Memory-Centric Data Storage for Mobile Systems

Jinglei Ren, *Tsinghua University;* Chieh-Jan Mike Liang, *Microsoft Research;*
Yongwei Wu, *Tsinghua University;* Thomas Moscibroda, *Microsoft Research*

https://www.usenix.org/conference/atc15/technical-session/presentation/ren

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Memory-Centric Data Storage for Mobile Systems

Jinglei Ren
Tsinghua University[*]

Chieh-Jan Mike Liang
Microsoft Research

Yongwei Wu
Tsinghua University[*]

Thomas Moscibroda
Microsoft Research

Abstract

Current data storage on smartphones mostly inherits from desktop/server systems a flash-centric design: The memory (DRAM) effectively acts as an I/O cache for the relatively slow flash. To improve both app responsiveness and energy efficiency, this paper proposes MobiFS, a memory-centric design for smartphone data storage. This design no longer exercises cache writeback at short fixed periods or on file synchronization calls. Instead, it incrementally checkpoints app data into flash at appropriate times, as calculated by a set of app/user-adaptive policies. MobiFS also introduces transactions into the cache to guarantee data consistency. This design trades off data staleness for better app responsiveness and energy efficiency, in a quantitative manner. Evaluations show that MobiFS achieves 18.8× higher write throughput and 11.2× more database transactions per second than the default Ext4 filesystem in Android. Popular real-world apps show improvements in response time and energy consumption by 51.6% and 35.8% on average, respectively.

1 Introduction

App experience drives the success of a mobile ecosystem. Particularly, responsiveness and energy efficiency have emerged as two new crucial requirements of highly interactive mobile apps on battery-powered devices.

Recent work has shown the impact of data storage on app experience. Storage I/Os can slow down the app responsiveness by up to one order of magnitude [11, 19, 28, 36], and can substantially impact the device's energy consumption either directly or indirectly [29, 38, 50].

Modern mobile platforms typically inherit their data storage designs from desktops and servers. For example, Android and Windows Phone 8 currently default to the Ext4 and NTFS filesystem, respectively. However, these data storage designs neither reflect the different requirements nor exploit the unique characteristics of smartphones. The limited number of foreground apps, increasingly adequate DRAM capacity, and the networked nature of most apps open up a new design space that is not applicable to desktops and servers.

In this paper, we advocate a **memory-centric design** of data storage on smartphones. To elevate energy efficiency and app responsiveness as first-class metrics, we switch from the traditional flash-centric design to a memory-centric design. The flash-centric design, derived from desktops/servers, assumes the persistent flash medium as the primary store, and regards the memory (DRAM) as a temporary cache. Most recent optimizations [16, 20, 22, 36, 37, 38, 51] still follow this traditional philosophy. In contrast, we re-examine the underlying assumption of mobile storage design. Our memory-centric design views the memory as a long-lived data store, and the flash as an archival storage layer. Concretely, (1) frequent writebacks of in-memory dirty data become unnecessary; (2) individual file data synchronization calls (typically, `fsync`), which are costly, can be safely aggregated and scheduled out of the critical path of app I/O. Both changes have significant implications on app responsiveness and energy efficiency [11, 19, 28, 36]. Instead, we incrementally checkpoint app data at optimal variable intervals that adapt to app behavior, user interactions and device states.

Our key idea is to **trade off durability for energy efficiency and app responsiveness**. To realize the tradeoffs in a quantitative way, we interpret durability as a continuous variable, instead of a binary discrete variable (durable or not). Intuitively, given a specific probability of system failure, the less stale the persistent version is, the more "durable" the data is. Therefore we use data staleness as the metric of durability. Besides, these tradeoffs rely on the decoupling of durability and consistency in storage. Recent efforts have explored a similar decoupling in different domains [6, 35], but they do not apply to our memory-centric design, and they do not show how to optimize the tradeoffs for mobile apps. Overall, there is a lack of systematic and quantitative studies on these tradeoffs in mobile systems.

The gains from our design mainly come from its adaptability to mobile app behavior and usage. The tra-

[*]Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Beijing; Research Institute of Tsinghua University, Shenzhen.

ditional `fsync` and fixed flush intervals are not suited for optimizing energy performance in mobile systems and often negatively impact user experience. Instead, our measurements motivate an *app/user-adaptive* design of checkpointing for mobile apps. Concretely, we answer the algorithmic questions regarding *what* in-memory data to checkpoint (i.e., save to flash), and *when* to do so. Our solution determines the ideal checkpointing times for each app independently, by considering both device states and user interactions.

Meanwhile, loosening the timing of checkpointing is feasible in the mobile context because smartphones exhibit favorable properties. First, being self-contained (e.g., powered by batteries), smartphones are less exposed to losing data on volatile memory caused by external factors (e.g., power loss). Second, both hardware and software advances lower the data loss probability due to system crashes. Only 6% of users experience system failures more than once per month, according to our online survey. Third, most mobile apps are networked, meaning that their data are recoverable from remote servers (e.g., Gmail, Facebook, Browser). We investigate 62 most popular free apps on Google Play, and only 8 are vulnerable to local data loss (Section 3)[1].

To manipulate the checkpointing time, we change the semantics of POSIX `fsync` to be asynchronous. For some apps and databases (e.g., SQLite) that rely on synchronous `fsync` to guarantee data consistency, we design *Versioned Cache Transactions* (VCTs) to enforce atomic transactions on the filesystem cache. Combining adaptive checkpointing and VCT ensures data consistency while minimizing overhead caused by periodically frequent writebacks.

We implement our filesystem, *MobiFS*, at the system call layer for three reasons. First, this position is below upper-layer apps and databases, so it allows MobiFS to intercept and manage all storage I/O. At the same time, we can selectively enable/disable our features for individual apps[1]. Second, our solution does not alter standard filesystem interfaces, so no changes to upper-layer apps are necessary. Third, our solution is agnostic to underlying flash management implementation. For example, it can be integrated with Ext4 [33], Btrfs [45] or the latest F2FS [26].

In summary, we make multiple **contributions**. (1) We establish the feasibility and significance of the memory-centric design for mobile storage. (2) We exploit, in the mobile context, the tradeoffs between data staleness, energy efficiency and app responsiveness. MobiFS introduces transactions to the page cache of regular filesystems, without modifying app storage interfaces. (3) We propose a new measure to quantify the tradeoff between data staleness and energy efficiency, and characterize various I/O patterns of apps. These empirical results drive a policy framework that organizes and balances multiple factors – data staleness, energy, and responsiveness. (4) We implement a fully working prototype integrated with both Ext4 [33] and Btrfs [45]. Experiment results suggest up to 35.8% reduction in energy consumption and 51.6% improvement on app responsiveness, as compared to the default Android filesystem. It also achieves 18.8× higher write throughput and 11.2× more database transactions/sec.

[1]Apps with critical data (e.g., unreproducible photos) can still opt to use a regular flash partition.

2 Background

Filesystem and Page Cache. A typical filesystem consists of three main components: (1) the interface routines to serve system calls; (2) the in-memory cache of hot data, typically the *page cache*; (3) the management of the persistent media. Our work revisits two of these parts: the system call routines and the cache.

