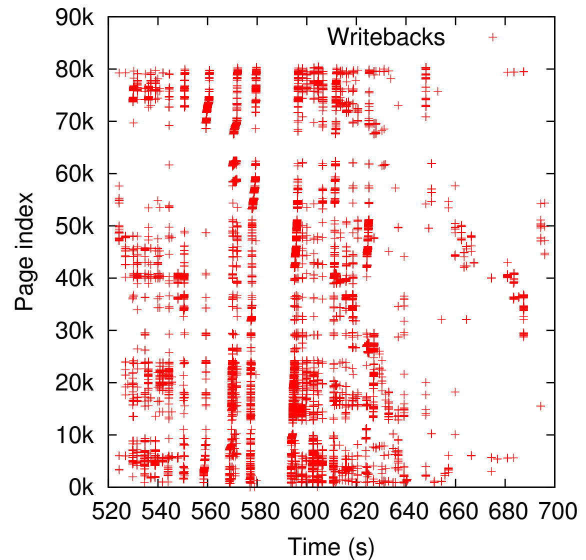



**Figure 8.** 180 second sample of background cold page writebacks of a large space of guest physical memory.

in the I/O mixture at differing percentages. Beyond 10% there is little difference between a mixed and pure write workload.

Fragmentation within an AU will result in a decrease in sequential write performance. In the above experiments, the image file had zero measured fragmentation, with logically contiguous blocks placed on contiguous FAT clusters. Figure 6 presents a comparison of sequential I/O performance in the non-fragmented and worst case fragmentation cases. Worst case fragmentation here is simulated by placing files of 1 MB in size until the card is filled, and then releasing a single FAT cluster from each file prior to image file allocation. This results in the image file being spread across the maximum number of AUs. The curves representing performance of sequential I/O on a non-fragmented filesystem are the same as those in Figure 1. When the file system is severely fragmented, data is only contiguous within a 32 KB FAT cluster, so read performance is limited by the speed of 32 KB reads. Sequential write performance devolves to that of 32 KB random writes.

Beyond the above observations, there are implementation details that apply to specific cards. For example, the initial blocks backing the file allocation table are optimized for the smaller, non-sequential writes that occur in the region [16]. These card specific details are also not relied upon in the LBS design presented in Section 4.

## 3. Virtual machine I/O mixture

The I/O mixture from the perspective of the host is non-sequential for three reasons in our system: guest filesystem design, opportunistic checkpointing and application behavior.

The guest uses ext3 and FAT filesystems over paravirtualized block storage devices. With the simpler FAT filesystem, non-sequentiality can arise from application access patterns and fragmentation. The journaling ext3 filesystem introduces additional non-sequential writes, jumping between ordinary data and the journal. Figure 7 shows a sample of a block write trace to an ext3 partition during an Android 2.2 web browsing session. Non-sequential writes can be observed as the application accesses four different regions of data (the four "stripes" below 0x09c40). Access to metadata, inodes and the journal interrupt data access with additional non-sequentiality.

MVP also supports virtual machine checkpointing. It is possible to save all virtual machine state on the host storage and to restore it later. A virtual machine's state is composed of the following parts:

- The *virtual platform*, including CPU registers and virtual device state (approximately 200 KB).
- The *storage*, maintained in persistent images by the paravirtualized block storage devices.
- The *memory*, which may require $\geq$ 512 MB of data to be written on checkpoint.

Of these components, saving the VM's large memory dominates the space and time required to save a checkpoint. To shorten the duration of checkpoint creation, unused memory is written to persistent storage proactively in the background. An adapted Clock-Pro [14] working set estimation algorithm is used to select blocks to write. Cold blocks are preferred with the assumption that they are the least likely to change prior to explicit checkpoint creation time. Unfortunately, this working set driven selection often leads to non-sequential ordering of page writebacks, as shown in the sample given in Figure 8.

Application behavior in the presence of the guest buffer cache introduces another source of non-sequential I/O. We examined several mobile workloads by generating traces of their I/O (details of the tracing procedure are given in Section 5).

1. **Android Boot**. The initial boot of an Android OS. The trace ends when Android issues its BOOT_COMPLETED intent. A considerable number of writes occur during Android's optimization of application bytecode.
2. **Contacts Database**. Import 2000 contacts into the Android Contacts application. Search for and delete 40 contacts.
3. **Mail Client**. Use the Android Mail 2.2.1 client to access an IMAP mailbox. The mailbox is 24 MB and contains 356 messages and 3 folders with lengths and attachments generated by the SPECmail2009 benchmark [27] initialization script to reflect size distributions of a large corporation.
4. **Slideshow**. Browse through 52 NASA images [18] using the Astro file browser [17]. Astro creates thumbnails and scales

| Name | Writable | Filesystem | Description |
|---|---|---|---|
| /system | | squashfs | Android binaries |
| /data | x | ext3 | user-installed programs and data |
| /cache | x | ext3 | cache space used by Android |
| /sdcard | x | FAT | SD card for multi-media files |
| /flex | | squashfs | enterprise customizations |

**Table 2.** Disk partitions in the Android guest under test.

photos from their original size to fit on the device's 800x480 pixel screen.

5. **Web Browsing**. A one second sample of web browsing activity using the Android 2.2 browser.

The Android guest that produced the traces has 5 partitions as shown in Table 2. Note that squashfs partitions are not writable and are managed by flat files in our implementation rather than LBS.

The traces stress the partitions in different ways. Table 3 describe the characteristics of each trace. Unaccessed partitions are elided from the tables. While some partitions are lightly used or not accessed at all during the workload, we see that workloads such as the Contacts Database and Email Client are dominated by small writes to the data partition.

Recall from Section 2 that random workloads with a write percentage as little as 10% can perform as slowly as if 100% of their I/Os were writes. In the case of the Contacts Database and Email Client workloads, the high percentage of writes to /data — with 12%-17% of these writes being non-sequential — is cause for concern and motivates a virtualization layer that will transform the workload into large sequential writes. In most of our traces, the number of barrier commands per I/O is low, but the email trace stands out with 33% of its I/O commands being barriers. To achieve consistency, barriers force the contents of any buffers to be flushed to the disk which inhibits the ability to batch commands into large, fast I/O operations.

