On the Energy Overhead of Mobile Storage Systems

Jing Li[†] Anirudh Badam[*] Ranveer Chandra[*]
Steven Swanson[†] Bruce Worthington[§] Qi Zhang[§]

[†]UCSD [*]Microsoft Research [§]Microsoft

Abstract

Secure digital cards and embedded multimedia cards are pervasively used as secondary storage devices in portable electronics, such as smartphones and tablets. These devices cost under 70 cents per gigabyte. They deliver more than 4000 random IOPS and 70 MBps of sequential access bandwidth. Additionally, they operate at a peak power lower than 250 milliwatts. However, software storage stack above the device level on most existing mobile platforms is not optimized to exploit the low-energy characteristics of such devices. This paper examines the energy consumption of the storage stack on mobile platforms.

We conduct several experiments on mobile platforms to analyze the energy requirements of their respective storage stacks. Software storage stack consumes up to 200 times more energy when compared to storage hardware, and the security and privacy requirements of mobile apps are a major cause. A storage energy model for mobile platforms is proposed to help developers optimize the energy requirements of storage intensive applications. Finally, a few optimizations are proposed to reduce the energy consumption of storage systems on these platforms.

1 Introduction

NAND-Flash in the form of secure digital cards (SD cards) [36] and embedded multimedia cards (eMMC) [13] is the choice of storage hardware for almost all mobile phones and tablets. These storage devices consume less energy and provide significantly lower performance when compared to solid state disks (SSD). Such a trade-off is acceptable for battery-powered hand-held devices like phones and tablets, which run mostly one user-facing app at a time and therefore do not require SSD-level performance.

SD cards and eMMC devices deliver adequate performance while consuming low energy. For example, an eMMC 4.5 [35] device that we tested delivers 4000 random read, and 2000 random write 4K IOPS. Additionally, it delivers close to 70 MBps sequential read, and 40 MBps sequential write bandwidth. While the sequential bandwidth is comparable to that of a single-platter 5400 RPM magnetic disk, the random IOPS performance is an order of magnitude higher than a 15000 RPM magnetic disk. To deliver this performance, the eMMC device consumes less than 250 milliwatts (see Section 2) of peak power.

Storage software on mobile platforms, unfortunately, is not well equipped to exploit these low-energy characteristics of mobile-storage hardware. In this paper, we examine the energy cost of storage software on popular mobile platforms. The storage software consumes as much as 200 times more energy when compared to storage hardware for popular mobile platforms using Android and Windows RT. Instead of comparing performance across different platforms, this paper focuses on illustrating several fundamental hardware-independent, and platform-independent challenges with regards to the energy consumption of mobile storage systems.

We believe that most developers design their applications under the assumption that storage systems on mobile platforms are not energy-hungry. However, experimental results demonstrate the contrary. To help developers, we build a model for energy consumption of storage systems on mobile platforms. Developers can leverage such a model to optimize the energy consumption of storage-intensive mobile apps.

A detailed breakdown of the energy consumption of various storage software and hardware components was generated by analyzing data from fine-grained performance and energy profilers. This paper makes the following contributions:

1. The hardware and software energy consumption of storage systems on Android and Windows RT platforms is analyzed.

2. A model is presented that app developers can use to estimate the amount of energy consumed by storage systems and optimize their energy-efficiency accordingly.
3. Optimizations are proposed for reducing the energy consumption of mobile storage software.

The rest of this paper is organized as follows. Sections 2, 3, and 4 present an analysis of the energy consumption of storage software and hardware on Android and Windows RT systems. A model to estimate energy consumption of a given storage workload is presented in Section 5. Section 6 describes a proposal for optimizing the energy needed by mobile storage systems. Section 7 presents related work, and the conclusions from this paper are given in Section 8.

2 The Case for Storage Energy

Past studies have shown that storage is a performance bottleneck for many mobile apps [21]. This section examines the energy-overhead of storage for similar apps. In particular, background applications such as email, instant messaging, file synchronization, updates for the OS and applications, and certain operating system services like logging and bookkeeping, can be storage-intensive. This section devises estimates for the proportion of energy that these applications spend on each storage system component. Understanding the energy consumption of storage-intensive background applications can help improve the standby times of mobile devices.

Hardware power monitors are used to profile the energy consumption of real and synthetic workloads. Traces, logs and stackdumps were analyzed to understand where the energy is being spent.

2.1 Setup to Measure Energy

An Android phone and two Windows RT tablets were selected for the storage component energy consumption experiments. While these platforms provide some OS and hardware diversity for the purposes of analyses and initial conclusions, additional platforms would need to be tested in order to create truly robust power models.

2.1.1 Android Setup

The battery of a Samsung Galaxy Nexus S phone running Android version 4.2 was instrumented and connected to a Monsoon Power Monitor [26] (see

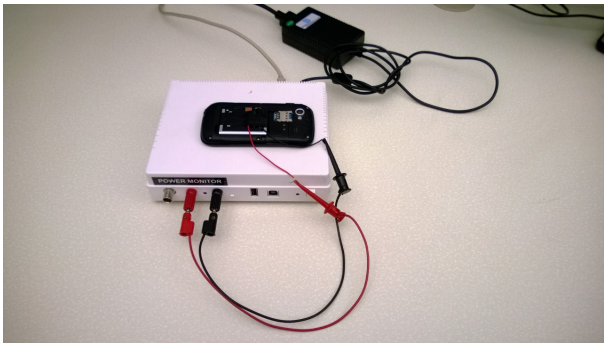


Figure 1: Android 4.2 power profiling setup: The battery leads on a Samsung Galaxy Nexus S phone were instrumented and connected to a Monsoon power monitor. The power draw of the phone was monitored using Monsoon software.

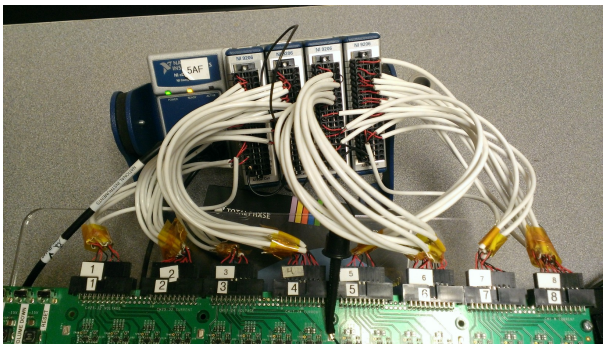


Figure 2: Windows RT 8.1 power profiling setup #1: Individual power rails were appropriately wired for monitoring by a National Instruments DAQ that captured power draws for the CPU, GPU, display, DRAM, eMMC, and other components.