Traditional filesystems with POSIX [15] interfaces use two ways to minimize data staleness and guarantee consistency while optimizing I/O performance. (1) Asynchronous `write`[2] moves data into the page cache. Dirty pages are written to flash after a small fixed time interval (default is 5 seconds in Android). (2) Synchronous `fsync` immediately enforces data persistence of the specified file. Databases rely on `fsync` to maintain consistency. Take write-ahead logging for example: the database first records updates in a separate log, without affecting the main database file, and then invokes `fsync` over the log. This ensures a consistent state of the log file in persistent media. Finally, logged changes are applied to the main database file.

Data Consistency and Staleness. A system failure may lead to data loss in the page cache, with essentially two negative outcomes: inconsistency and staleness. Consistency in this paper refers to *point-in-time consistency* [44], meaning that the persistent data always corresponds to a point of time T in the write history – all writes before T are stored in flash and all writes after T are not. Asynchronous `write` can not guarantee consistency. Specifically, when data is kept in the page cache, it may be overwritten and results in writes being reordered. If only partial cache is flushed before a system crash, the in-flash data could violate point-in-time consistency.

Meanwhile, *data staleness* is typically less of a concern for most apps in the case of a system crash. Data staleness is the "distance" between the current volatile in-memory data and the persistent in-flash data. This distance can be measured either with respect to versions [4] or time [43].

[2]For clarity, `write` only refers to an asynchronous one without special flags such as O_SYNC.

3 Insights

The memory-centric approach has become feasible on smartphones, as advances in both hardware and software make its underlying assumptions tenable.

Insight 1 *Memory capacity on smartphones is ample enough for app data storage.*

The DRAM capacity on modern smartphones has grown significantly (8× since 2010, from 512 MB to 4 GB), with 2 GB being the standard today. This amount of memory is already sufficient to run Windows XP on a desktop. Although app data requirement has also been increasing, it has been doing so in a slower pace. For example, typical web page requests have increased in size by only 94% during the same time period [1]. Moreover, smartphone users tend to run a small number of active apps/services at the same time due to the limited screen size. Further evaluation can be found in Section 7.2.

Insight 2 *Storing app data on smartphone memory is not as risky as it sounds.*

First, smartphones have a battery power supply. Such battery-backed RAM (BBRAM) is regarded as reliable in the desktop/server setting [12, 47, 49]. Second, the reliability of smartphones has improved to the extent that memory data loss due to system failures is sufficiently rare. This observation is based on our online survey about the frequency of mobile system crashes (not app crashes) experienced by average users. Among all 117 users responding to the survey, only 6% encounter more than one failure per month, and the average frequency is once per 7.2 months. Third, most apps store data on online services or the cloud anyway.

Our detailed case study of the top 62 free apps in the Google Play app store (covering all categories, representative of most popular and frequently used apps) well supports the observation above. At one extreme, there are apps that are always in sync with online servers, e.g., Facebook, Google Maps, Glide video texting, Fitbit and most games (so does Apple's Game Center). At the other extreme, some apps rely on local data exclusively or extensively, e.g., WhatsApp (for privacy protection) and Polaris Office. Data of these apps is vulnerable before being saved to flash. Meanwhile, there are apps in between these two extremes. For example, Skype may store messages on the server for "30 to 90 days" to synchronize states across multiple devices, so the data loss risk is negligible.

Overall, only 8 apps are counted as vulnerable to local data loss, for which a system crash may largely affect user experience. Users/developers have the flexibility of configuring these apps to use a regular flash partition with traditional `fsync`. Note that these exceptions only raise a slight *configuration* burden, rather than a *programming* burden. In our experience, an app-level configuration option is more practical and easy-to-use than enforcing new programming interfaces.

Insight 3 *Reducing the amount of data flushed to flash is one key to save app energy.*

First, the write energy dominates the app I/O energy. Prior measurements [5] have shown that reading consumes about 1/6 energy of writing for the same amount of data. Meanwhile, our system-call traces of Google Play top 10 apps suggest that the data amount of reads is only 41% of writes on average. Therefore, the overall read energy is only 6.3% of write energy.

Second, the amount of data to flush, rather than the number of batches, is the dominant factor of write energy. In our experiment, writing 40 MB data in batches ranging from 4 to 40 MB results in a net energy consumption difference within 1.5% on a Samsung smartphone. In addition, standby is not a good state for data flushing, because fixed overhead can be amortized if the device is active. Up to 129% extra energy is used if data is flushed after the device switches to standby.

In conclusion, considering that the total write data issued from an app is externally determined, we can focus on how much data is overwritten before flushing, as an indicator of the app's energy efficiency.

Insight 4 *Relaxing the timing of flushes is a key to app responsiveness.*

Flushing impacts app responsiveness in two ways: (1) When flushing in a `fsync` call, the app has to wait until the data is saved to the slow flash. This situation is encountered frequently [16, 28] as databases rely on `fsync`. (2) When flushing is invoked for background writeback, it competes for CPU cycles with active app workloads, as shown in [19, 36] and from our evaluation.

In either case, the timing of flushes plays a key role: if flushing is out of the `fsync` path to avoid user interaction/CPU peaks, its negative impacts on the app responsiveness would be minimal. Our memory-centric view leverages this insight.

Insight 5 *App/user-specific I/O access patterns suggest adaptive policies to balance the staleness-energy tradeoff, which can be achieved in a quantitative way.*

I/O access patterns can vary widely among apps/users. We follow three steps to quantify this variability as well as the key tradeoff. **(1)** We define a data staleness metric that is suitable to our context. Traditional definitions are with respect to either time [43] or versions [4]. However, the time-based staleness is hardly associated with energy efficiency, and there is no strict data versioning in a regular filesystem. Instead, we define the *data staleness*, s, as the total amount of data that an app has ever written since the last checkpoint. If an app writes two pages of data to the same address, the data staleness is increased by two

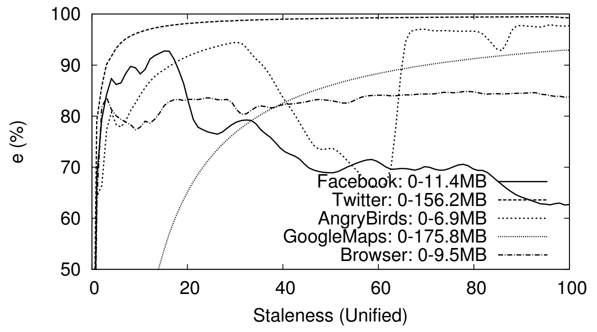

Figure 1: Different shapes of e curves suggest app-specific I/O patterns. The staleness ranges are unified to 0-100, and actual values are noted beside the app names.

pages (similar to the version-based staleness in this case). **(2)** Based on INSIGHT 3, we define the *energy efficiency ratio*, $e = o/s$, where o is the total amount of data that has been overwritten since the last checkpoint. As $o < s$, e is within [0, 1). A larger e indicates a larger proportion of pending write data to be merged before flushing and hence higher energy efficiency. **(3)** Based on the two definitions above, we draw the e curve over s to capture the extent to which increased staleness improves energy efficiency. The different shapes of e curves in Figure 1 suggest the optimal flushing time is different for different apps (and also for different users). Ideally, data in memory should be flushed when e reaches the maximum.