The table also shows that except in the case of Android Boot, write activity consists mostly of small (no larger than 4 KB), less efficient I/O. Read activity is more likely to contain larger I/O commands which should perform well if not interspersed with writes.

# 4. Logging block store (LBS)

We observed in Section 2 and Section 3 a mismatch with the underlying SD card characteristics and the guest Android VM and checkpoint I/O mixture. There are also two additional challenges facing a storage virtualization and checkpointing solution employing commodity SD cards on a hosted hypervisor:

- *Security*. The VM image and checkpoint files stored on the SD card must be protected from malicious host applications. On Android, any application which has been granted permission to access the SD card may access any file. This permissiveness stems from the lack of access control in the FAT filesystem.
- *Reliability*. A VM on a mobile device executes in a hostile physical environment in which power may be lost due to battery exhaustion, a phone being dropped or the host kernel crashing. The FAT filesystem does not provide any resiliency against data loss in such events.

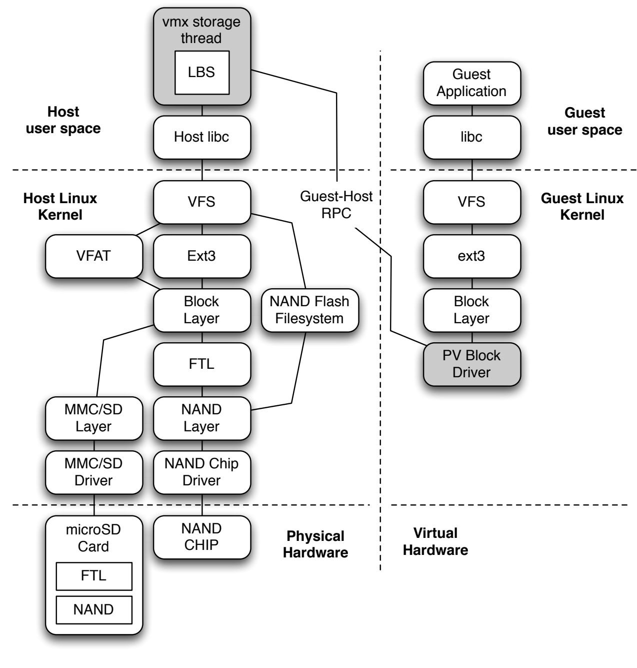



**Figure 9.** MVP storage architecture.

In this section we present a VM image format that addresses the performance, security and reliability problems described above. We call this image format the *logging block store* (LBS).

As an illustration of the role LBS plays in the MVP hypervisor, consider the various layers in both the guest and host storage stacks shown in Figure 9. The guest kernel contains a paravirtualized block storage driver, providing the *front-end* of the virtual device. The driver is responsible for communicating requests in the kernel's block request queue with a thread executing in host user space, residing in a process known as the vmx. The storage virtualization *back-end* is implemented with standard POSIX file operations on host filesystems residing on both internal and external storage. LBS is the component responsible for the back-end virtualization in the vmx storage thread.

## 4.1 LBS format

As suggested by its name, LBS is a log structured file format intended to represent VM disk images at the block granularity. The log structure allows us to bridge the gap between the non-sequential I/O mixture observed in Section 3 and the SD card performance characteristics described in Section 2.

An LBS image is split between a data file, with suffix .lbsd, and a meta-data file, suffixed .lbsm. We locate the data file on the FAT filesystem backed by the SD card due to its significant size and place the meta-data file on internal storage, since it is a fraction of the size of its data counterpart and is able to enjoy the security and robustness benefits of the location. Figure 10 illustrates the layout of the respective LBS backing files.

The data file consists of a number of fixed sized *blocks* (1 KB) organized into fixed size *clusters* (256 KB). We refer to the block index in the data file as the *physical block number* (PBN). The *logical block number* (LBN) refers to the index of a 1 KB block inside the VM disk image being represented via LBS. A page mapping table is maintained by LBS in the vmx process to translate between LBN and PBN. Clusters exist for the purpose of garbage collection (GC) and ensuring that contiguous sequential runs of

| | read | | | | | | write | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| partition | size (KB) | count | I/O Sizes (KB) 1 | $\leq 4$ | $> 4$ | skip % | size (KB) | count | I/O Sizes (KB) 1 | $\leq 4$ | $> 4$ | skip % | write % | barrier % |
| Android Boot | | | | | | | | | | | | | | |
| `/system` | 18562 | 1275 | 21 | 39 | 1215 | 0 | 0 | 0 | 0 | 0 | 0 | NA | 0 | 0 |
| `/data` | 28 | 24 | 24 | 0 | 0 | 41 | 16468 | 1204 | 120 | 43 | 1041 | 1 | 98 | 2 |
| `/cache` | 11 | 7 | 7 | 0 | 0 | 27 | 18 | 9 | 7 | 1 | 1 | 19 | 56 | 6 |
| `/flex` | 29 | 6 | 3 | 1 | 2 | 7 | 0 | 0 | 0 | 0 | 0 | NA | 0 | 0 |
| Contacts Database | | | | | | | | | | | | | | |
| `/system` | 763 | 64 | 8 | 6 | 50 | 1 | 0 | 0 | 0 | 0 | 0 | NA | 0 | 0 |
| `/data` | 855 | 83 | 32 | 21 | 30 | 4 | 55281 | 21048 | 15414 | 3348 | 2286 | 17 | 100 | 4 |
| Email Client | | | | | | | | | | | | | | |
| `/system` | 6218 | 511 | 80 | 26 | 405 | 1 | 0 | 0 | 0 | 0 | 0 | NA | 0 | 0 |
| `/data` | 2473 | 191 | 53 | 51 | 87 | 3 | 52100 | 16668 | 10520 | 2595 | 3553 | 12 | 99 | 33 |
| `/sdcard` | 6 | 3 | 2 | 1 | 0 | 17 | 112 | 97 | 90 | 7 | 0 | 43 | 97 | 0 |
| Slideshow | | | | | | | | | | | | | | |
| `/system` | 5664 | 449 | 55 | 23 | 371 | 1 | 0 | 0 | 0 | 0 | 0 | NA | 0 | 0 |
| `/data` | 500 | 82 | 27 | 23 | 32 | 8 | 6452 | 659 | 173 | 73 | 413 | 2 | 89 | 10 |
| `/sdcard` | 38 | 18 | 14 | 2 | 2 | 22 | 13701 | 1883 | 1407 | 197 | 279 | 6 | 99 | 0 |
| Browsing | | | | | | | | | | | | | | |
| `/data` | 43895 | 1662 | 313 | 228 | 1121 | 1 | 12609 | 3512 | 2149 | 542 | 821 | 10 | 68 | 18 |