Figure 3: Windows RT 8.1 power profiling setup #2: Pre-instrumented to gather fine-grained power numbers for a smaller set of power rails including the CPU, GPU, Screen, WiFi, eMMC, and DRAM.

Figure 1). In combination with Monsoon software, this meter can sample the current drawn from the battery 10's of times per second. Traces of application activity on the Android phone were captured using developer tools available for that platform [1, 2].

2.1.2 Windows RT Setup

Two Microsoft Surface RT systems were instrumented for power analysis. The first platform uses a National Instruments Digital Acquisition System (NI9206) [27] to monitor the current drawn by the CPU, GPU, display, DRAM, eMMC storage, and other components (see Figure 2). This DAQ captures 1000's of samples per second.

Figure 3 shows a second Surface RT setup, which uses a simpler DAQ chip that captures the current drawn from the CPU, memory, and other subsystems 10's of times per second. This hardware instrumentation is used in combination with the Windows Performance Toolkit [42] to concurrently profile software activity.

2.1.3 Software

Storage benchmarking tools for Android and Windows RT were built using the recommended APIs available for app-store application developers on these platforms [3, 43]. These microbenchmarks were varied using the parameters specified in Table 1. A "warm" cache is created by reading the entire contents of a file small enough to fit in DRAM at least once before the actual benchmark. A "cold" cache is created by rebooting the device before running the benchmark, and by accessing a large enough range of sectors such that few read "hits" in the DRAM are expected. The write-back experiments use a small file that is caches in DRAM in such a way that writes are lazily written to secondary storage. Such a setting enables us to estimate the energy required for writes to data that is cached. Each microbenchmark was run for one minute. The caches are always warmed from a separate process to ensure that the microbenchmarking process traverses the entire storage stack before experiencing a "hit" in the system cache.

To reduce noise, most of the applications from the systems were uninstalled, and unnecessary hardware components were disabled whenever possible (e.g., by putting the network devices into airplane mode and turning off the screen). For all the components, their idle-state power is subtracted from the power consumed during the experiment to accurately reflect only the energy used by the workload.

Parameter	Value Range
IO Size (KB)	0.5, 1, 2, 4, ..., or 1024
Read Cache Config	Warm or Cold
Write Policy	Write-through or Write-back
Access Pattern	Sequential or Random
IO Performed	Read or Write
Benchmark Language	Managed Language or Native `C`
Full-disk Encryption	Enabled or disabled

Table 1: Storage workload parameters varied between each 1-minute energy measurement.

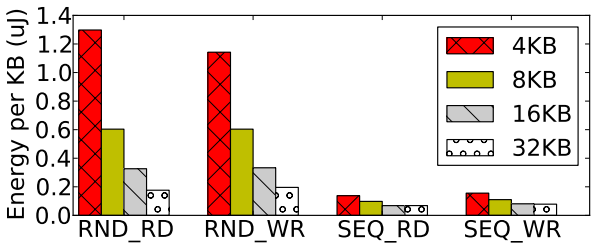


Figure 4: Storage energy per KB on Surface RT: Smaller IOs consume more energy per KB because of the per-IO cost at eMMC controller.

2.2 Experimental Results

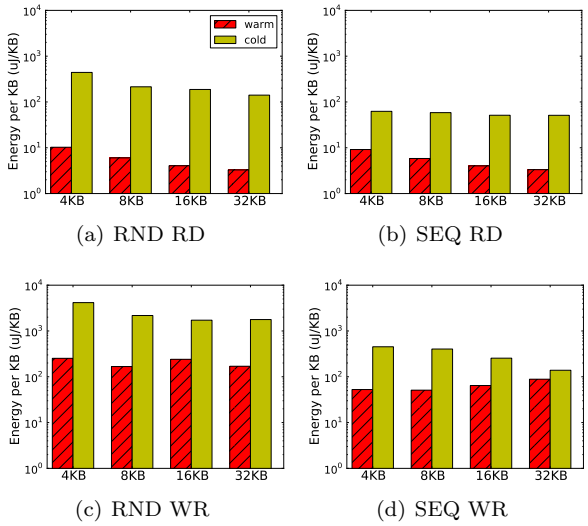
The energy overhead of the storage system was determined via microbenchmark and real application experiments. The microbenchmarks enable tightly controlled experiments, while the real application experiments provide realistic IO traces that can be replayed.

2.2.1 Microbenchmarks

Figure 4 shows the amount of energy per KB consumed by the eMMC storage for various block sizes and access patterns on the Microsoft Surface RT.

- The eMMC device requires 0.1–1.3 μJ/KB for its operations. Sequential operations are the most energy efficient from the point of view of the device.
- Random accesses of 32 KB have similar energy efficiency as sequential accesses. Smaller random accesses are more expensive – requiring more than 1 μJ/KB. This is due to the setup cost of servicing an IO at the eMMC controller level.

From a performance perspective, for a given block size, read performance is higher than write performance, and sequential IO has higher performance than random IO. We expect this to be due to the simplistic nature of eMMC controllers. Studies have shown other trends with more complex controllers [9]. For eMMC, however, the delta between read and write performance (and energy) will likely widen in the future, since eMMC devices have been increasing in read performance faster than they have been increasing in write performance.



(a) RND RD (b) SEQ RD

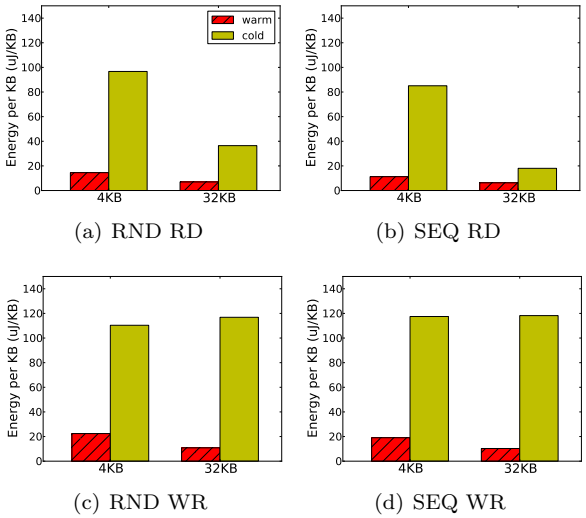
(c) RND WR (d) SEQ WR

Figure 5: System energy per KB on Android: The slower eMMC device on this platform results in more CPU and DRAM energy consumption, especially for writes. "Warm" file operations (from DRAM) are 10x more energy efficient.