4 Design

The design of MobiFS is guided by insights in the previous section. As Figure 2 shows, MobiFS consists of five major components: (1) the *page cache*, which stores file data in memory; (2) the *write log*, which maintains a write history; (3) the *transactions*, which group entries in the write log and protects them from inconsistency due to overwriting/reordering; (4) the *checkpointer*, which is based on an underlying flash store to persist transactions atomically; (5) the *policy engine*, which marks transaction boundaries, detects user interactions, and decides the timing and target transactions to checkpoint.

MobiFS is designed to work with the existing page cache shared with the OS. For each `write` to the page cache, MobiFS first updates the write log, by appending a new entry or updating an existing entry with the target page address. We also maintain a *page reverse-mapping* from the dirty page back to the write log. Based on the write log and page reverse-mapping, MobiFS establishes atomic transactions. A transaction defines the scope of overwriting and reordering. Finally, the policy engine guides the checkpointer to save transactions without interfering with user interactions. It makes app-specific decisions, according to each app's behavior statistics and current device states.

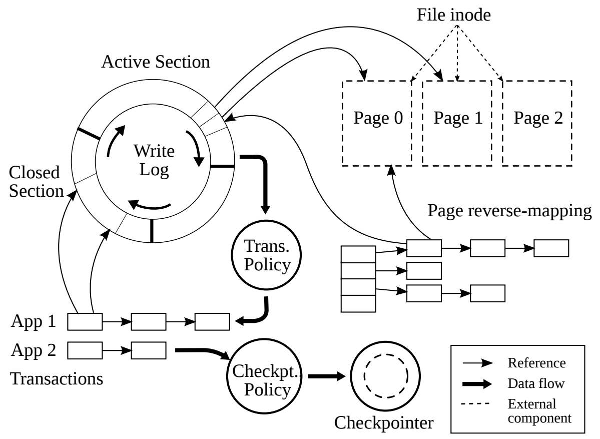

Figure 2: Architecture of MobiFS

In a typical set-up, apps have their own write log and transactions, and they share the page reverse-mapping, the policy engine and the checkpointer process(es).

4.1 Write Log

The write log is a chronological record of all writes from an app. The write log is divided into sections. New entries are inserted to the *active* section at the end of the write log, and the *closed* section contains entries that are ready for checkpointing.

The scope of a write log covers all directories accessible by an Android app (`/data/data/[PackageName]`, etc). Note that SQLite is an embedded database for individual apps, whose files are also covered by write logs.

One observation that enables app-specific optimization is the relatively static and isolated app data paths, which avoids the consistency issue of cross-app coordination. However, there might be cases where several apps have access to the same data file, e.g., a galley app and a file management app can manipulate the same set of pictures. In such a case, apps are responsible for handling situations where files are manipulated by a third party. In fact, this is the expectation of mobile operating systems such as Android.

4.2 Versioned Cache Transaction

The write log can have different versions of a cached page. *Versioned Cache Transaction* (VCT) captures this information. A VCT goes through three states in its lifetime. When it is *open*, it accepts new entries in the active section. These entries reference the latest version of a cached page. When *closed*, all entries in the active section are moved into the closed section and protected from further modifications. A VCT in the closed section can not be re-opened. When it is *committed*, all pages referenced by entries in a closed section are flushed atomically. After the commit, the VCT and its entries are evicted from the write log.

Overwriting and reordering are only allowed within a single VCT. As the checkpointer will guarantee the durability and atomicity of a committed VCT, such optimizations will not leave any inconsistent state in flash.

When a `write` comes, MobiFS has to handle three situations: (1) If the reverse-mapping of the target page does not exist, this means the app writes to a page that is not in the write log. MobiFS appends a new entry, and associates it with a new reverse-mapping. (2) If the reverse-mapping exists and points to the closed section, this means the app writes on a page within a protected VCT. MobiFS copy-on-writes over the target page. A new entry is appended for the modified copy. (3) If the reverse-mapping exists and points to the active section, this means the target page is not in a closed VCT and can be overwritten directly.

4.3 Crash Recovery

VCT boundaries do not necessarily coincide with `fsync`. In a crash, MobiFS relies on the underlying flash management component to recover any partially checkpointed VCT. Take our Ext4 variant for example. It either rolls back to the last transaction, or replays the journal to persist the latest one. In this design, apps and databases always see the data state corresponding to a point of time in history. This is true even after recovering from a system crash. We note that MobiFS guarantees consistency, but not the typical definition of durability.

4.4 Policy Engine

Two categories of policies are running in the policy engine: the *transactioning policy* and the *checkpointing policy*. The former addresses when to close a VCT, while the latter addresses when to save which VCTs into flash. Our general rule is to do checkpointing during the idle time (e.g., when a user is reading the screen content).

The concrete transactioning and checkpointing algorithms we implement in MobiFS are described in Sections 5.2, 5.3 and 5.4, respectively. However, note that our policy engine is an extensible framework, so alternative algorithms may be used.

4.5 Checkpointer

The checkpointer has two responsibilities. First, it invokes an underlying flash component to save data in flash. Second, as MobiFS is loaded, the checkpointer checks a target partition and attempts recovery of any inconsistent data. The flash management component of many filesystems like Ext4 and Btrfs can be easily adopted to implement the checkpointer. The checkpointer exposes four interfaces:

- BEGIN_TRANSACTION, invoked at the beginning of a VCT commission.
- APPEND_ENTRY, invoked for each entry in the target VCT after a successful invocation of the above.
- END_TRANSACTION, invoked at the end of a VCT commission after all its entries are appended.
- WAIT_SYNC, used if flushing is asynchronous.

The underlying flash management component should guarantee the durability and atomicity of the data written between BEGIN_TRANSACTION and END_TRANSACTION.

5 Policy

In this section, we describe our policy design and specific algorithms employed in MobiFS.

5.1 Overview

The policy design has to balance several contradictory requirements of mobile systems: data staleness, energy efficiency, and app responsiveness. We organize their relations into a modular extensible policy framework.

The policy framework assembles three modules. (1) Individual transactions are made ready for checkpointing according to the e curve, in favor of energy efficiency (Section 5.2). This does not rely on any unrealistic assumption of user operation distribution. Instead, we use a second module to predict dynamic user behaviors, so that (2) Transactions may get delayed and queued before checkpointing, in favor of app responsiveness. (Section 5.3). (3) Coordination of multiple apps is managed by a scheduling model (Section 5.4).

We make energy- and responsiveness-optimizing decisions independent, avoiding complex multi-objective optimization with simplistic assumptions. This keeps the algorithms concise as well as effective for practical systems. Many heuristics used in this section are derived from substantial first-hand experience.

5.2 Transactioning Algorithm

Increasing data staleness improves the chance of data overwriting (thus, energy saving), but it pays the price of a higher data loss risk. Hence we face the question: To what extent should MobiFS trade off data staleness for energy efficiency? MobiFS decides by evaluating the energy saving *per* data staleness unit, namely the e ratio (Section 3). Intuitively, the peak of the e curve is the best tradeoff point, as it maximizes the energy saving. Different from related efforts that set a fixed large staleness threshold [32, 35, 43], our philosophy is to reduce data loss risk unless there is a reason (improving energy efficiency) to do otherwise.