**Table 3.** Characteristics of I/O traces (read/write breakdown). The size of the trace is expressed in KB and as a count of total operations. A rough classification of I/O sizes is provided in which each column is exclusive of adjacent columns. "Skip" is a measure of sequentiality: the number of block accesses that are not adjacent to a previous access. A skip percentage of 100 represents a completely non-sequential workload; a skip percentage of 0 is completely sequential. Write and barrier percentages are relative to the total number of I/O operations per-partition.

blocks exist to speedup writes. Writes always append to the end of the current *active* cluster, with GC maintaining a pool of free clusters to provide when the active cluster is full. Read operations first map from LBN to the current PBN for the logical block, using the page mapping table, prior to reading from the data file.

To minimize write latency and increase the size of sequential writes, LBS employs a write buffer. The write buffer is flushed on guest barrier operations and when it is full. Significant performance improvements are possible by write buffering when operating at the sequential write performance plateau, which occurs at 128 KB in Figure 1 — the write buffer size we employ is 256 KB.

Both the guest and host kernels maintain a page cache. When a file is read by a guest application, the data of that file are cached by the guest kernel, and the same data would ordinarily be cached by the host kernel. To avoid this double caching, the `vmx` storage thread uses the `O_DIRECT` flag when opening the `.lbsd` file, instructing the host kernel to avoid using the page cache for this file.

The meta-data file is an append-only log of meta-data and barrier entries. On each guest write, an entry is appended to the meta-data file, reflecting changes to the LBN→PBN mapping. When the guest kernel issues barriers, a barrier entry is appended, as described in Section 4.2. Meta-data entries are run-length encoded and zero blocks are optimized in both data and meta-data files. At the end of each meta-data entry is a series of fields providing block checksums and timestamps for non-zero blocks; Section 4.2 and Section 4.3 provide further details.

The meta-data file is not garbage collected online. Instead, when the VM image is opened, or when its size reaches a fixed limit, we perform an offline, atomic sweep to eliminate stale entries.

The in-memory LBN→PBN structures require 12 MB per 1 GB of logical block space. While we do not face the severe memory constraints that exist in other systems, for example FTLs on SD cards, it is wasteful and unnecessary to maintain the entire set of structures in memory at all time. Instead, they are placed in a `mmap`ed file located in internal storage, allowing the host kernel's existing page cache writeback and eviction mechanisms to manage the portions of the meta-data mapping structures that are maintained in memory and those on flash, based on the working set behavior of LBS. This has similarities to DFTL [11] page map caching and by placing the backing file on internal storage, we separate the competing data and meta-data I/O streams, conferring a performance advantage analogous to the HAT FTL scheme [12].

LBS data files are fully allocated on the FAT at creation, preventing further fragmentation once initialized. This is intended to alleviate the fragmentation penalty discussed in Section 2.

### 4.2 Reliability

There are several modes by which VM storage could possibly fail and which we wish to mitigate against:

- When a host crashes, due to battery depletion, software bugs or other causes:

  *[CRASH-META]* Corruption of the filesystem meta-data on which the `.lbsd/.lbsm` files reside during write operations.

  *[CRASH-DATA]* Corruption of the filesystem data on which the `.lbsd/.lbsm` files reside during write operations.

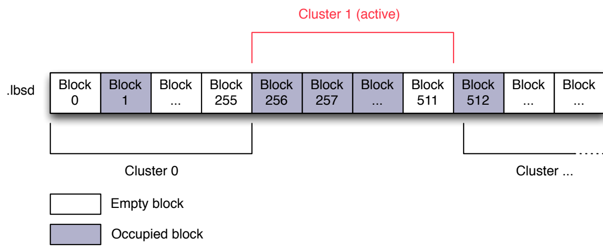
  *[BUFFER-LOSS]* Loss of write buffers in either main memory or on the flash controller. Buffers may be partially written back (in any order) at crash time.

- When a NAND device fails:

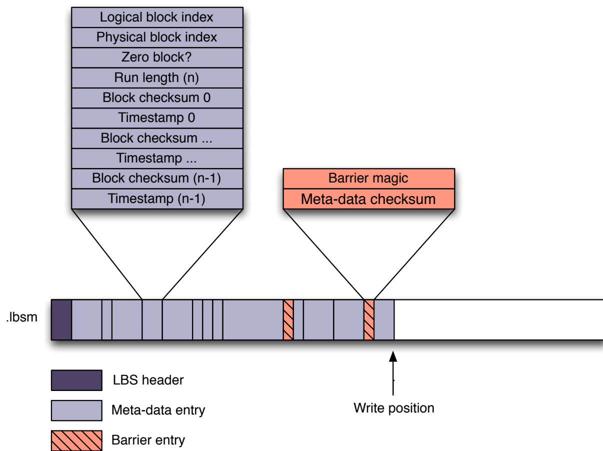
  *[MEDIA-FAIL]* Media failure and/or corruption in the internal or external NAND devices at the page or erase block granularity.

As a hosted hypervisor we rely on the guarantees provided by the underlying filesystems for *[CRASH-META]*. Internal storage is typically formatted with some log structured filesystem, such as YAFFS or JFFS2, which provides robustness for the `.lbsm` meta-data files. We accept as a fundamental limitation that possible FAT allocation table corruption presents an unmitigated point-of-failure for the data file.

*[CRASH-DATA]* is detectable through the use of checksums on both the LBS data and meta-data. Each data block has a 32 bit Fletcher checksum [9] or SHA-256 checksum computed when




(a) Data (`.lbsd`)




(b) Meta-data (`.lbsm`)

**Figure 10.** LBS file formats.

written and the checksum is stored in the corresponding meta-data entry. Barrier entries in the meta-data are written whenever the journaled guest `ext3` filesystem issues a barrier, and contain a Fletcher-32 checksum of the meta-data entries since the last barrier entry. In this way, if a missing terminating barrier or corruption of the LBS meta-data is detected, it is possible to rollback to the last barrier while maintaining the expected guest barrier semantics during recovery.

