Design and Implementation of an Overlay File System for Cloud-Assisted Mobile Apps

Nafize Rabbani Paiker, Jianchen Shan, Cristian Borcea, *Member, IEEE*, Narain Gehani, Reza Curtmola, *Member, IEEE*, and Xiaoning Ding

Abstract—With cloud assistance, mobile apps can offload their resource-demanding computation tasks to the cloud. This leads to a scenario where computation tasks in the same program run concurrently on both the mobile device and the cloud. An important challenge is to ensure that the tasks are able to access and share the files on both the mobile and the cloud in a manner that is efficient, consistent, and transparent to locations. Existing distributed file systems and network file systems do not satisfy these requirements. Current systems for offloading tasks either do not support file access for offloaded tasks or do not offload tasks with file access. The paper addresses this issue by designing and implementing an application-level file system called *Overlay File System* (OFS). To improve efficiency, OFS maintains and buffers local copies of data sets on both the cloud and the mobile device. OFS ensures consistency and guarantees that all the reads get the latest data. It combines write-invalidate and write-update policies to effectively reduce the network traffic incurred by invalidating/updating stale data copies and to reduce the execution delay when the latest data cannot be accessed locally. To guarantee location transparency, OFS creates a unified view of the data that is location independent and is accessible as local storage. We overcome the challenges caused by the special features of mobile systems on an application-level file system, like the lack of root privilege and state loss when application is killed due to the shortage of resource and implement an easy to deploy prototype of OFS. The paper tests the OFS prototype on Android OS with a real mobile app and real mobile user traces. Extensive experiments show that OFS can effectively support consistent file accesses from computation tasks, no matter whether they are on a mobile device or offloaded to the cloud. In addition, OFS reduce both file access latency and network traffic incurred by file accesses.

Index Terms—Cloud, mobile devices, task offloading, storage, file system, consistency

1 Introduction

Various systems have been designed to allow mobile apps to use cloud resources (e.g., public cloud, personal cloud, or cloudlet) by offloading their resource-demanding tasks to the cloud in the form of threads, objects, or procedures [2], [3], [4], [5], [6]. For example, a mobile app may record video clips on a mobile device, analyze and augment them in the cloud, and then play back the video clips on the mobile device. This leads to a scenario where the computation tasks in the same mobile app can be offloaded to the cloud and/or run concurrently on both the mobile device and the cloud. These tasks work collaboratively and may need to save, read, and overwrite files on both the mobile device and the cloud.

The decomposition and distribution of tasks and their memory states have been studied extensively, and a few programming models, along with the supporting middleware and system infrastructure, have been developed, e.g., Avatar [3], [7], MAUI [5], CloneCloud [6], Sapphire [2], and COMET [4]. However, supporting efficient file access, especially file sharing between the tasks in the same mobile app running on both the mobile device and the cloud remains a challenging issue and has received little attention. Due to this issue, systems such as MAUI and COMET cannot offload tasks in mobile apps if the tasks need to access files.

Existing file systems are not effective in handling remote file access for the offloaded tasks of mobile apps. This seriously limits the capability of mobile systems to freely offload tasks to the cloud. Network file systems and distributed file systems, such as NFS [8] and Dropbox [9], only support remote file access from the platforms where their client software is properly set up. However, setting up the client software usually requires root privilege, which the mobile user may not have. It also needs the credentials of the user to access the file server, which the user may not be willing to release to the cloud. Moreover, if a task is accessing an open file saved in a network/distributed file system, it must reopen the file after the task is offloaded in order to continue accessing the file. This requires that mobile apps must be aware of task offloading, which makes programming cumbersome and error-prone.

Another issue with existing network file systems and distributed file systems is that they cannot satisfy the consistency requirements of cloud-assisted mobile apps at low overhead. For example, to guarantee correct execution, computation tasks concurrently running on the cloud and the mobile device often require strong consistency (i.e., no stale data returned to the tasks). However, most network/distributed file systems, especially those designed for mobile devices

- *The authors are with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102-1982. E-mail: {nrp48, js622, borcea, narain.gehani, reza.curtmola, xiaoning.ding}@njit.edu.*

Manuscript received 18 Feb. 2017; revised 15 Sept. 2017; accepted 2 Oct. 2017. Date of publication 16 Oct. 2017; date of current version 11 Mar. 2020. (Corresponding author: Cristian Borcea.)
Recommended for acceptance by J. Cao.
Digital Object Identifier no. 10.1109/TCC.2017.2763158

2168-7161 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

(e.g., Coda [10], [11]), cannot guarantee such consistency. Some systems even rely on users to manually resolve inconsistencies. The inconsistencies caused by such systems will lead to incorrect results or application crashes. Some other file systems (e.g., NFS) support strong consistency but at high costs of network traffic and energy on the mobiles, and thus are not practical for mobile apps.