The goal of the tradeoff point location (TPL) algorithm is to determine the log entry that marks the end of the current VCT. Each `write` increments the data staleness value, which corresponds to a point in the e curve. Whenever a VCT is closed, the new curve starts at $e = 0$

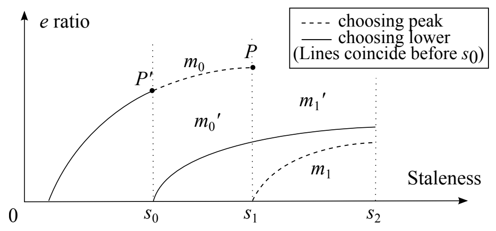

Figure 3: The e curves produced by choosing different transactioning points.

(c.f. Figure 3). Ideally, the best point to close a VCT is the peak point with highest e, as explained next. Suppose, for contradiction, that an algorithm decides to close a VCT at P' with $x = s_0$, rather than at the peak P which is $x = s_1$. Then, we can improve this algorithm by shifting the closing point to P, while keeping the subsequent closing point $x = s_2$ the same as with the supposed algorithm. We can show that by doing so, we increase the *probability* of data overwriting[3]. Without loss of generality, we let $s_0 < s_1$. The amount of overwritten data during $[s_0, s_1]$ ($[s_1, s_2]$) is m_0 (m_1) for our strategy; it is m'_0 (m'_1) for the supposed algorithm. Then we have $m_0 - m'_0 > m'_1 - m_1$ for the following reasons: $[s_0, s_1]$ still sees our curve quickly rising, so there should be much data to overwrite, yet if it is cut by the supposed algorithm, much data loses the chance to overwrite. In contrast, as P is the peak, the original curve would go down in $[s_1, s_2]$, meaning that little overwritten data is found in there. A simple transformation of the above formula leads to $m_0 + m_1 > m'_0 + m'_1$. Therefore, by cutting at P which is necessary to confine data staleness, our strategy has less a chance to overwrite data.

In practice, we have to deal with additional challenges. To mitigate fluctuations in the curve that may lead the algorithm to a locally optimal point, we use linear fitting within a sliding window. The algorithm remembers the latest k points, fits a line, and judges the peak via the gradient of the line. We choose linear fitting, instead of higher order curve fitting, because the algorithm runs on every write so that its complexity should not impose high CPU overhead. Meanwhile, we set a staleness (or time) limit to prevent the opposite – unbounded waiting for a peak. Evaluation of this algorithm is in Section 7.5.

5.3 Interval Prediction

The goal of this algorithm is to predict the length of an interval within which the user is expected not to actively operate the smartphone. These are idle intervals when flushing should be scheduled. The algorithm is triggered when there are pending VCTs. To evaluate the effectiveness of such an algorithm, we call an user operation

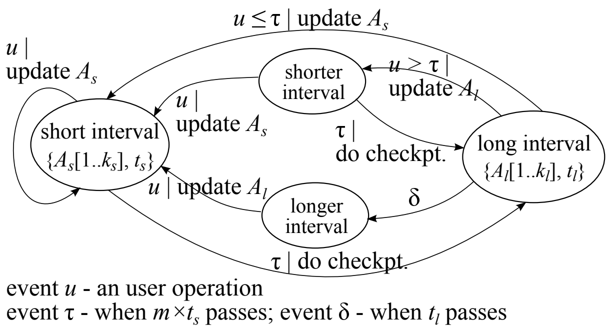
$u \leq \tau$ | update A_s
u | update A_s
shorter interval
$u > \tau$ | update A_l
u | update A_s
short interval $\{A_s[1..k_s], t_s\}$
τ | do checkpt.
long interval $\{A_l[1..k_l], t_l\}$
u | update A_l
longer interval
δ
τ | do checkpt.
event u - an user operation
event τ - when $m \times t_s$ passes; event δ - when t_l passes

Figure 4: Finite-state machine for interval prediction

unexpectedly occurring within a predicted idle interval a *responsive conflict* (RC). Note that idle interval prediction errors can cause several RCs.

We use a state-machine-based prediction method that well balances the low conflict number (i.e., low potential impact on responsiveness) and the long predicted interval (i.e., large potential energy saving as more VCTs can be merged and flushed once). Our algorithm is based on the observation that users usually switch back and forth between short and long intervals, e.g., when reading News Feeds on Facebook, the user may quickly skim some posts before spending time to read one. It is not the goal of this paper to compose a full-fledged interaction model for app users. Instead, we establish the policy framework, and show that, for our purpose, a simple state-machine model is sufficient to learn user patterns and achieve good prediction qualities (evaluated in Section 7.4).

Figure 4 depicts our finite-state machine model. There are two central states: the `short interval` (`long interval`) state when the user operates with short (long) intervals. Each of the two states maintains a recent history of intervals $A[1 \ldots k]$, and uses the minimal value t as a prediction of the next interval. Each of them also has a timer, which is set for a corresponding timeout event whenever necessary. Subscripts "s" and "l" denote the two central states, respectively. Meanwhile, the other two intermediate states, `shorter interval` and `longer interval`, help to decrease or increase interval predictions.

Intuitively, the state machine works as follows. While staying in the `short interval` state, it will loop if the coming event is a user operation. However, if the user operation does not come before a timeout event τ (Figure 4), the machine assumes that the user may begin a long interval, so it changes to the `long interval` state. Afterwards, if the predicted time t_l successfully passes without user operation (event δ), the state machine enters the `longer interval` state which waits until a user operation happens. Otherwise in the `long interval` state, if a user operation comes later than τ, we assume the user

[3]This is not a rigorous mathematical proof. Counterexamples may exist, but overall it is sufficient for the policy design.

still operates with long intervals but the interval prediction should be decreased, so it goes into the `shorter interval` state; if the user operation comes so quickly (before τ) that we guess the user switches to short intervals, the state is directly set to `short interval`.

5.4 Transaction Scheduling

The scheduling problem arises when a user interacts with multiple (background) apps or switches between apps. A typical scenario can be playing a game while listening to radio, and a background service repeatedly checks emails. MobiFS may have multiple write logs with closed VCTs for commission. The scheduler needs to prioritize VCTs to checkpoint, balancing the goals of fairness, high responsiveness and energy efficiency. Our algorithm considers three factors in the decision: (1) Transaction length, or the number of pages to checkpoint. We judge whether the transaction can fit into the predicted interval. (2) Transaction affinity. Transactions from the same app have affinity, because they can be merged if checkpointed together, thereby often saving extra energy. (3) Transaction age, the number of intervals a VCT has previously been skipped by the scheduler.

We use a priority-based scheduling algorithm, with four rules ordered in descending precedence. The algorithm maintains three queues as a way to batch VCTs of the same age. These queues have varying priority in getting flushed. Whenever a VCT is selected to be scheduled, other VCTs of the same app are prioritized. For simplicity of discussion, we may directly use apps as the unit of scheduling thereafter.