The LBS implementation ensures that internal write buffers are flushed and that `fsync`/`fdatasync` calls are made on first the `.lbsd` and then the `.lbsm` on a guest issued barrier operation to manage *[BUFFER-LOSS]*.

When meta-data GC is performed, there is a possibility of failure occurring due to a host crash. To reduce the likelihood of this occurring, we write to a secondary meta-data file `.lbsm2`, and only when the GC is complete is an atomic `rename` system call issued.

While there already exist error detection mechanisms within the FTLs for both internal and external storage, the checksums provide an additional mechanism to detect *[MEDIA-FAIL]*. This is particularly useful when a malicious attacker may freely modify blocks in the `.lbsd`, producing symptoms similar to media failure from the point-of-view of LBS while being undetectable from the point-of-view of the FTL, which regards the modification as a legitimate write operation. This attack is further detailed in the next section.

It is a significant challenge to test and validate the failure paths of a storage virtualization stack, since the failure modes we expect are rare and unlikely to appear under normal conditions. The resilience of the LBS design and implementation to the failure modes described above has been validated with in-situ model checking techniques, similar to Yang et al [33] but applied at the block storage device level instead of filesystems. The LBS implementation was recompiled and linked against a `ptrace`-based test harness instead of the `vmx`. The test harness provided a means to simulate an adversarial environment, for example by injecting system call failures or partially corrupting uncommitted data during a simulated crash. A number of serious implementation bugs were corrected with the results provided by this technique and confidence in correctness gained at both design and implementation level.

### 4.3 Security

We have derived a threat model capturing the concerns of MVP's users based on discussions with enterprises, carriers and OEMs. There are two classes of attacks that are widely seen as being the most relevant to a virtual phone:

- *Physical attacks*. Here we are primarily concerned with the confidentiality of data if a phone or SD card is lost or stolen. This is also known as confidentiality of *data-at-rest*.
- *Malicious host applications*. Malware may attempt to compromise the confidentiality or integrity of the VM image by exploiting weaknesses in the host OS security model, either while the VM is running or when it is suspended.

The LBS data file is encrypted at block granularity with an XTS-AES cipher. If an SD card or phone is stolen, there is no plain text on the storage media available to an attacker. The key is located on the protected internal storage and has the same confidentiality guarantees that exist for the Android application keystore.

Malicious host applications are a significant concern on Android if relying on the SD card for storage. Since the SD card is FAT formatted, the standard Unix permission model that Android uses to sandbox applications is limited by the lack of ownership and group information support in the filesystem. The XTS-AES cipher provides some reprieve from this threat, since malicious host applications may not learn the contents of the LBS data file, but does not directly protect against *replay* and *randomization* attacks. Replay attacks occur when a legitimate encrypted block from the past is reintroduced in the LBS data file by a malicious application, causing the guest behavior to be influenced by the contents of the injected block. The attacker does not need to know the plain text contents of the block, simply the associated guest behavior that was previously observed. Randomization attacks occur when a block is intentionally corrupted, e.g. fuzzed, leading to guest applications or the kernel behaving outside the usual control envelope where they depend on the contents of a block.

To combat replay and randomization attacks, we maintain a timestamp and checksum respectively in the LBS meta-data. The timestamp refers to a logical clock maintained by LBS, incremented after each write. The checksum selection is configurable and may be either a Fletcher-32 or truncated SHA-256 hash, allowing for a tradeoff to be made between the cheap to compute but not cryptographically strong Fletcher checksum and the more expensive SHA-256, depending on policy. In both cases, the checksum is applied to the plain text block contents prior to XTS-AES and the resulting checksum is hidden from the malicious host application, being located in the LBS meta-data, protected by Unix permissions on the more full featured YAFFS or JFFS2 used for internal storage. Hence, when attempting a randomization attack, the attacker is limited by a lack of knowledge of the checksummed block contents.

### 4.4 Garbage collection

Cluster garbage collection (GC) takes place in a thread separate from the main storage virtualization backend thread. GC starts when

the number of free clusters drops below a low watermark and stops when the number of free clusters climbs back up above a high watermark. The watermarks hence provide GC control hysteresis.

The GC thread selects clusters to free based on heuristics that take into account the "emptiness" of a cluster, i.e. the number of unoccupied blocks. Occupied blocks from selected clusters are arranged in a queue and their contents are loaded in the background into memory. When enough data exists to complete the current active cluster, the preloaded blocks are extracted from the queue and written to the active cluster. The LBS data file is overprovisioned with space, e.g. the number of physical blocks allocated might be 112% of the logical blocks, to ensure that even a full guest image maintains adequate space for performant GC.

It is desirable to minimize the number of non-sequential jumps in the write position, even when write buffering and achieving sequential access within a cluster. We have found that the simple occupancy heuristic mentioned above results in the freeing of non-contiguous physical clusters over time, resulting in free clusters scattered around the data file. One solution to this problem is to combine additional heuristics using a scoring for each cluster, and garbage collect contiguous occupied clusters that have the highest score. The current scoring function has four weighted components:

- *[EMPTINESS]* The number of unoccupied blocks divided by blocks-per-cluster. A cluster with more free blocks is valued higher.
- *[LEFT EMPTY]* An award to clusters whose left sibling is empty because we want as many contiguous free clusters as possible. We assume that most I/O is ascending, so the right sibling is likely to be considered in the future; without this component, we might not otherwise examine the left sibling.
- *[OUTLIER CORRECTION]* An award provided to those clusters which have an abnormally high amount of occupied blocks compared to the clusters around them. Outlier clusters can be favored by treating them as if they are almost as empty as their surrounding clusters. Reclaiming an unusually full cluster in the midst of many empty clusters can extend the length of a contiguous series of clusters.
- *[WRITE POSITION]* A boost in score is given to the cluster next to the cluster currently being written. This promotes contiguous writes.