Figure 5 shows that the energy per KB required by storage software on Android is two to four orders of magnitude higher than the energy consumption by the eMMC device (even though the eMMC controller in the Android platform is an older and slower generation device, the device power is in a range similar to that of the RT's eMMC device).

- Sequential reads are the most energy-efficient at the system level, requiring only one-third of the energy of random reads.
- Cold sequential reads require up to 45% more system energy than warm reads, as shown in Figure 5(b).
- Writes are one to two orders of magnitude less efficient than reads due to the additional CPU and DRAM time waiting for the writes to complete. Random writes are particularly expensive, requiring as much as 4200 μJ/KB.

The impact of low-end storage devices on performance has been well studied by Kim *et al.* [21]. Low performance, unfortunately, translates directly into high energy consumption for IO-intensive applications. We hypothesize that the idle energy consumption of CPU and DRAM (because of not entering deep idle power states soon enough) contribute to this high energy. However, we expect the energy wastage from idle power states to go down with the usage of newer and faster eMMC devices like the ones found in the tested Windows RT systems and other newer Android devices.



(a) RND RD (b) SEQ RD

(c) RND WR (d) SEQ WR

Figure 6: System energy per KB on Windows RT: The faster eMMC 4.5 card on this platform reduces the amount of idle CPU and DRAM time. "Warm" file operations (from DRAM) are 5x more energy efficient.

Figure 6 presents the energy per KB needed for the entire Windows RT platform. All "warm" IO requires less than 20 μJ/KB, whereas writes to the storage device require up to 120 μJ/KB. These energy costs are reflective of how higher performant eMMC devices can reduce energy wastage from non-sleep idle power states (tail power states). While some of this is the energy cost at the device, most of it is due to execution of the storage software, as discussed later in this section.

2.2.2 Application Benchmarks

Disk IO logs from several storage-intensive applications on Android and Windows RT were replayed to profile their energy requirements. During the replay, OS traces were captured for attributing power consumption to specific pieces of software, as well as

Email	Synchronize a mailbox with 500 emails totaling 50 MB.
File upload	Upload 100 photos totaling 80 MB to cloud storage.
File download	Download 100 photos totaling 8 0MB from cloud storage.
Music	Play local MP3 music files.
Instant messaging	Receive 100 instant messages.

Table 2: Storage-intensive background applications profiled to estimate storage software energy consumption.

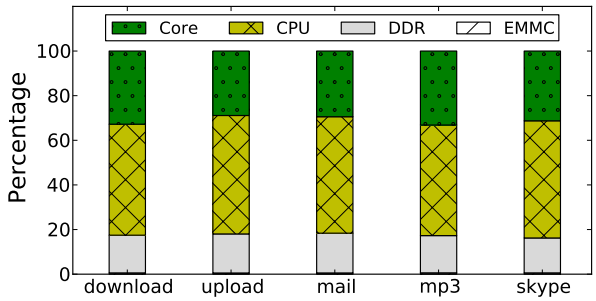


Figure 7: Breakdown of Windows RT energy consumption by hardware component. Storage software consumes more than 200x more energy than the eMMC device for background applications.

noting intervals where the CPU or DRAM were idle.

This paper focuses primarily on storage-intensive background applications that run while the screen is turned off, such as email, cloud storage uploads and downloads, local music streaming, application and OS updates, and instant messaging clients. However, many of the general observations hold true for screen-on apps as well, although display-related hardware and software tend to take up a large portion of the system energy consumption. Better understanding and optimization of the energy consumed by such applications would help increase platform standby time.

Table 2 presents the list of application scenarios profiled. Traces were taken when the device was using battery with the screen turned off.

During IO trace replay on Windows RT, power readings are captured for individual hardware components. Figure 7 plots the energy breakdown for eMMC, DRAM, CPU and Core. The "Core" power rail supplies the majority of the non-CPU compute components (GPU, encode/decode, crypto, etc.).

Library Name	% CPU Busy Time
Filesystem APIs	19.6
CLR APIs	25.8
Encryption APIs	42.1
Other APIs	12.5

Table 3: Breakdown of functionality with respect to CPU usage for a storage benchmark run on Windows RT. Overhead from managed language environment (CLR) and encryption is significant.

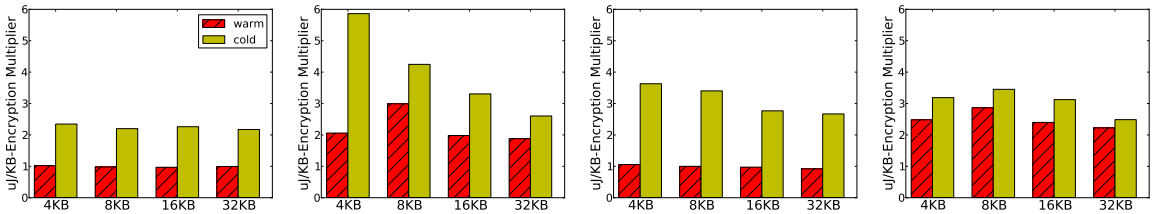
The storage software consumes between 5x and 200x more energy than the storage IO itself, depending on how the DRAM power is attributed. The fact that storage software is the primary energy consumer for storage-intensive applications is consistent with our hypothesis from the microbenchmark data. The IO traces of these applications also showed that a majority (92%) of the IO sizes were less than 64KB. We will, therefore, focus on smaller IO sizes in the rest of the paper.

Table 3 provides an overview of the stack traces collected on the Windows RT device using the Windows Performance Toolkit [42] for the mail IO workload. The majority of the CPU activity (when it was not in sleep) resulted from encryption APIs (∼42%) and Common Language Runtime (CLR) APIs (∼26%). The CLR is the virtual machine on which all the apps on Windows RT run. While there was a tail of other APIs, including filesystem APIs, contributing to CPU utilization, the largest group was associated with encryption.