Rule 1 (transaction affinity): Whenever the scheduling algorithm skips an app in a queue, the app is moved to a higher-priority queue (if there is one).

Rule 2 (transaction age): Apps are first enqueued in the lowest-priority queue, and promoted to higher-priority queues as time goes on (as described in Rule 1). When there is no feasible choice in all queues, we find a shortest VCT in the highest urgent queue to checkpoint.

Rule 3 (transaction length): An app in the candidate queue is feasible to checkpoint only when its first VCT's length is shorter than the available predicted interval.

Rule 4 (queue replenishment): If an app is unable to checkpoint all its VCTs within a scheduled time, VCTs left are moved to the lowest-priority queue.

6 Implementation

We implement a fully-working prototype in Android 4.1 (Linux 3.0.31), and integrate it with both Ext4 [33] and Btrfs [45]. The code base has 1,996 lines of C code, excluding the reused components from Ext4 or Btrfs. MobiFS does not need kernel recompilation for deployment.

6.1 Main Components

Write Log. We implement the write log with a circular array, as it well supports the required sorting operation. To save space, some logical entry fields (e.g., the page index and version number) are compacted to a single physical data type. The write log also embeds a `kobject` structure, such that MobiFS can export user-space interfaces under the `/sys` directory for easy configuration. Moreover, the log supports certain parallelism in operations by distinguishing protection for checkpointing and appending – the tail of the circular log is protected by a spinlock, and the head is protected by a mutex that only postpones `writes` when the tail grows to reach the head. Finally, to locate which log covers a certain file, we record the log index into the `i_private` field of the `inode` structure. When a new file is created, its `i_private` is derived from its parent directory.

Page Reverse-Mapping. One approach to implement the page reverse-mapping is adding a reference pointer to the `page` structure. Since `struct page` is already packed (e.g., 24+ flags reside in a 32-bit variable), this approach requires enlarging the structure size. Instead, we opt for a customized hash table, which uses a reader-writer lock for each bucket instead of a table-wide lock. Pages associated with entries have its `_count` field incremented so that the Linux Page cache will not evict them. This also means MobiFS must unpin pages before memory space runs low, to avoid out-of-memory problems.

Checkpointer. The checkpointer is a kernel thread, which sleeps if there is no VCT to checkpoint. Upon being woken up by the policy engine, it first runs the interval prediction over the recorded user interaction history. Then, it finds the appropriate VCTs to checkpoint, according to the VCT scheduling policy.

Policy Engine. The implementation of the policy engine needs to consider some of the kernel limitations. For example, the kernel does not directly support floating points due to FPU register overheads. Hence we have to multiply e by 10^3 in our OPL algorithm to preserve thousandth precision.

User Interaction Logger. We record screen events in a queue. An issue is that some single logical user operations, such as dragging, incurs multiple events with small intervals (< 0.01 ms). Therefore, we need a filter to combine these events to one logic operation.

6.2 Integration with Storage Components

Our prototype bases its flash I/O implementation on some existing filesystem components. To support durable and atomic transactions, there are mainly two methods, the write-ahead logging (WAL) and copy-on-write (COW). Ext4 and Btrfs are two typical filesystems that use the methods, respectively.

Ext4. Ext4 uses WAL to achieve durable and atomic transactions. All file writes are first performed in a journaling area, and then moved to the main flash data set. The integration with Ext4 needs to consider the write-twice nature of Ext4 journal, where all data is written on flash twice. This may diminish the gain from MobiFS' overwriting. Fortunately, empirical results suggest that MobiFS can still achieve significant energy savings.

Btrfs. Btrfs relies on COW to achieve durable and atomic transactions. Similar to WAL, COW does not directly update the target area on flash, but makes a new copy of the data for modification. While Btrfs is highly anticipated, it is still in an experimental phase. Therefore, our MobiFS integration with Btrfs (Btr-MobiFS) is not as mature as with Ext4.

7 Evaluation

We evaluate MobiFS by three main metrics - app/user adaptability (Section 7.3), app responsiveness (Section 7.4), and energy consumption (Section 7.5). Before discussing benefits, we estimate memory footprints of MobiFS for running individual apps (Section 7.2).

7.1 Methodology

The evaluation results consist of both trace-driven simulations and actual device measurement; both benchmarks and real apps. We use a Samsung Galaxy Premier I9260 smartphone (with dual-core 1.5 GHz CPU, 1 GB RAM, Android 4.1), and two Kingston microSD cards (with the default 128 MB journal and 4 KB block size). A Monsoon Power Monitor [3] measures device energy consumption. By default, MobiFS refers to our Ext4-based implementation, and Ext4 uses the default ordered mode on Android, journaling only metadata.

Simulation traces are collected from five users operating each of the following top apps (logged in with their own accounts) for five minutes: Facebook (FB), Pandora (PA), Angry Birds (AB), Netflix (NF), Twitter (TT), Google Maps (GM), Citrix Receiver (CR), Flipboard (FL), Web Browser (WB), and WeChat (WC). Traces include I/O system calls, page cache accesses and screen touch timestamps.

Benchmarks consist of the following: (1) AnTuTu's I/O and database benchmarks, (2) RL Benchmark for SQLite (with 13 workloads), and (3) MobiBench for simulating I/O characteristics of Android system. We also use an in-house benchmark that issues sequential `writes` of 8 MB data to the same region of a file 16 times, and invokes `fsync` once every two `writes`.

For experiments that monkey real apps, we choose Browser, Facebook and Twitter, because they are representative of three typical I/O characteristics: Browser incurs few `fsyncs` and is mainly influenced by Ext4 flushing; Twitter is the opposite, triggering more than 50 `fsyncs` per second; Facebook has a moderate number of `fsyncs`. We use monkeyrunner [2] to replay programmed user interaction paths. We test Browser with an in-lab Apache2 web server via 802.11n Wi-Fi to minimize noise introduced by network dynamics.

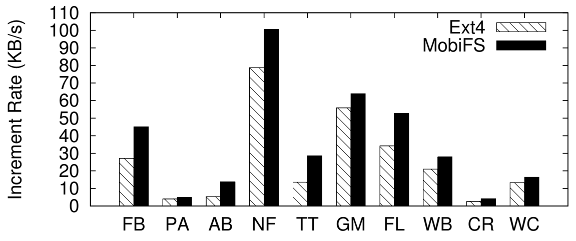

Figure 5: Memory footprint increment rates of Ext4 and MobiFS for different apps.

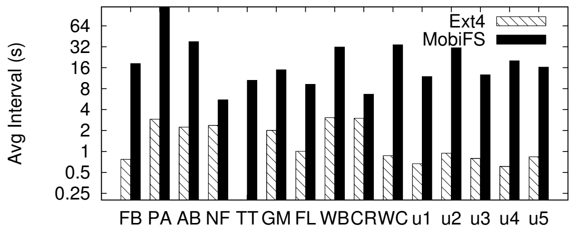

Figure 6: Adaptive checkpoint intervals of MobiFS for different apps/users. Right most user statistics (ux) are collected on Facebook.