The components are equally weighted for the experiments in this paper, but an improved scoring function may exist in which component weight is dependent on the amount of free space for GC in the file. The more free space we have, the more important *[WRITE POSITION]* and *[LEFT EMPTY]* become. When there is little free space available for GC (typically below 25%), *[EMPTINESS]* starts to dominate the other factors.

The storage virtualization stack supports the propagation of guest TRIM commands. These are leveraged within LBS to assist GC by providing a more precise view of block status. More details can be found in Section 6.2.

## 5. Evaluation

This section describes the experiments used to quantify LBS performance. The benefits of LBS are shown with both synthetic and trace-based tests. The costs of write amplification, integrity checking and encryption are measured and discussed.

### 5.1 Experimental setup

Both synthetic and realistic, user-centric tests were used to characterize the performance of LBS. All experiments were conducted on an HTC Nexus One phone. The Nexus One contains a 1 GHz Qualcomm Snapdragon chipset and 512 MB of DRAM [10]. The phone ran an aftermarket version of Android 2.3 (Gingerbread) called CyanogenMod 7.1.

| | |
|---|---|
| device ID | the partition accessed by this I/O |
| R/W | flag indicating whether this I/O is a read or a write |
| fragment | position in scatter-gather list |
| offset | location on disk |
| length | number of bytes accessed |
| barrier | flag indicating that this record represents a barrier request |

**Table 4.** Trace record

The synthetic test is *sdperf* from Section 2. For realistic and repeatable results of the storage behavior of mobile applications, we wrote a trace-driven execution tool called *blksim*. We first gather an I/O trace using the guest block device driver during the use of an application. A trace record contains the fields shown in Table 4.

To replay the trace, *blksim* opens a raw block device in the guest with the `O_DIRECT` flag. Using a raw device removes filesystem effects. `O_DIRECT` removes the effect of the guest OS buffer cache and ensures that the virtualized device is presented with the same stream of I/O that was captured in the trace. Prior to enabling performance measurement, the trace is read into memory to create one command stream for each disk partition represented in the trace; this prevents the I/O required to read the trace from impacting the measured portion of the experiment.

Each partition's command stream is then applied to the device, one at a time. Reads and writes are emulated with `pread` and `pwrite`. A custom `ioctl` is used to provide end-to-end barrier semantics in the guest by blocking the guest *blksim* thread until the host has completed the `fsync` to the physical device.

We use the workloads described in Section 3 with one change: the Android Boot and Slideshow workloads were expanded by repeating each five times to produce a longer-running trace to magnify any performance differences and reduce variance.

### 5.2 Benefit of LBS

The results of the *sdperf* test with the 8 GB ADATA Class 6 SD card are shown in Figure 11. We compare LBS to a simpler "flat" VM image format which does not buffer I/O and uses a straight-through mapping of virtual disk blocks to physical disk blocks in the VM image. The graph presents the ratio between bandwidth achieved with both file formats. Garbage collection was not triggered during the experiments.

Like Figure 1, the upper graph shows the similarity of sequential and random reads. The performance difference between flat and LBS is due to encryption, integrity checking and indirection. It suggests that flat VM image formats are a better choice for read-only partitions, especially with larger block sizes.

When writes are present, performance will be closer to that shown in the lower graph. Here we see regions in which LBS has advantages for SD cards. For small writes ($\leq$ 32 KB), the LBS write buffer allows it to exceed the bandwidth achieved with a simple flat file — even for sequential writes — by making fewer, faster large I/Os to the SD card. For all tested block sizes, LBS outperforms the flat file for random writes. This is due to both the write buffering present in LBS and its ability, due to its log-based structure, to transform a stream of random writes into a sequential stream more suited for the SD card.

The benefit of LBS over a flat file can also be seen in the *blksim* results shown in Figure 12. Each workload was repeated three times for each set of filesystem options. The average of the three runs is normalized to the runtime of a flat virtual disk. We show a cluster

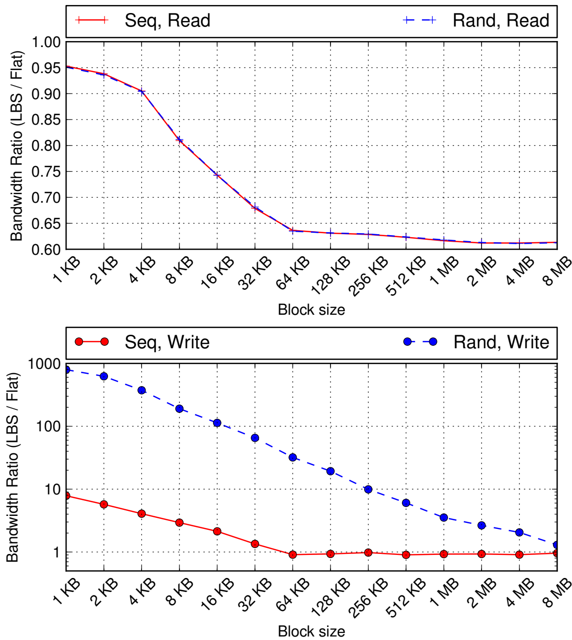



**Figure 11.** 8 GB ADATA Class 6 SD card I/O bandwidth: Ratio between LBS and Flat file format.

of bars for each partition that had non-negligible traffic. 160 MB was set aside to provide LBS with sufficient free blocks such that garbage collection does not occur; the impact of garbage collection is shown in Section 5.3.

Recall that the Contact Database workload is almost completely small writes with little sequentiality and a relatively small number of barriers. Thus, it exhibits a dramatic 17× improvement over the flat virtual disk due to LBS transforming the I/O into large, sequential writes. The other write-heavy workloads benefit as well. The email workload is nearly 5× faster with LBS than it is with a flat file; the reason it is not better may be due to it having the highest percentage of barriers within its I/O trace. Each barrier forces a flush of the LBS write buffer, and smaller writes are less efficient.

### 5.3 Write amplification

As described in Section 4.4, a garbage collection thread runs when space is low to provide free clusters for future writes. While this causes additional write operations to occur, their cost can be mitigated for two reasons. First, if no other write activity is required, the additional writes can occur in the background. Second, the writes will be to contiguous blocks which are relatively inexpensive on SD cards.