To address these problems, we propose an application level file system named *Overlay File System* (OFS). OFS supports remote file access by providing the tasks on the mobile device and the cloud with an efficient, consistent, and transparent view of data that is accessible as local storage. It supports task offloading in the form of threads, objects, or procedures. OFS manages file access and file sharing in a mobile app. It effectively hides the boundary between the mobile device and the cloud, and provides a unified environment for the tasks in the mobile app, such that the tasks can migrate freely between the mobile device and the cloud. By default, OFS ensures that all tasks whether on the mobile or offloaded to the cloud read the latest data in the file. OFS uses an adaptive method named delayed-update, which combines the conventional write-invalidate and write-update policies, to reduce file access latency and network traffic overhead, while ensuring strong consistency. Some applications, e.g., health-monitoring apps, may not require strong consistency. For such applications, OFS also provides a relaxation mechanism that allows applications to use recent but not the latest copies of file data. This can further reduce file access latency and network traffic overhead. To guarantee location transparency, OFS creates a unified view of the data that is independent of location and is accessible as local storage.

Compared to conventional network/distributed file systems, OFS has several advantages for running cloud-assisted mobile apps. First, the strong consistency model ensures the correct execution of computation tasks distributed across the mobile device and the cloud. Second, tasks accessing files can be moved freely across different devices. This is because the states of files and file operations are in the app's user space, and thus can be duplicated and moved with the tasks to new locations. Third, at the application-level, it simplifies application development and system management. For example, with OFS, root privilege is not required to set up the system and there is no need to save the to-be-accessed files into a network/distributed file system before the app runs. Programmers do not have to worry about whether a task is running on the mobile or has been offloaded to the cloud.

The special features of mobile systems and the requirement to run OFS at the application level present a few implementation challenges. For example, most mobile devices are not rooted and applications do not have root privilege. In addition, mobile OSs (e.g., Android) may kill processes and reclaim their memory spaces, making it challenging to maintain OFS system states at the application level. Focusing on these challenges, the paper has studied the implementation techniques and built an OFS prototype on Android. The prototype implements major OFS functionalities into a set of "sticky" application services. An app get OFS services through the code injected by OFS with AspectJ [12].

The paper has also implemented a real app, named *photo enhancement app*, and has used this app and real mobile user traces to test the functionalities and performance of OFS. Our case study with the photo enhancement app shows that OFS can effectively support consistent file accesses from computation tasks, no matter whether they run on a mobile device or has been offloaded to the cloud, and that existing cloud storage systems, including Dropbox and Google Drive, cannot provide such support. The experimental results with the app and real user traces show that the delayed-update policy used in OFS can effectively reduce file access latency by up to 21 percent relative to commonly used write-update and write-invalidate consistency policies. The results also show that, with the delayed-update policy in OFS, the network traffic incurred by file accesses is significantly lower (by up to 67 percent) than that with the write-update policy, and is comparable to that with the write-invalidate policy, which is the lower limit to maintain consistency.

To the best of our knowledge, this is the first work that provides a system solution to support efficient and transparent file access in cloud-assisted mobile apps. We make the following contributions. First, we determine the requirements for a file system to effectively support offloading tasks to the cloud. Second, we design and implement OFS as a solution to meet these requirements. Third, we use a real app and user traces to show that OFS can effectively support task offloading and efficient execution of offloaded tasks by significantly decreasing both file access latency and network traffic incurred by file accesses.

2 Background

This section introduces first several approaches to offload tasks in mobile apps to the cloud. It also presents problem of using network/distributed file systems or cloud storage in the current context. Finally, it summarizes the requirements on file systems for cloud-assisted mobile apps, which underpin the design of OFS.

2.1 Approaches to Offload Computation to the Cloud

To effectively leverage cloud-assistance, a system needs to support task migration between the mobiles and the cloud. A few different methods can migrate tasks, including their code and the required in-memory data sets. Some systems (e.g., Sapphire and Avatar) encapsulate and transfer the code and memory state of a task (e.g., data in heaps) in an object. Other systems (e.g., COMET) offload tasks in the form of threads. They use distributed shared memory (DSM) and transfer the memory state on-demand when it is accessed remotely by the threads. A computation task may also be offloaded by making remote procedure calls (RPC) to the cloud. Since a VM is a complete running environment for an app, from memory state to storage, Cloud-assistance can also be implemented by migrating the VM containing the tasks (e.g., Cloudlet [13]). However, compared to moving a thread/object/procedure, migrating a VM incurs much higher overhead.