7.2 Memory Footprint

We estimate the worst-case footprints of MobiFS according to our app I/O traces, as shown in Figure 5. It is assumed that MobiFS does not checkpoint VCTs. The y axis is the increment rate of the average memory footprint[4] introduced by the filesystem. On average, Ext4 footprints increase by 25.6 KB/s without restriction, while MobiFS incurs an increment of 35.8 KB/s. In other words, having an extra 100 MB memory, MobiFS can support an app running 17.4 minutes without flushing in the worst case (with 100.5 KB/s increment rate). Note that, when the footprint is beyond a threshold, MobiFS can deliberately execute checkpointing to release RAM space. Overall, considering that RAM is ample for apps nowadays, the footprint of MobiFS is acceptable.

7.3 App/User Adaptability

This section evaluates MobiFS' adaptability to both apps and users, as implemented by our tradeoff point location algorithm (Section 5.2).

[4]To reflect different shapes of memory footprint curves, we use integration to calculate the average. For the target increment rate α and the known integral I of the footprint curve over the time interval Δt, we suppose $\frac{1}{2}\alpha\Delta t^2 = I$, so $\alpha = 2I/\Delta t^2$.

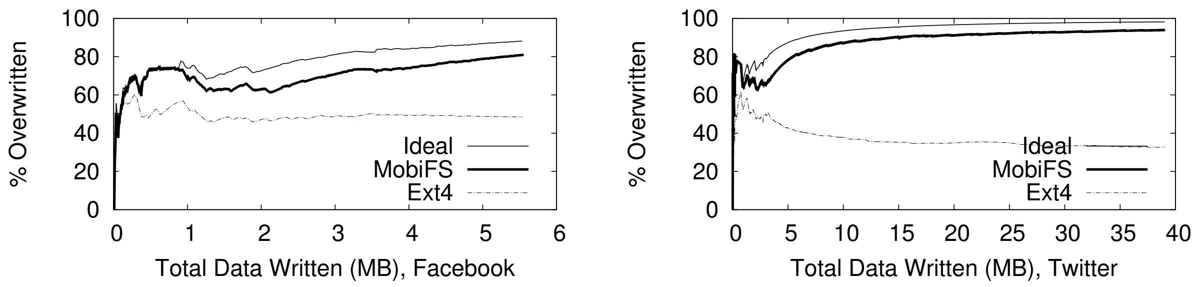

Figure 7: *e* curves against overall app written data.

Adaptability to Apps and Users. We run the tradeoff point location algorithm on the I/O traces and suppose immediate checkpointing without considering user interactions. Figure 6 shows the average of these calculated checkpoint intervals for each app. The average MobiFS checkpoint intervals fluctuate drastically, with a variance of 21.7×. Meanwhile, the geometric mean of MobiFS' average intervals is 17.5 times that of Ext4 flushes. We can see that not only does MobiFS largely extend the traditional Ext4 flush intervals, but also it is inherently adaptive to various apps.

Figure 6 also shows the average of checkpoint intervals for Facebook, as grouped by users. There is up to 2.6× variation of intervals among users for the same app. Such user-oriented adaptability is due to users exploring different contents, from different sources, and with different reading speeds.

Gains from Adaptability. Assuming no flushing should happen, the resulting *e* curve ("ideal") would present the highest potential for overwriting data. MobiFS tries to follow this ideal curve by adapting to individual apps and users. In contrast, Ext4 is limited by fixed flush intervals and traditional `fsyncs`. To illustrate MobiFS' gains from adaptability, Figure 7 compares a variant of the *e* curve with Ext4. The *e* ratio here is calculated against the overall data staleness *s* from the beginning, instead of from the last checkpoint. The observation is that MobiFS follows the ideal curve quite closely, and this higher overwrite ratio translates to energy efficiency improvement.

7.4 App Responsiveness

There are two factors that MobiFS focuses on to improve app responsiveness: minimizing responsiveness conflicts, and improving the I/O throughput.

Responsiveness Conflicts. When a user-idle interval is predicted by our interval prediction algorithm (Section 5.3), MobiFS would try to schedule a checkpointing operation to take up the full length of the interval. RCs occur when one or more unexpected user operations happen during such supposedly idle interval. We use two metrics to evaluate the prediction quality: (1) Average number of RCs per prediction; (2) Average predicted interval length. Longer intervals offer more opportunities for transaction merging which saves time/energy.

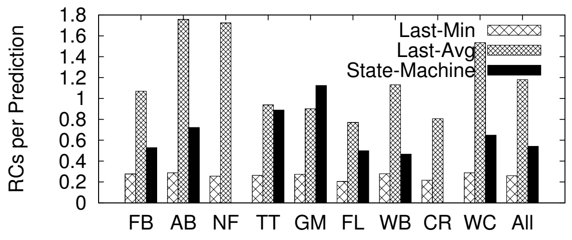

Figure 8: Responsive conflict ratios of three models.

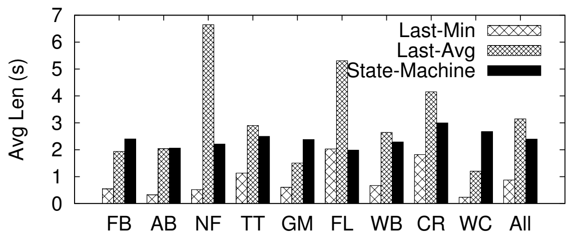

Figure 9: Average length of predicted intervals of three models.

Figures 8 and 9 separately illustrate the performance of MobiFS' state-machine-based solution on the two metrics, according to our user operation traces. It is compared with the commonly used last min model (LMM)/last average model (LAM), which predicts using the min/average of the last *k* measured intervals. As LMM always takes a conservative prediction, it inflicts only 0.26 RCs per prediction, smaller than LAM which inflicts 1.18. Naturally, our state machine algorithm cannot outperform LMM on this metric, but it achieves 54.8% less than LAM. On the other hand, LAM predicts much longer intervals than LMM. On average, our state machine achieves 75.9% length of LAM, and is over 2.7× that of LMM. As it can remember the previous "long" intervals for prediction, our state machine

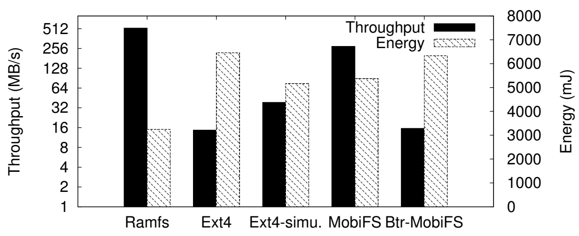

Figure 10: A baseline benchmark on different file-systems. Ext4 only journals metadata. Ext4-simu. is an Ext4 that simulates behaviors of MobiFS, i.e., it journals all data and only does so when MobiFS flushes.