In Table 5, we show this *write amplification* due to LBS for the two workloads that triggered garbage collection. The table aggregates the writes from all of the virtual disk partitions. In this context, we ignore the effect of hardware-level writes that may occur depending on the implementation of the SD card's flash translation layer. Column 1 lists the number of 1 KB writes generated by the guest block driver. Note that these writes have been filtered by the guest OS buffer cache at the time of trace capture.

Column 2 shows the number of writes that actually occur, measured in units of both 1 KB blocks and 256 KB LBS clusters. These counts were gathered by setting aside 100% of the virtual disk's space as a buffer for the garbage collector so that the workload never triggers the GC threshold, and no GC writes occur. A small

| trace | requested writes | LBS writes without GC | LBS writes with GC |
|---|---|---|---|
| Contacts | 219124 blocks | 219124 blocks 855 clusters | 223593 blocks, 873 clusters (802 contiguous) |
| Email | 216057 blocks | 215972 blocks 843 clusters | 220160 blocks 860 clusters (813 contiguous) |

**Table 5.** Software-level write amplification due to garbage collection. 12% additional storage used for garbage collection.

reduction in writes occurs in the Email Client workload due to overwriting stale data in the LBS write buffer.

Column 3 reflects both the reduction in writes due to buffering and the write amplification due to garbage collection (the LBS data file had 12% extra space for GC). While the number of blocks that must be written has increased, recall that blocks belong to larger clusters, many of which are contiguous. The large size and high contiguity make LBS efficient.

### 5.4 Cost of encryption and integrity checking

Figure 12 also shows how encryption and integrity checking affect performance. The cost of encryption varies for each trace from as little as 2% for the Email Client workload, with its high barrier percentage, to as much as 35% for Android Boot. The relative cost of encryption increases as block size increases. For larger blocks, fixed I/O overhead represents a smaller fraction of overall time, and encryption overhead (which is proportional to block size) has a greater relative impact. Investigating the use of hardware encryption offload engines is left as future work.

As mentioned in Section 4.2, data blocks can be protected by the fast Fletcher checksum or a slower-to-compute, cryptographically strong SHA-256 checksum. The incremental cost of integrity checking with Fletcher-32 ranges from 1%–6%. Unfortunately, cryptographic hashes such as SHA-256 are relatively expensive to compute on each block. Using a C implementation of Fletcher-32 and the standard Android 2.2 OpenSSL implementation on a Nexus One device (in ARM assembly language), the following throughput figures (with 1KB block sizes) represent the speed at which memory contents can be checksummed. SHA-256 requires nearly 7.7× longer than Fletcher-32 to process data in these tests.

| | |
|---|---|
| SHA-1 | 46.83 MB/s |
| SHA-256 | 31.26 MB/s |
| Fletcher (32 bits) | 240.81 MB/s |

Corporate policies may demand cryptographically strong hash algorithms such as SHA-256. In our tests, *sdperf* results were degraded by approximately 4–8% for block sizes less than 4 KB when SHA-256 was used instead of Fletcher-32. Overhead of the more expensive hash function increases with larger I/O as the cost of computation becomes a larger portion of the total I/O cost. While this is non-negligible, even the worst-case *sdperf* result (17% reduction in bandwidth for 256 KB reads, 13% reduction for writes) is much better than the 7.7× slowdown observed in memory-only tests.

## 6. Optimizing the mobile I/O stack

LBS effectively bridges the gap between the available storage media characteristics and VM I/O requirements. Enhancements in the guest, host and SD card have potential to further improve LBS performance and/or simplify the VM image format. These optimizations may prove beneficial even on non-virtualized systems.

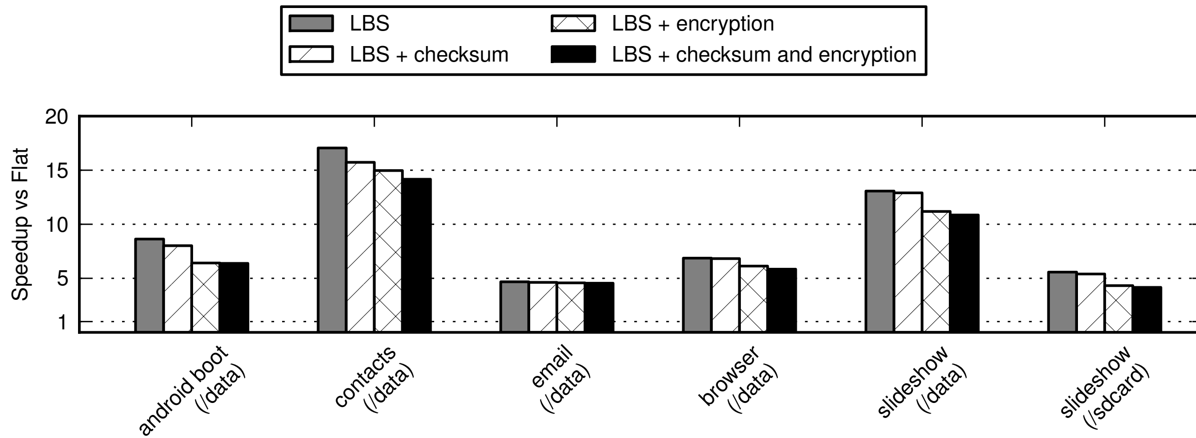



**Figure 12.** Performance of I/O traces without garbage collection.

### 6.1 Guest

The sequential vs. random and block size write asymmetry may also be addressed at the filesystem level in the guest, rather than with the block storage virtualization layer, with a suitable guest log structured filesystem. The YAFFS and JFFS families of flash filesystems employ a log structure and appear natural candidates on Android. Beyond scalability concerns, the chief drawback in a virtualized setting is the increased complexity at the guest-VMM surface, where a device model capable of emulating NAND devices is required, since these filesystems leverage the spare area associated with pages in NAND. MVP currently presents a simple paravirtualized block device to the kernel's block layer, minimizing both guest driver and VMM complexity and overhead. In addition, such filesystems are optimized for direct NAND control and some translation will be required to bridge between the FTL-abstracted NAND on the SD card and the virtual NAND devices. Other Linux log structured filesystems, e.g. LogFS and NILFS, are experimental as of 2.6.35 and are unsuitable for enterprise VMs as a result.