The energy overhead of native filesystem APIs has been studied recently [8]. However, the overhead from disk encryption (security requirements) and the managed language environment (privacy and isolation requirements) are not well understood. Security, privacy, and isolation mechanisms are of a great importance for mobile applications. Such mechanisms not only protect sensitive user information (e.g., geographic location) from malicious applications, but they also ensure that private data cannot be retrieved from a stolen device. The following sections further examines the impact of disk encryption and managed language environments on storage systems for Windows RT and Android.

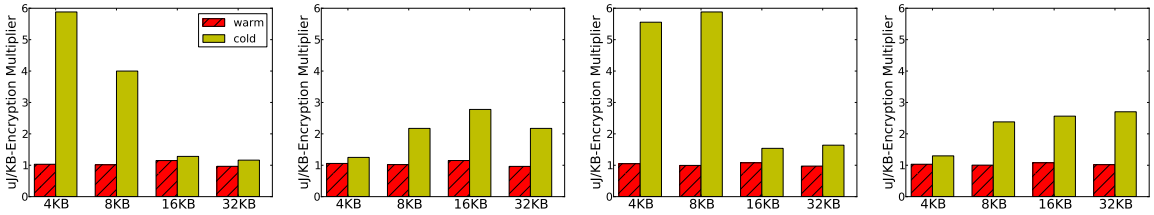
3 The Cost of Encryption

Full-disk encryption is used to protect user data from attackers with physical access to a device. Many cur-



(a) RND RD (b) RND WR (c) SEQ RD (d) SEQ WR

Figure 8: The impact of enabling encryption on the Android phone is 2.6–5.9x more energy per KB.



(a) RND RD (b) RND WR (c) SEQ RD (d) SEQ WR

Figure 9: The impact of enabling encryption on the Windows RT tablet is 1.1–5.8x more energy per KB.

rent portable devices have an option for turning on full-disk encryption to help users protect their privacy and secure their data. BitLocker [6] on Windows and similar features on Android allow users to encrypt their data. While enterprise-ready devices like Windows RT and Windows 8 tablets ship with BitLocker enabled, most Android devices ship with encryption turned off. However, most corporate Exchange and email services require full-disk encryption when they are accessed on mobile devices.

Encryption increases the energy required for all storage operations, but the cost has not been well quantified. This section presents analyses of various unencrypted and encrypted storage-intensive operations on Windows RT and Android.

Experimental Setup: Energy measurements were taken for microbenchmark workloads with variations of the first set of parameters shown in Table 1 as well as with encryption enabled and disabled while using the managed language APIs for Android, and Windows RT systems. The results are shown in Figures 8 and 9 for Android and Windows RT respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when storage encryption is enabled.

"Warm" and "cold" variations are shown. As before, "warm" represents a best-case scenario where all requests are satisfied out of DRAM. "Cold" represents a worst-case scenario where all requests require storage hardware access. In all cases, except Android writes as shown in Figures 8(b) and 8(d), "warm" runs have lower energy requirements per KB.

The cost of encryption, however, still needs to be paid when cached blocks are flushed to the storage device. Section 5 presents a model to analyze the energy consumption for a given storage workload for cached and uncached IO.

Figure 8 presents the encryption energy multiplier for the Android platform:

- The energy overhead of enabling encryption ranges from 2.6x for random reads to 5.9x for random writes.
- Encryption costs per KB are almost always reduced as IO size increases, likely due to the amortization of fixed encryption start-up costs.
- Android appears to flush dirty data to the eMMC device aggressively. Even for small files that can fit entirely in memory and for experiments as short as 5 seconds, dirty data is flushed, thereby incurring at least part of the energy overhead from encryption. Therefore, Android's caching algorithms do not delay the encryption overhead as much as expected. They may also not provide as much opportunity for "over-writes" to reduce the total amount of data written, or for small sequential writes to be concatenated into more efficient large IOs.

Figure 9 presents the energy multiplier for enabling BitLocker on the Windows RT platform:

- The energy overhead of encryption ranges from 1.1x for reads to 5.8x for writes.
- The energy consumption correlation with request size is less obvious for the Windows platform. While increasing read size generally reduces energy costs because of the usage of crypto engines for larger sizes, as was the case for the Android platform, write sizes appear to have the opposite trend. All of the shown request sizes are fairly small when the CPU was used for encryption; we found that that this trend reverses as request sizes increased beyond 32 KB.
- DRAM caching does delay the energy cost of encryption for reads and writes, even for experiments as long as 60 seconds. This could provide opportunity to reduce energy because of over-writes, and also due to read prefetching at larger IO sizes and concatenation of smaller writes to form larger writes.

On Windows RT, encryption and decryption costs are highly influenced by hardware features and software algorithms used. Hardware features include the number of concurrent crypto engines, the types of encryption supported, the number of engine speeds (clock frequencies) available, the amount of local (dedicated) memory, the bandwidth to main memory, and so on. Software can choose to send all or part (or none) of the crypto work to the hardware crypto engines. For example, small crypto tasks are faster on the general purpose CPU. Using the hardware crypto engine can produce a sharp drop in energy consumption when the size of a disk IO reaches an algorithmic inflection point with regard to performance. See Section 6 for a hardware optimization we propose to bring down the energy cost of encryption for all IO sizes.

4 The Runtime Cost

Applications on mobile platforms are typically built using managed languages and run in secure containers. Mobile applications have access to sensitive user data such as geographic location, passwords, intellectual property, and financial information. Therefore, running them in isolation from the rest of the system using managed languages like Java or the Common Language Runtime (CLR) is advisable. While this eases development and makes the platform more secure, it affects both performance and energy consumption.

Any extra IO activity generated as a result of the use of managed code can significantly increase the average storage-related power, especially since mobile storage has such a low idle power envelope. This section explores the performance and energy impact of using managed code.

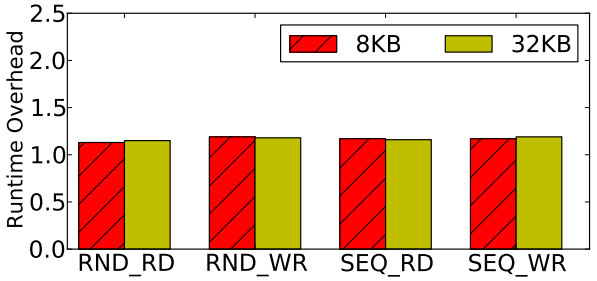


Figure 10: Impact of managed programming languages on Windows RT tablet: 13–18% more energy per KB for using the CLR.