In this paper, we target the approaches that offload computation tasks in the form of objects, threads, or procedures. The cost function used by the system to balance the overhead and the benefit of task offloading is beyond the scope of the paper. At the current stage, we assume that there is a cost function that comprehensively considers the overhead

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

of both transferring in-memory data and accessing files remotely for making task offloading decisions.

The tasks in an app run concurrently at the cloud and the mobile device. They often need to access their data sets saved in files. The needs cannot be satisfied by transferring the files to be accessed by a task before offloading the task to the cloud. It is not easy to identify all these files, especially when a task needs to access new files that are generated after it starts. Thus, not all the files can be transferred a priori. More importantly, tasks on the mobile device and the cloud may update and read the same set of files concurrently. This method cannot guarantee the consistency of the shared files. Inconsistency leads to incorrect results or application crashes. For these reasons, systems supporting task offloading (e.g., COMET and MAUI) usually cannot migrate tasks if they need to access files.[1]

This problem can be mitigated by using networked/distributed file systems (e.g., NFS) or cloud storage platforms (e.g., Dropbox). However, existing networked/distributed file systems and cloud storage systems are not designed for collaborative tasks on mobile devices. They are designed for scenarios in which a file is opened, modified, and closed on one device, and then is opened and accessed elsewhere. Concurrent reads and writes on different devices to the same file are not designed or implemented [14]. Thus, their implementations cannot support consistent file access and file sharing with low overhead.

2.2 Requirements on File System Design

To support remote file access and file sharing among the distributed tasks of cloud-assisted mobile apps, a file system should be able to locate and transfer data, and to manage data sharing. To accommodate features of mobile apps and hardware characteristics of mobile devices, a file system must satisfy the following requirements:

- *Location transparency*: The file system should be able to provide an app with access to remote files as though they were local, and should be able to maintain file sessions during the location changes of a task (i.e., task migrations) such that a task does not need to close all its files before migration. In the paper, a file session is defined as the set of file operations between opening and closing a file and the set of states that are managed by the file system to correctly handle the operations. Existing file systems cannot provide enough transparency. For example, a task can only access the files opened on its current device and must re-open the files after it moves to another device.
- *Consistency*: Reading stale data may lead to incorrect results or crash an app. Thus, the file system must guarantee strong consistency by default so that a task always reads the latest updates. However, in the case where an app can tolerate relaxed consistency, the file system should be able to take the opportunity to relax consistency and improve performance.
- *Performance*: Mobile devices have limited resources in terms of energy and network bandwidth. Thus, cloud-assisted apps often need to pay for the network traffic through cellular networks. It is important for the file system to satisfy file access requests with low latency (for higher performance and power efficiency) and little network traffic (for lower monetary cost and energy consumption). Existing networked/distributed file systems are not optimized for cloud-assisted apps.
- *Easy deployment*: To freely offload tasks, a design that can simplify the deployment of the file system and data is highly desirable. Since a mobile user may have limited privileges on the cloud platform accepting offloaded tasks, the deployment of the file system should require minimal privileges in addition to those needed to run the task. At the same time, the file system should have minimal requirements on data deployment. Conventional networked/distributed file systems usually require that files be deployed under specific directories to enable remote access. However, it is challenging, if not impossible, to identify all the files to be accessed remotely by mobile apps and organize them accordingly, since the files to be accessed by mobile apps may be determined by user requests. At the same time, most networked/distributed file systems require root privilege to deploy and to run, which is missing on most mobile devices.

1. The DSM model implemented in COMET can be extended to help accessing memory-mapped files. However, the files must be opened and memory-mapped on the mobile device before tasks are offloaded to the cloud. Opening a file and establishing memory mapping in the cloud require additional system support beyond the DSM mechanism. The DSM model cannot facilitate file access through a standard file I/O interface.

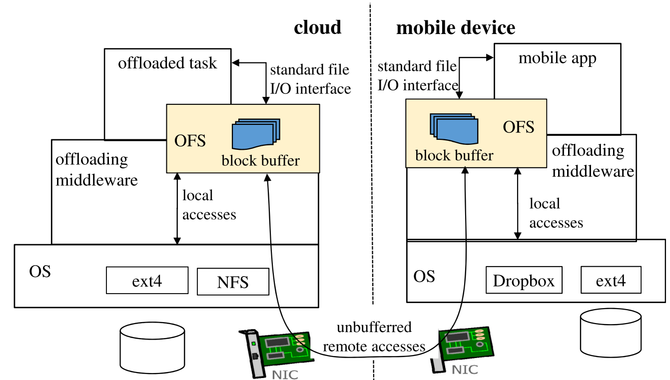


Fig. 1. Overall architecture of offloading ecosystem.