A stable implementation of a Linux log structured filesystem operating on the block layer will open up the possibility of deployment within MVP guests, lessening the need for the log structure of LBS, but not the security and reliability aspects. The guest memory working set behavior and resulting checkpoint subsystem writebacks cannot be modified as simply as the filesystem, since these are properties of the application and kernel and not under our control. As a result, the host filesystem is still presented with a non-sequential I/O stream, and a system such as LBS will continue to be advantageous for VM checkpointing on SD cards — even if a log structured filesystem is in use in the guest.

### 6.2 Host

The fundamental assumption guiding a hosted mobile hypervisor architecture is that the host OS and hardware are the purview of OEMs, silicon and mobile OS vendors. Below we provide several suggestions for these entities that should prove beneficial for VMs or applications that make extensive use of the SD card:

- As mentioned in Section 4, the LBS data files are allocated on the FAT filesystem at initialization time, reducing fragmentation and ensuring availability of space at runtime. This is a slow operation for large images: since FAT does not support sparse files or extents, space must be reserved by allocating each block of the file (e.g., with zeroes). MVP provides a host kernel patch that allocates filesystem structures for an LBS file and omits the block zeroing. OEMs may optionally apply this patch to improve the speed of VM provisioning.
- Similar to the guest in Section 6.1, formatting the host's SD card with alternate filesystems could lessen the requirements on the virtualization layer in terms of I/O reordering, security and reliability. This would come at the cost of limiting the inter-operability of the SD card with other devices expecting a FAT filesystem or via USB mass storage. In addition, SD card FTLs are optimized for FAT and the use of other filesystems will require careful tuning [5] (or cooperation from SD card manufacturers to design hardware to suit a new filesystem standard).

  There is also a trend towards the use of eSD/eMMC chips on recent phones for internal storage. These devices share the problems of microSD cards discussed above, with the exception that they can use a custom filesystem and often employ `ext3`/`ext4`. The relationship between these filesystems, the more general I/O mixture from Android applications and middleware and the FTLs on the devices is an interesting area for exploration.
- SD card access control granularity can be improved without modifying the filesystem, for example by the use of loopback mounted encrypted images on the SD card with `dm-crypt` [24], with mounts restricted to specific applications or capabilities. This approach has been supported for read-only application code since Android 2.2, but not for application data, which is where the VM images are located. In addition, integrity checking is currently lacking from solutions based around these mechanisms on the Android and Linux platforms.
- The TRIM command for solid-state disks provides a means to detect when a block is no longer in use. TRIM commands are supported by LBS, which marks given blocks as free in the meta-data. This can reduce the garbage collector overhead and increases the write bandwidth.

  An example of where we greatly benefit from TRIM in LBS GC is the interaction between memory ballooning [30], used to balance memory between the host and guest, and the checkpoint subsystem's continuous writeback. It is common for the balloon to release large amounts of guest memory, which translates directly to the discarding of the released pages from the checkpoint image.

SD cards support erase operations which are natural candidates for the translation of TRIM or discard commands. This would enable the SD card's FTL to also reduce its garbage collection overhead. Unfortunately, the FAT filesystem does not provide a means for applications like the `vmx` to convey this information. Appropriate user-level primitives would assist here.

- Section 5.2 and Section 5.4 demonstrate the non-negligible cost of hashing and encryption in software. Mobile SoCs include dedicated cryptographic engines [2], however these are not exposed at user-level on Linux. A standard API providing access to these features would make practical their adoption in LBS when the setup and invocation costs are outweighed by the offloading advantage.

  Another area that hardware security features could assist is in key storage. As mentioned earlier, the LBS AES key is only as secure as the host application keystore, which is typically protected by file permissions only. The hardware based secure storage functionality offered on some SoCs [2] could provide further protection for keys, but again there is no standardized means to access this at the application level today which hinders a portable implementation.

### 6.3 SD card

FTLs with support for a VM or general purpose workload I/O mixture are another way to attack the mismatch LBS solves. An example of such cards are Windows Phone 7 compliant cards [29], such as the 8 GB SanDisk in Section 2, where small random writes are not penalized. However, the batching achieved by the LBS write buffer may still prove beneficial, since it amortizes the kernel context switch, host controller and SD card communication costs.

SD cards can include a small protected area for DRM-style device use [23], however this is not a general purpose secure storage mechanism and is not exported in a standard way to user-level applications. If SD cards were to provide hardware encrypted partitions, similar to some secure USB keys [13], then data-at-rest confidentiality can be guaranteed for the VM without the need for interposition at the VM layer. Protection from malicious host applications will still require effective access control at the operating system level, i.e. a generally accessible FAT partition requires encryption in the `vmx` before the data hits the filesystems.

## 7. Related work

The performance characteristics of many SD cards are studied by Bergmann and the Linaro Project [5, 16]. In addition to the observations in Section 2 regarding the disadvantage of non-sequentiality in write patterns at small block sizes, they delve into reverse engineered internal FTL characteristics. While the Linaro effort is focused on understanding these characteristics to motivate kernel changes to better support SD cards, we demonstrate application level optimizations, above the FAT layer, instead and apply them to the VM backing store domain.

Bouganim et al [7] present a benchmarking methodology for block addressed flash memory devices. Our performance characterization in Section 2 is similar in nature to a subset of the microbenchmarks they describe, targeted at the filesystem level instead of block layer. We also chose to focus on bandwidth instead of latency, since we found this both easier to measure and throughput to be a more important consideration than individual I/O latency for operations such as checkpoint suspend and resume.

While performance modeling and system optimization for SSDs has been extensively studied, e.g. [1, 8, 21, 25], SD cards have received comparably little attention in the literature, perhaps as a result of their assumed role as ancillary media stores rather than devices backing key workloads such as databases and virtual machines. Some studies have included examples of related low cost consumer storage in the form of Compact Flash (CF) cards [19] and USB flash drives [6, 7], but without publicly available SD card implementation details, it is unclear whether this related work can be applied to a given SD card.