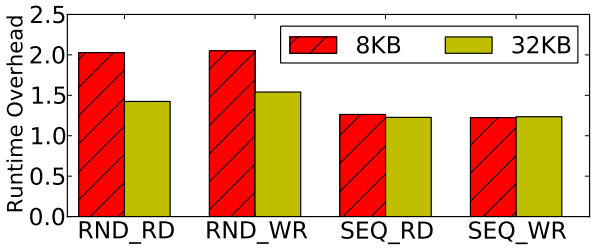
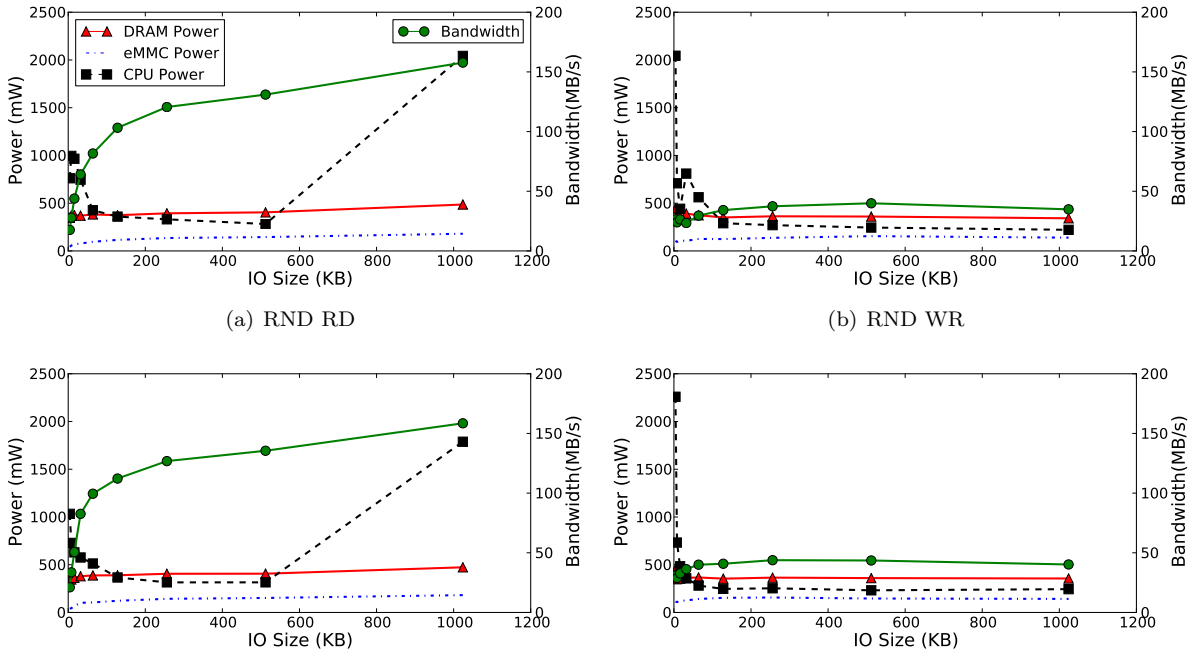


Figure 11: Impact of managed programming language on Android phone: 24–102% more energy per KB for using the Dalvik runtime.

Experimental Setup: The first set of parameters from Table 1 are again varied during a set of microbenchmarking runs using native and managed code APIs for Windows RT, and Android with encryption disabled. The pre-instrumented Windows RT tablet is specially configured (via Microsoft-internal functionality) to allow the development and running of applications natively. The native version of the benchmarking application uses the `OpenFile`, `ReadFile`, and `WriteFile` APIs on Windows. The Android version uses the Java Native Interface [20] to call the native `C fopen`, `fread`, `fseek`, and `fwrite` APIs.

The measured energy consumption for the Windows and Android platforms are shown in Figures 10, and 11, respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when using managed rather than native code.

- On Windows RT, the energy overhead on storage systems from running applications in a managed environment is 12.6–18.3%.



(a) RND RD

(b) RND WR

(c) SEQ RD

(d) SEQ WR

Figure 12: Power draw by DRAM, eMMC, and CPU for different IO sizes on Windows RT with encryption disabled. CPU power draw generally decreases as the IO rate drops. However, large (e.g., 1 MB) IOs incur more CPU path (and power) because they trigger more working set trimming activity during each run.

- The overhead on Android is between 24.3–102.1%. We believe that the higher energy overhead for smaller IO sizes (some not shown) is likely due to a larger prefetching granularity used by the storage system. For larger IO sizes (some not shown), the overhead was always lower than 25%.

Security and privacy requirements of applications on mobile platforms clearly add an energy overhead as demonstrated in this section and the previous one. If developers of storage-intensive applications take these overheads into account, more energy-efficient applications could be built. See Section 6 for a hardware optimization that we propose for reducing the energy overhead due to the isolation requirements of mobile applications.

5 Energy Modeling for Storage

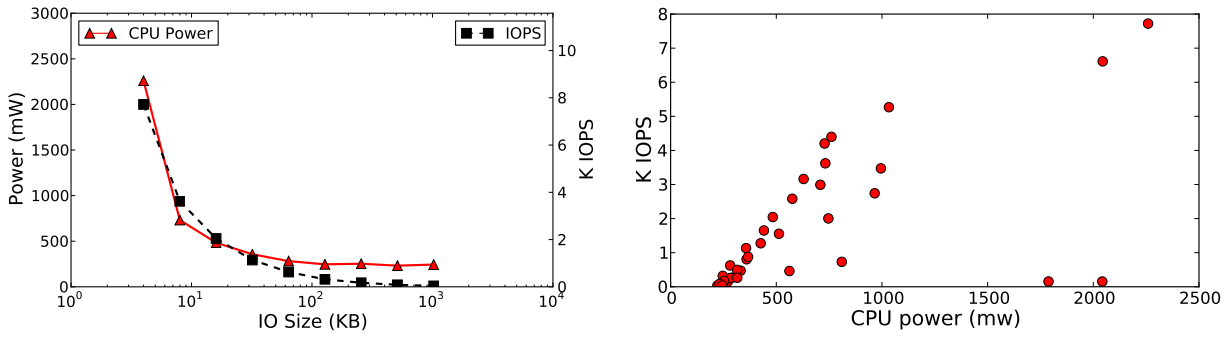
As shown in the previous sections, encryption and the use of managed code add a significant amount of overhead to the storage APIs – in terms of energy. Therefore, we believe that it is necessary to empower developers with tools to understand and optimize the energy consumed by their applications with regard to storage APIs.