3 OFS Design

3.1 Overall System Architecture

OFS is a component of the system that offloads and manages computation tasks. Fig. 1 illustrates the position of OFS on the mobile device and the cloud platform, and explains how OFS interacts with other components in these platforms. Unlike conventional file systems, which are part of the operating system, OFS functions at the application level. Its code is executed in user mode, and its data structures (e.g., information about the files, file accesses, and the buffer caching file data) are maintained in user space. However, OFS relies on the native file systems in the OS to actually read data from the storage or write data into the storage.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

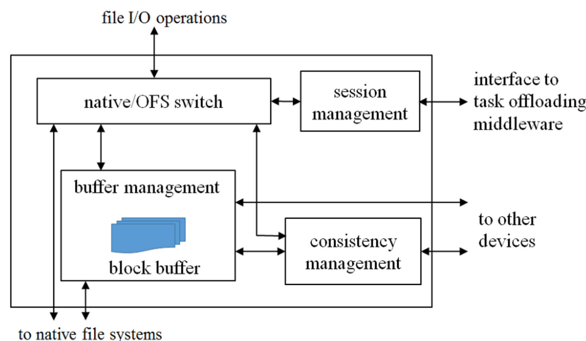


Fig. 2. Overlay File System (OFS) architecture.

There are several reasons for this application-level design. First, OFS is solely designed to provide file accesses for the correct and efficient execution of mobile apps. It does not provide system-wide management, e.g., user access control, or a tree of files and directories presented to the user. It does not manage storage space either. Second, building OFS at the application level makes it an *overlay* file system that sits above all the native file systems, thus allowing it to work with any native file systems through the standard system call interface. Third, keeping all the functionality and data structures within virtual memory spaces at the application level simplifies deployment. For example, there is no need to acquire root privilege to set up the file system. Finally, this design helps to improve efficiency since accessing the data structures and file data cache in virtual memory space does not incur costly kernel-application context switches.

The objective of OFS is to provide efficient, transparent, and consistent file access and file sharing for tasks in a cloud-assisted mobile app. For this purpose, OFS intercepts and monitors the file access requests from the tasks in the app. These requests can be intercepted without modifying existing apps using techniques such as code injection and byte code manipulation. How this is achieved in our OFS prototype will be introduced in Section 4.2. OFS fulfills the requests for accessing local files by passing them to the OS and then to the corresponding native file systems holding the files. For the requests accessing remote files, OFS maintains a buffer named *block buffer* to cache the blocks read from remote files through the network. To fulfill the requests, OFS looks up the *block buffer* and serves the requests if the desired file blocks are cached there. Otherwise, it redirects the unsatisfied requests to the platform storing the files. Note that a file may be stored on the mobile and requested by a task from the cloud or vice versa.

OFS maintains consistency between the blocks in the block buffer and their counterparts saved in remote files, such that a task can always access the latest updates no matter where it runs. To handle other file related requests (e.g., opening files and creating files), OFS forwards these requests to the platform storing the files and updates the related metadata.

3.2 OFS Architecture and Design

Fig. 2 shows the major components of OFS. The *native/OFS switch* intercepts file I/O requests before they reach the OS and decides for each request whether it should be handled by a native file system or by OFS. Generally, OFS handles all the requests to the files that are currently accessed by offloaded tasks, and forwards other requests to native file systems. Thus, in the cloud, all the requests made by offloaded tasks are handled by OFS. On the mobile device, if a file is not currently accessed by offloaded tasks, the accesses to the file should be forwarded to the corresponding native file system; otherwise, they are handled by OFS. To improve performance, read-only files (e.g., libraries) can be distributed on both sides and accessed locally without the intervention of OFS.

The native/OFS switch needs to notify the *consistency manager* about all the accesses before it passes the requests to either a local file system or the buffer management component. When handling a write request, it only proceeds after the consistency manager confirms that the write will not cause inconsistency. When handling a read request, it just notifies the consistency manager, since the access information is needed there to detect access patterns.

The *buffer management* is in charge of managing the block buffer. To look up the buffer, we maintain a mapping table for each file and save the mapping table in the data structure of the file. We also maintain the status of the blocks in the mapping table. Thus, when the file is accessed, OFS can quickly locate the mapping table, from which it determines whether the requested block is buffered, and, if it is, whether the buffered block is up-to-date.