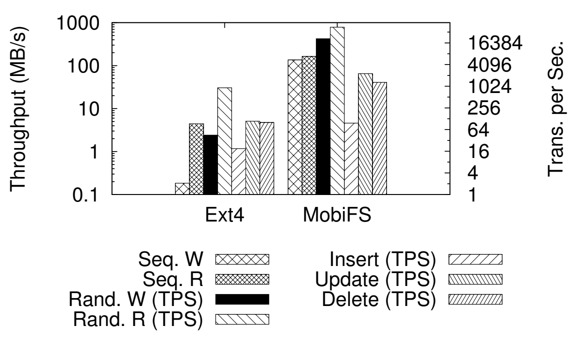

Figure 11: Itemized performance of MobiBench on Ext4 and MobiFS.

even outperforms LAM in 3 out of 9 apps. To sum up, MobiFS has a low RC numbers close to LMM, and realizes long interval length as LAM. It achieves the sweet spot between LMM and LAM.

I/O Throughput. As I/O largely affects app responsiveness, we evaluate typical I/O performance metrics of MobiFS on the device. Table 1 and Figure 11 show results from AnTuTu (ATT), RL and MobiBench. Results suggest that MobiFS can outperform Ext4 by up to 480× (e.g., random writes), and by one order of magnitude in typical database operations.

User-Perceived Latency We evaluate app responsive latency by the time required for monkeyrunner to finish a predefined user interaction path on the device. This method has advantages in (1) eliminating diversity in real user operations that are not mutually comparable and (2) reflecting user-perceived latency that excludes users' reaction time. As Figure 12 shows, monkeyrunner operates the browser to visit 50 websites, and the page loading time drops by 49.0% (-0.36 s/op) when switching from Ext4 to MobiFS. For Facebook, MobiFS reduces the time required to load the news feed five times by 53.6% (-0.85 s/op). Finally, for twitter, the time for loading #Discover tag ten times is reduced by 51.9% (-0.47 s/op). Overall, MobiFS significantly reduces the user-perceived latency of real apps.

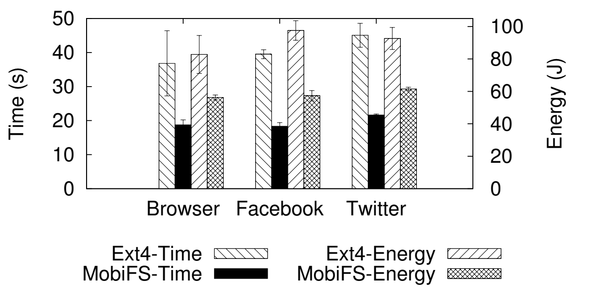

Figure 12: Responsiveness and energy consumption of apps on Android Ext4 and MobiFS.

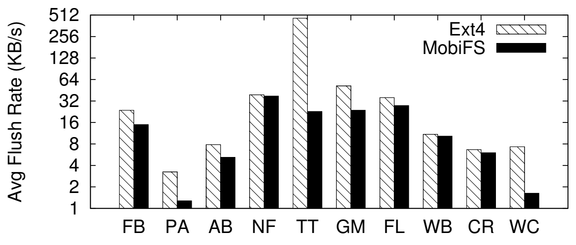

Figure 13: Flushed data of various apps on Ext4 and MobiFS.

7.5 Energy Consumption

MobiFS reduces energy consumption primarily by reducing the amount of data flushed to flash. This section first uses trace-driven simulation to quantify this reduction, and then evaluates the real device energy saving.

Reduction in Flushed Data. Figure 13 compares the amount of data flushed to the permanent storage media in the case of Ext4 and MobiFS, according to the traces. The flush data saving varies among apps, which depends on the number of overlapping writes that an app issues. By the geometric mean of all apps, 53.0% less data were flushed in the case of MobiFS, as compared to Ext4.

Our evaluation shows that MobiFS requires 66.4% more energy than regular Ext4 for the same flush size due to write-twice (Section 6.2), so the overall simulated energy cost of MobiFS is 78.3% of Ext4. Meanwhile, if we calculate average flush sizes, Ext4 flushes 4.29× the amount of data that MobiFS flushes, which means that MobiFS consumes 61.2% less energy than Ext4.

Device Energy Saving. We first consider the baseline energy figures in Figure 10. MobiFS logically flushes only half the amount of data compared to Ext4, but due to write-twice (Section 6.2) it should incur similar energy consumption with Ext4. In practice, however, both Ext4-simu. and MobiFS require less energy (over 16.8%), partially because internal data movement incurs less CPU processing than independent write system calls. On the other hand, although it does not write twice, Btr-MobiFS costs similar energy with Ext4, due to COW overheads.

Item		Ext4	MobiFS	Improve.
Perf.	ATT(score)	689.9±21.5	1817±51.0	+163%
	RL(sec.)	38.6±0.4	19.1±0.4	-50.1%
Energy* (J)	ATT	24.3±0.8	20.3±0.7	-16.4%
	RL	43.8±0.5	37.6±0.6	-14.2%

Table 1: Performance and energy of AnTuTu (ATT) and RL Benchmark on Android Ext4 and MobiFS. ATT scores favor the higher; RL time favors the lower.

Moreover, by comparing MobiFS with Ext4-simu., we can see that the energy cost of the unique components of MobiFS only counts 4.0%.

Results from benchmark tools on the device also justify MobiFS's contribution in reducing energy consumption. Table 1 shows energy savings under the AnTuTu and RL benchmarks, as compared to Ext4. Note that these workloads hardly manifest MobiFS' full potential, because our design is highly oriented to real app/user behaviors, as evaluated in the following experiment.

Figure 12 compares the energy cost of real apps in the case of MobiFS and regular Ext4. Specifically, the energy cost of the whole device drops on average by 32.1%, 41.3% and 33.6% with Browser, Facebook and Twitter, respectively. We can see that MobiFS substantially improves the energy efficiency of mobile apps.

8 Related Work

Latest Mobile Filesystems. F2FS [26] (on Moto X) observes 128% higher random write throughput than Ext4 [24]. DFS [17] improves the I/O performance by delegating storage management to the flash hardware. In contrast, our memory-centric solution can achieve nearly two orders of magnitude of improvements on read/write performance (Figure 11).

Revisiting `fsync`. MobiFS decouples the consistency and durability functionalities of `fsync`. The same methodology has been exploited to different extents. xsyncfs [40] stalls any user-visible output until the durability is accomplished. We have a more aggressive tradeoff for performance than xsyncfs, considering the unique features of mobile systems. OptFS [6] introduces `osync` and `dsync`. The former ensures only *eventual* durability. In a sense, we also follow this durability model. However, OptFS' mechanism ensures consistency of journaling disk writes by checksums, while we realize consistency in the page cache. It does not study policy design for mobile systems. Other similar work [32, 35, 43] simply uses a static time bound on staleness, and does not adaptively tradeoff in the same way as MobiFS does for mobile apps.

Memory Data Management. Main-memory databases [10, 12, 18, 41], adaptive logging [23], recoverable virtual memory [46], flash-oriented [9, 21, 31] and NVM-based [8, 14, 27, 49] storage systems optimize the performance of data flushing/writeback. NVM-based swapping [51] shows less performance improvement than our design. qNVRAM [30] implements a persistent page cache but requires new APIs to use. External journaling [16] requires extra storage devices, and does not optimize energy efficiency. Fjord [20] distinguishes apps mainly by cloud-related properties, and changes software configuration accordingly. Host-side flash caching [25] preforms a similar tradeoff between performance and staleness. Beyond all the above work, we advance at identifying minimal modifications to `fsync` and the page cache in a constrained mobile system, a systematic study of key tradeoffs, and a policy design with app/user-adaptive optimization.