This section first attempts to formalize the energy consumption characteristics of the storage subsystem. It then presents EMOS (Energy MOdeling for Storage), a simulation tool that an application or OS developer can use to estimate the amount of energy needed for their storage activity. Such a tool can be used standalone or as part of a complete energy modeling system such as WattsOn [25]. For each IO size, request type (read or write), cache behavior (hit or miss), and encryption setting (disabled or enabled), the model allows the developer to obtain an energy value.

5.1 Modeling Storage Energy

The energy cost of a given IO size and type can be broken down into its power and throughput components. If the total power of read and write operations are P_r and P_w, respectively, and the corresponding read and write throughputs are T_r and T_w KB/s, then the energy consumed by the storage device per KB for reads (E_r) and writes (E_w) is:

$$E_r = P_r/T_r, E_w = P_w/T_w$$



(a) CPU vs IOPS Correlation

(b) CPU vs IOPS Scatter plot

Figure 13: CPU power & IOps for different sizes of random and sequential reads on the Surface RT. Both metrics follow an exponential curve and show good linear correlation. The two outliers in the scatter plot towards the bottom right are caused by high read throughput triggering the CPU-intensive working set trimming process in Windows RT.

The hardware "energy" cost of accessing a storage page depends on whether it is a read or a write operation, file cache hit or miss, sequential or random, encrypted or not, and other considerations not covered by this analysis, such as request inter-arrival time, interleaving of different IO sizes and types, and the effects of storage hardware caches or device-level queuing.

In this model, P is comprised of CPU(P_{CPU}), memory (P_{DRAM}), and storage hardware(P_{EMMC}) power. Figure 12 shows the variation of each of these power components for uncached, unencrypted, random, and sequential, reads and writes via managed language microbenchmarking apps that we described in Section 2.

P_{DRAM} can be modeled as follows:

- For writes, the DRAM consumes 450 mW when the IO size is less than 8 KB. When the IO size is greater than or equal to 8 KB, this power is closer to 360 mW. This may be due to a change in memory bus speed for smaller IOs (with more IOps and higher CPU requirements driving up the memory-bus frequency).
- For reads, DRAM power increases linearly with request size from 350 mW for 4 KB reads to 475 mW for 1 MB reads. Write throughput rates are low enough that DRAM power variation for different write sizes is low. This is likely caused by more "active" power draw at the DRAM and the controller as utilization increases.

Storage unit power (P_{EMMC}) can be modeled as follows:

- For writes, the eMMC power variation due to sequentiality and request size is fairly low – from 105 mW for 4 KB IOs to 140 mW for 1 MB IOs.
- For random and sequential reads, the eMMC power varies from 40 mW for 4 KB IOs to 180 mW for 1 MB IOs, with most of the variation coming from IO sizes less than 4 KB. 4KB or less IOs are traditionally more difficult for these types of eMMC drives, because some of their internal architecture is optimized for transfers that are 8KB or larger (and aligned to corresponding logical address boundaries).

The graphs show that P_{CPU} follows an exponential curve with respect to the IO size. However, the CPU power actually tracks the storage API IOps curve, which is T/IO_size. Since IOps actually follows an exponential curve when plotted against IO size, a linear correlation exists between P_{CPU} and IOps (see Figure 13). The two scatter plot outliers that consume high CPU power at low IOps are the 1 MB sequential and random read operations. The bandwidth of these workloads (160 MB/s) was large enough and the experiments were long enough for the OS to start trimming working sets. If the other request size experiments were run for long enough, they would also incur some additional power cost when trimming finally kicks in.

With Encryption: If similar graphs were plotted for the experiments with encryption enabled, the following would be seen for the Surface RT:

- All component power values generally increase with IO size.
- P_{DRAM} is higher for reads than writes, staying fairly constant at 515 mW. For writes, the

Platform	Caching	IO Size	RND_RD	RND_WR	SEQ_RD	SEQ_WR
Windows RT	Hit	8KB	14.2	22.4	11.2	19.0
		32KB	11.4	18.2	8.6	18.2
	Miss	8KB	96.7	110.4	85.0	117.5
		32KB	36.4	116.8	18.0	118.2
Android	Hit	4KB	10.3	252.9	9.1	52.6
		8KB	6.0	167.2	5.8	51.0
		16KB	4.0	240.7	4.0	64.4
		32KB	3.3	169.7	3.3	88.5
	Miss	4KB	441.9	2402.7	62.5	451.8
		8KB	214.4	2176.7	58.5	403.5
		16KB	187.6	1720.9	51.3	254.9
		32KB	141.0	1776.0	51.1	138.8

Table 4: Energy (uJ) per KB for different IO requests. Such tables can be built for a specific platform and subsequently incorporated into power modeling software usable by developers for optimizing their storage API calls.

power increases linearly with IO size, varying from 370 mW for 4 KB IOs to 540 mW for 1 MB IOs. This variation is mostly because of the extra memory needed for encryption to complete.

- P_{EMMC} values for reads and writes are similar to their unencrypted counterparts. Given that encryption (and decryption) in current mobile devices is handled using on-SoC hardware, this is to be expected.
- P_{CPU} is fairly linear with IOPS for reads, but the power characteristics for writes are more complex. This may be due to the dynamic encryption algorithms discussed previously, where request size factors into the decision on whether to use crypto offload engines or general-purpose CPU cores to perform the encryption.

Specific measurements can change for newer hardware, however the general trends that we expect to hold are the following: P_{DRAM} would be significantly higher when encryption is enabled vs when it is disabled. This will be true as long as the hardware crypto engines do not have enough dedicated RAM. P_{EMMC} is expected to be the same whether encryption is enabled or disabled as long as the crypto engines are inside the SoC and not packaged along with the eMMC device. P_{CPU} is expected to be higher when encryption is enabled as long as the hardware crypto engines are unable to meet the throughput requirements of storage for all possible storage workloads. P_{CPU} is also expected to be correlated with the application level IOps because of software setup costs required on a per IO basis. The power trends for reads vs. writes will continue as long as eMMC controllers increase read performance at a faster pace than write performance.