Log structured filesystems were proposed to address the I/O bottleneck caused by fast CPU and slow disks [22]. While the original work focused on mechanical disks, the same relative penalty exists for mobile devices using commercial SD cards. The use of a log structured filesystems to improve performance on flash memory has been studied in the Cloudburst project [4]. LBS shares several of the same goals as Cloudburst, but the works differ in that LBS focuses on NAND flash rather than NOR flash. The Cloudburst virtual disk was implemented between the filesystem and the physical media, directly interfacing with NOR flash chips. LBS is a different sort of virtual disk: one that exists between a VM's virtualized media and a host filesystem. Log structured filesystems have also been employed on flash media for wear leveling and reliability [32], LBS already benefits from the wear leveling provided by an SD card's FTL but is required to add missing reliability features. DFS [15] utilizes features of a log structured FTL with large virtual address space and crash recovery support, implemented as a kernel device driver for FusionIO drives, to simplify flash filesystem implementation. The tradeoff between performance and memory use was explored by NANDFS [35], LBS delegates this decision to the kernel's page cache layer.

The concept of *reading* VM checkpoint images sequentially is discussed in a paper on fast restore [34]. In this work, the ability to quickly restore a VM is aided by sequential placement on disk of memory that is not necessarily contiguous. Work on fast checkpointing of VMs trades checkpoint speed for potential slowdown during restore due to random placement of memory blocks on disk [20]. Our work focuses on quickly *writing* checkpoints to disk, but it shares the desire to enable sequential placement of disk blocks despite memory access patterns that are not necessarily sequential.

## 8. Conclusion

The storage virtualization layer in a mobile hypervisor needs to take into account the performance, security and reliability characteristics of the host devices and filesystems providing bulk storage. In this paper we have presented the motivation, design, implementation and performance evaluation for a block storage virtualization layer capable of matching an Android VM's I/O mixture with these characteristics of commodity SD cards and the FAT filesystem. SD cards, while not designed with the use case in mind, are capable of effectively supporting a storage virtualization backend with such a layer. Going forward, there are many avenues presented in Section 6 for VMware, OEMs, silicon vendors, mobile operating systems and the systems research community to explore in further enhancing VM and general purpose storage I/O on low cost commodity devices, an area that has to date been under-explored.

## Acknowledgments

The authors would like to thank Craig Newell for his direction; Prashanth Bungale, Viktor Gyuris, Andrew Isaacson, Priti Mishra and Ian Wienand for their careful reviews; and our managers for permitting us to spend time on this paper.

## References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, June 2008.

[2] Jerome Azema and Gilles Fayad. *M-Shield^TM Mobile Security Technology: making wireless secure*. Texas Instruments, February 2008.

[3] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware Mobile Virtualization Platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 44:124–135, December 2010.

[4] Gretta Bartels and Timothy Mann. Cloudburst: A compressing, log-structured virtual disk for flash memory. Technical Report 2001-001, Compaq Systems Research Center, February 2001.

[5] Arnd Bergmann. Optimizing Linux with cheap flash drives. *Linux Weekly News*, February 2011. URL: `http://lwn.net/Articles/428584/` [visited September 2011].

[6] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Operating Systems Review*, 41(2):88–93, April 2007.

[7] Luc Bouganim, B. Jónsson, and Philippe Bonnet. uFLIP: Understanding flash IO patterns. In *Conference on Innovative Data Systems Research*, January 2009.

[8] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. Modeling and simulating flash based solid-state disks for operating systems. In *WOSP/SIPEW International Conference on Performance Engineering*, January 2010.

[9] John G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247 – 252, January 1982.

[10] Google, Inc. Nexus One [online]. URL: `http://www.google.com/phone/detail/nexus-one` [visited October 2011].

[11] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[12] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *IEEE Symposium on Massive Storage Systems and Technologies*, May 2010.

[13] IronKey. *IronKey Basic S200 datasheet*, 2009.

[14] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference*, April 2005.

[15] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage*, 6:14:1–14:25, September 2010.

[16] Linaro. Flash card survey [online]. URL: `https://wiki.linaro.org/WorkingGroups/Kernel/Projects/FlashCardSurvey` [visited September 2011].

[17] Metago. ASTRO File Manager version 2.5.2 [online]. URL: `http://market.android.com/details?id=com.metago.astro` [visited August 2011].

[18] NASA. *Ornamentation* photographs [online]. December 2010. URL: `http://www.flickr.com/photos/nasacommons/sets/72157625514008231/` [visited October 2011].

[19] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. *The VLDB Journal*, 19:67–90, February 2010.

[20] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space efficient virtual machine checkpointing. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2011.

[21] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *USENIX Annual Technical Conference*, June 2009.

[22] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles*, October 1991.

[23] SanDisk. *SanDisk SD Card Product Manual*, 2.2 edition, November 2004. Document No. 80-13-00169.

[24] Christophe Saout. dm-crypt: a device-mapper crypto target [online]. URL: `http://www.saout.de/misc/dm-crypt/` [visited October 2011].

[25] Mohit Saxena and Michael M. Swift. FlashVM: virtual memory management on flash. In *USENIX Annual Technical Conference*, June 2010.

[26] SD Group (Panasonic, SanDisk, Toshiba) and SD Card Association. *SD Specifications Part 1. Physical Layer. Simplified Specification. Version 3.01*, May 2010.

[27] Standard Performance Evaluation Corporation. SPECmail2009, March 2009. URL: `http://www.spec.org/mail2009/`.

[28] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, June 2001.

[29] Microsoft Support. Windows Phone 7 Secure Digital Card limitations [online]. URL: `http://support.microsoft.com/kb/2450831` [visited October 2011].

[30] C. Waldspurger. Memory resource management in VMware ESX Server. In *Symposium on Operating Systems Design and Implementation*. USENIX, December 2002.

[31] Seth Weintraub. Industry first: Smartphones pass PCs in sales [online]. Feb. 2011. URL: `http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010/` [visited October 2011].

[32] David Woodhouse. JFFS: The journaling flash file system. In *Ottawa Linux Symposium*, July 2001.

[33] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24:393–423, November 2006.

[34] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2011.

[35] Aviad Zuck, Ohad Barzilay, and Sivan Toledo. NANDFS: a flexible flash file system for ram-constrained systems. In *ACM International Conference on Embedded Software*, October 2009.