Energy/Responsiveness Optimization. BlueFS [39] carefully chooses the least costly replica among multiple nodes. SmartStorage [38] sacrifices 4%-6% performance for energy efficiency by tuning storage parameters, while we achieve orders of magnitude of performance promotion along with energy saving. Capsule [34] only considers random or sequential access patterns. SmartIO [36] focuses on prioritizing reads over writes. Mobius [7] takes into account node location, network congestion, etc. While increasing I/O burstiness for energy efficiency [42, 48] shares a similar logic with us, we also consider adaptive strategies and asynchronous `fsync`. Simba [13] crafts a sync interface for both local and cloud data. Similar to Simba, we also provide consistency cross filesystem and database for local data. After all, our observation on the *e* curves and resulting multi-objective policy designs distinguish MobiFS from these above optimization works.

9 Conclusion

MobiFS identifies a fundamentally new sweet spot in the staleness-performance and staleness-energy tradeoffs that lie at the core of a filesystem for smartphones. Its new memory-centric rationale, along with app/user-adaptive incremental checkpointing, and the VCTs to support asynchronous `fsync`, provides a good reference for next-generation data storage design tailored for the mobile environment. Evaluations via user traces, micro-benchmarks, and real apps on the real device illustrate the sound policy design and practical benefits.

Acknowledgement

We thank our shepherd, Chia-Lin Yang, and the anonymous reviewers for their valuable feedback. This work is partially supported by National High-Tech R&D (863) Program of China (2012AA012600), National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), and Chinese Special Project of Science and Technology (2013zx01039-002-002).

References

[1] HTTP archive trends. `http://httparchive.org/trends.php`, 2014.

[2] The monkeyrunner tool. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`, 2015.

[3] Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`, 2015.

[4] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012).

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *USENIX ATC* (2010).

[6] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *SOSP* (2013).

[7] CHUN, B.-G., CURINO, C., SEARS, R., SHRAER, A., MADDEN, S., AND RAMAKRISHNAN, R. Mobius: unified messaging and data serving for mobile apps. In *MobiSys* (2012).

[8] COBURN, J., BUNKER, T., SCHWARZ, M., GUPTA, R., AND SWANSON, S. From aries to mars: Transaction support for next-generation, solid-state drives. In *SOSP* (2013).

[9] DAI, H., NEUFELD, M., AND HAN, R. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys* (2004).

[10] DEBRABANT, J., PAVLO, A., TU, S., STONEBRAKER, M., AND ZDONIK, S. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow. 6*, 14 (Sept. 2013).

[11] DESNOYERS, P. What systems researchers need to know about nand flash. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems* (2013), HotStorage '13.

[12] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *SIGMOD* (1984).

[13] GO, Y., AGRAWAL, N., ARANYA, A., AND UNGUREANU, C. Reliable, consistent, and efficient data sync for mobile apps. In *FAST* (2015).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994).

[15] IEEE AND THE OPEN GROUP. IEEE Std 1003.1-2008 (POSIX.1-2008). `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2015.

[16] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX ATC* (2013), pp. 309–320.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. Dfs: A file system for virtualized flash storage. In *FAST* (2010).

[18] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1*, 2 (Aug. 2008).

[19] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *FAST* (2012).

[20] KIM, H., AND RAMACHANDRAN, U. Fjord: Informed storage management for smartphones. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (2013).

[21] KIM, H., RYU, M., AND RAMACHANDRAN, U. What is a good buffer cache replacement scheme for mobile flash storage? In *SIGMETRICS* (2012).

[22] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android I/O: Multi-version b-tree with lazy split. In *FAST* (2014).

[23] KIM, Y.-S., JIN, H., AND WOO, K.-G. Adaptive logging for mobile device. *Proc. VLDB Endow. 3*, 1-2 (2010).

[24] KLUG, B. Moto x review. `http://www.anandtech.com/show/7235/moto-x-review/9`, 2013.

[25] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *FAST* (2013).

[26] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *FAST* (2015).

[27] LEE, E., KANG, H., BAHN, H., AND SHIN, K. Eliminating periodic flush overhead of file I/O with non-volatilebuffer cache. *IEEE Transactions on Computers*, 99 (2014).

[28] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *The ACM SIGBED International Conference on Embedded Software* (2012), EMSOFT '12.

[29] LI, J., BADAM, A., CHANDRA, R., SWANSON, S., WORTHINGTON, B., AND ZHANG, Q. On the energy overhead of mobile storage systems. In *FAST* (2014).

[30] LUO, H., TIAN, L., AND JIANG, H. qnvram: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014), HotStorage '14.

[31] LV, Y., CUI, B., HE, B., AND CHEN, X. Operation-aware buffer management in flash-based systems. In *SIGMOD* (2011).

[32] MA, D., FENG, J., AND LI, G. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *SIGMOD* (2011).

[33] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007).

[34] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys* (2006).

[35] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., NAREDDY, K., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., AND KHAN, O. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI* (2014).

[36] NGUYEN, D. T. Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (2014), UbiComp '14 Adjunct, ACM.

[37] NGUYEN, D. T., PENG, G., GRAHAM, D., AND ZHOU, G. Smartphone application launch with smarter scheduling. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (2014), UbiComp '14 Adjunct.

[38] NGUYEN, D. T., ZHOU, G., QI, X., PENG, G., ZHAO, J., NGUYEN, T., AND LE, D. Storage-aware smartphone energy savings. In *UbiComp* (2013).

[39] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the blue file system. In *OSDI* (2004).

[40] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *OSDI* (2006).

[41] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011).

[42] PAPATHANASIOU, A., AND SCOTT, M. Energy efficiency through burstiness. In *Fifth IEEE Workshop on Mobile Computing Systems and Applications* (2003), HotMobile '03.

[43] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *OSDI* (2010).

[44] RECOVERY SPECIALTIES, LLC. Data consistency: Explained. `http://recoveryspecialties.com/dc01.html`, 2015.

[45] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The linux b-tree filesystem. *ACM Trans. Storage (TOS) 9*, 3 (2013).

[46] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight recoverable virtual memory. In *SOSP* (1993).

[47] WANG, A.-I., REIHER, P. L., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX ATC* (2002).

[48] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: a novel I/O semantics for energy-aware applications. In *OSDI* (2002).

[49] WU, M., AND ZWAENEPOEL, W. envy: a non-volatile, main memory storage system. In *ASPLOS* (1994).

[50] XU, F., LIU, Y., MOSCIBRODA, T., CHANDRA, R., JIN, L., ZHANG, Y., AND LI, Q. Optimizing background email sync on smartphones. In *MobiSys* (2013).

[51] ZHONG, K., WANG, T., ZHU, X., LONG, L., LIU, D., LIU, W., SHAO, Z., AND SHA, E.-M. Building high-performance smartphones via non-volatile memory: The swap approach. In *The ACM SIGBED International Conference on Embedded Software* (2014), EMSOFT '14.