5.2 The EMOS (Energy MOdeling for Storage) Simulator

The EMOS simulator takes as input a sequence of timestamped disk requests and the total size of the filesystem cache. It emulates the file caching mechanism of the operation system to identify hits and misses. Each IO is broken into small primitive operations, each of which has been empirically measured for its energy consumption.

Ideally, component power numbers (P_{CPU}, P_{DRAM}, and P_{EMMC}) would be generated for every platform. It is infeasible for a single company to take on this task, but the possibility exists for scaling out the data capture to a broader set of manufacturers. For the purposes of this paper, the EMOS simulator is tuned and tested on the Microsoft Surface RT, and Samsung Nexus S platforms.

For each platform, the average energy needed for completing a given IO type (read/write, size, cache hit/miss) is measured. The energy values are aggregated from DRAM, CPU, eMMC, and Core (idle energy values are subtracted). A table such as Table 4 can be populated to summarize the measured energy consumption required for each type of storage request. We show only a few request sizes in the

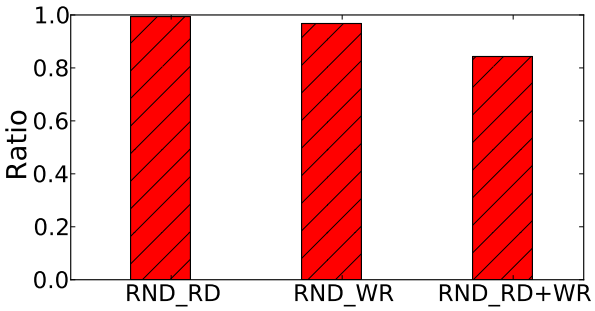


Figure 14: Experimental validation of EMOS on Android shows greater than 80% accuracy for predicting 4KB IO microbenchmark energy consumption.

table for the sake of brevity.

Simulation of cache behavior: Cache hits and misses have different storage request energy consumption. Since many factors affect the actual cache hit or miss behavior (e.g., replacement policy, cache size, prefetching algorithm, etc.), a subset of the possible cache characteristics was selected for EMOS. For example, only the LRU (Least Recently Used) cache replacement policy is simulated, but the cache size and prefetch policy are configurable.

EMOS was validated using the 4 KB random IO micro-benchmarks on the Android platform without any changes to the default cache size, or prefetch policy. The measured versus calculated energy consumption of the system were compared for workloads of 100% reads, 100% writes, and a 50%/50% mix. Figure 14 shows that while the model is accurate for pure read and write workloads, it is only 80% accurate for a mixed workload. We attribute this to the IO scheduler and the file cache software behaving differently when there is a mix of reads and writes, as well as changes in eMMC controller behavior for mixed workloads. Future investigations are planned to fully account for these behaviors.

6 Discussion: Reducing Mobile Storage Energy

We suggest ways to reduce the energy consumption of the storage stack through hardware and software modifications.

6.1 Partially-Encrypted File systems

While full-disk encryption thwarts a wide range of physical security attacks, it may be an overkill for some scenarios. It puts an unnecessary burden on accessing data that does not require encryption. For example, most OS files, application binaries, some caches, and possibly even media purchased online may not need to be encrypted. A naive solution would be to partition the disk into encrypted and unencrypted file systems / partitions. However, if free space cannot be dynamically shifted between the partitions, this solution may result in wasted disk space. More importantly, some entity has to make decisions about which files to store in which file systems, and the user would need to explicitly make some of these decisions in order to achieve optimal and appropriate partitioning. For example, a user may or may not wish his or her personal media files to be visible if a mobile device is stolen.

Partially-encrypted filesystems that allow some data to be encrypted while other data is unencrypted represent a better solution for mobile storage systems. This removes the concern over lost disk space, but some or all of the difficulties associated with the encrypt-or-not decision remain. Nevertheless, opens the option for individual applications to make some decisions about the privacy and security of files they own, perhaps splitting some files in two in order to encrypt only a portion of the data contained within. This increases development overhead, but it does provide applications with a knob to tune their energy requirements.

GNU Privacy Guard [19] for Linux and Encrypting File Systems [15] on Windows provide such services. However, care must be taken to ensure that unencrypted copies of private data not be left in the filesystem at any point unless the user is cognizant (and accepting) of this vulnerability. Additional security and privacy systems are needed to fully secure partially-encrypted file systems. Once the data from an encrypted file has been decrypted for usage, it must be actively tracked using taint analysis. Information flow control tools [14, 18, 46] are required to ensure that unencrypted copies of data are not left behind on persistent storage for attackers to exploit.

6.2 Storage Hardware Virtualization

Low-cost storage targeted to mobile platforms relies on storage software features. Isolation between applications is provided using managed languages, per-application users and groups, and virtual machines on Android and Windows RT for applications developed in Java and .NET, respectively. Storage software overhead can be reduced by moving much of this complexity into the storage hardware [8].

Mobile storage can be built in a manner such that each application is provided with the illusion of a

private filesystem. In fact, Windows RT already provides such isolation using only software [28]. Moving such isolation mechanisms into hardware can enable managed languages to directly use native APIs for applications to obtain native software like energy-usage with isolation guarantees.

6.3 SoC Offload Engines for Storage

Various components inside mobile platforms have moved their latency- and energy-intensive tasks to hardware. Audio, video, radio, and location sensors have dedicated SoC engines for frequent, narrowly-focused tasks, such as decompression, echo cancellation, and digital signal processing. This type of optimization may also be appropriate for storage. For example, the SoC can fully support encryption and improve hardware virtualization. Some SoC's already support encryption in hardware, but they do not meet the throughput expectations of applications. Crypto engines inside SoCs must be designed to match the throughput of the eMMC device at various block sizes to reduce the dependence of the OS on energy-hungry general-purpose CPU for encryption. Dedicated hardware engines for file system activity could provide metadata or data access functionality while ensuring privacy, and security.

7 Related Work

To our knowledge, a comprehensive study of storage systems on mobile platforms from the perspective of energy has not been presented to date. Kim et al [21] present a comprehensive analysis of the performance of secondary storage devices, such as SD cards often used on mobile platforms. Past research studies have presented energy analysis of other mobile subsystems, such as networking [4, 17], location sensing [41], the CPU complex [24], graphics [40], and other system components [5]. Carroll *et al.* [7] present the storage energy consumption of SD cards using native IO. Shye *et al.* [38] implement a logger to help analyze and optimize energy consumption by collecting traces of software activities.

Energy estimation and optimization tools [12, 47, 16, 25, 31, 30, 34, 33, 45] have been devised to estimate how much energy an application consumes during its execution. This paper uses similar techniques to analyze energy requirements from the perspective of the storage stack as opposed to a broader OS perspective or a narrower application perspective.

Energy consumption of storage software has been analyzed in the past for distributed systems [23], servers [32, 37, 39], PCs [29] and embedded systems [10], as opposed to the mobile platforms analyzed in this paper. Mobile storage systems are sufficiently different from these systems because of their security, privacy, and isolation requirements. This paper examines the energy overhead of these requirements.

Storage systems using new memory technologies like phase-change memory (PCM) focus on analyzing and eliminating the overhead from software [8, 11, 22, 44]. However, existing storage work for new memory technologies focuses only on native IO performance. This paper also includes analysis of managed language environments.

8 Conclusions

Battery life is a key concern for mobile devices such as phones and tablets. Although significant research has gone into improving the energy efficiency of these devices, the impact of storage (and associated APIs) on battery life has not received much attention. In part this is due to the low idle power draw of storage devices such as eMMC storage.

This paper takes a principled look at the energy consumed by storage hardware and software on mobile devices. Measurements across a set of storage-intensive microbenchmarks show that storage software may consume as much as 200x more energy than storage hardware on an Android phone and a Windows RT tablet. The two biggest energy consumers are encryption and managed language environments. Energy consumed by storage APIs increases by up to 6.0x when encryption is enabled for security. Managed language storage APIs that provide privacy, and isolation consume 25% more energy compared to their native counterparts.

We build an energy model to help developers understand the energy costs of security and privacy requirements of mobile apps. The EMOS model can predict the energy required for a mixed read/write micro-benchmark with 80% accuracy. The paper also supplies some observations on how mobile storage energy efficiency can be improved.

9 Acknowledgments

We would like to thank our shepherd, Brian Noble, as well as the anonymous FAST reviewers. We would like to thank Taofiq Ezaz, and Mohammad Jalali for helping us with the Windows RT experimental setup. We would also like to thank Lee Prewitt, and Stefan Saroiu for their valuable feedback.

References

[1] Android Application Tracing. http://developer.android.com/tools/debugging/debugging-tracing.html.

[2] Android Full System Tracing. http://developer.android.com/tools/debugging/systrace.html.

[3] Android Storage API. http://developer.android.com/guide/topics/data/data-storage.html.

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. ACM IMC*, Chicago, IL, Nov. 2009.

[5] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus Energy Tradeoffs in Host-Based Mobile Malware Detection, June 2011.

[6] BitLocker Drive Encrytion. http://windows.microsoft.com/en-us/windows7/products/features/bitlocker.

[7] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[8] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS*, London, United Kingdom, Mar. 2012.

[9] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proc. ACM SIGMETRICS*, Seattle, WA, June 2009.

[10] S. Choudhuri and R. N. Mahapatra. Energy Characterization of Filesystems for Diskless Embedded Systems. In *Proc. 41st DAC*, San Diego, CA, 2004.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM SOSP*, Big Sky, MT, Oct. 2009.

[12] M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[13] eMMC 4.51, JEDEC Standard. http://www.jedec.org/standards-documents/results/jesd84-b45.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy MOnitoring on Smartphones. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[15] Encrypting File System for Windows. http://technet.microsoft.com/en-us/library/cc700811.aspx.

[16] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation of Mobile Applications, Dec. 1999.

[17] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[18] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An Auditing File SYstem for Theft-Prone Devices. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[19] GNU Privacy Guard: Encrypt files on Linux. http://www.gnupg.org/.

[20] Java Native Interface. http://developer.android.com/training/articles/perf-jni.html.

[21] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage on Smartphones. 8(4):14:1–14:25, 2012.

[22] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proc. 11th USENIX FAST*, San Jose, CA, Feb. 2013.

[23] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *ACM SIGOPS OSR*, 44:61–65, 2010.

[24] A. P. Miettinen and J. K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proc. 2nd USENIX HotCloud*, Boston, MA, June 2010.

[25] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proc. 18th ACM MobiCom*, Istanbul, Turkey, Aug. 2012.

[26] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[27] National Instruments 9206 DAQ Toolkit. http://sine.ni.com/nips/cds/view/p/lang/en/nid/209870.

[28] .NET Isolated Storage API. http://msdn.microsoft.com/en-us/

library/system.io.isolatedstorage.isolatedstoragefile.aspx.

[29] E. B. Nightingale and J. Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proc. 5th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[30] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine Grained Energy Accounting on Smartphones. In *Proc. 7th ACM EUROSYS*, Bern, Switzerland, Apr. 2012.

[31] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones using System Call Tracing. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[32] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proc. 18th ACM ICS*, Saint-Malo, France, June 2004.

[33] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatschek. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[34] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich. Energy Management in Mobile Devices with Cinder Operating System. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.

[35] Samsung eMMC 4.5 Prototype. http://www.samsung.com/us/business/oem-solutions/pdfs/eMMC_Product%20Overview.pdf.

[36] Secure Digital Card Specification. https://www.sdcard.org/downloads/pls/simplified_specs/.

[37] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[38] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. 42nd IEEE MICRO*, New York, NY, Dec. 2009.

[39] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage. In *Proc. 6th USENIX FAST*, San Jose, CA, 2008.

[40] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Signh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *Proc. WWW*, Lyon, France, Apr. 2012.

[41] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh-Koniecpol. A Framework for Energy Efficient Mobile Sensing for Automatic Human State Recognition. In *Proc. 7th ACM Mobisys*, Krakow, Poland, June 2009.

[42] Windows Performance Toolkit. http://msdn.microsoft.com/en-us/performance/cc825801.aspx.

[43] Windows RT Storage API. http://msdn.microsoft.com/en-us/library/windows/apps/hh758325.aspx.

[44] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proc. IEEE/ACM SC*, Seattle, WA, Nov. 2011.

[45] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In *Proc. USENIX ATC*, Boston, MA, June 2012.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *Proc. 7th USENIX OSDI*, Seattle, WA, Dec. 2006.

[47] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. 8th IEEE/ACM/IFIP CODES+ISSS*, Taipei, Taiwan, 2010.