We use an LRU-like algorithm to evict blocks to keep the buffer size within a pre-set limit, which is selected by the user during installation based on the memory capacity of the devices. Due to the high network overhead, it is not cost-effective to offload tasks accessing a large amount of data. Thus, a small size limit (e.g., 1/32 of memory capacity as the default limit) should work well for most of the workloads.

We create the *block buffer* in the virtual address space. This is not only for fast access and ease of deployment, but also to simplify the system design, since the management of the physical space of the buffer (e.g., space allocation/deallocation and swapping) can be done with by the memory management of the operating system. At the same time, it puts the physical memory space occupied by the block buffer under unified management with other system components and apps. This helps the operating system balance system memory usage for the overall benefit of system performance. For space efficiency, the block buffer only caches the content of remote files. It does not buffer the content in local files to avoid double buffering in both the block buffer and the OS buffer cache.

The *session management* component maintains file sessions and prevents them from being interrupted by task migrations. Specifically, when a task is migrated, the session management component is notified. On the destination platform, the session management component must correctly set up the state required by the unfinished file sessions in the task. For example, it must copy file states, such as the current offset in each file and the opening mode of the file, from the source platform.

Though buffering data improves efficiency, it incurs consistency issues. The *consistency management* component provides the consistency guarantee that is required by concurrent programs. For this purpose, it monitors all the accesses to the shared files, as well as the blocks cached in the block buffer. Enforcing consistency usually incurs a

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

large amount of network traffic (e.g., when *write-update* policy is used) or increased read access latency due to increased misses in the buffer (e.g., when *write-invalidate* policy is used). Both long access latency and increased network traffic are not desirable for task offloading in mobile apps. Thus, we use an adaptive algorithm named *delayed-update* combining write-invalidate and write-update (Section 3.3) to reduce both latency and network traffic.

3.3 Consistency Management in OFS

3.3.1 Consistency Management Design Objectives

OFS aims to provide an environment in which the tasks of a mobile app can access and share their files concurrently from both the mobile device and the cloud in the same way as they do when they run on the same device, where they share the OS buffer cache and can always see the latest updates. This will not only guarantee the correct execution of mobile apps, but will also simplify app development, because programmers will not be concerned with getting stale data in apps. Therefore, the first design objective is to ensure strong consistency.

Enforcing strong consistency may incur high overhead. There are two common policies for keeping consistency. *Write-invalidate policy* invalidates all the duplicates of a file block before writing the block locally. *Write-update policy* ensures that a write operation does not complete until all the duplicates are updated. The write-invalidate policy minimizes the amount of data transferred over the network (i.e., network overhead) but increases the latency for read operations because invalidating duplicates reduces the number of local accesses. The write-update policy helps to keep the duplicates valid and, thus, read access latency low, but incurs a large amount of network traffic for broadcasting all updates and high overhead for write accesses. Therefore, the second design objective is to reduce the network traffic incurred by enforcing strong consistency and, at the same time, keep the access latency low.

Strong consistency may not be always desirable. There are situations in which enforcing strong consistency is not necessary or the overhead incurred by enforcing strong consistency is too high. Thus, the third design objective is to satisfy consistency demands other than strong consistency. For example, a health monitoring app collects wellness data of a user every second using the sensors on a mobile device and analyzes the data in the cloud. While the latest data is preferred by the analysis in the cloud, using the data collected a few seconds ago still generates sensible results. If the mobile device is short of resources (e.g., low power level), updating the data lazily is a better choice than enforcing strong consistency.

3.3.2 Delayed-Update Algorithm

To achieve the strong consistency, we design a hybrid approach named *delayed-update*, which combines the write-invalidate and write-update policies. This new policy gives better file latency and reduces network traffic. On a write operation, it invalidates duplicates first to ensure consistency. Then, instead of waiting for a read operation to trigger an update of a duplicate, it predicts when a duplicate is about to be read and it updates this duplicate just before the read. The delayed-update approach reduces network traffic because it does not transfer the updates that have been overwritten before a read. It keeps the access latency low because duplicates are validated and updated before reads. A challenging issue with delayed-update is to predict when the duplicates should be validated and updated. We address this issue by monitoring the file access patterns of of mobile apps, as described later in this section.

In some scenarios, accessing the latest data is not required. For example, in a health-monitoring app, health related data, such as body temperatures and heart rates, is collected and saved periodically. The values of the data may not change rapidly over time. Thus, it may not cause problems if the health-monitoring app uses the data collected recently, e.g., 5 seconds ago. For such scenarios, OFS provides a relaxation mechanism that allows an app to use recent but not the latest copies of file data. The mechanism extends the delayed-update approach with a knob named *relaxation* to relax the requirement on enforcing consistency. Using the same health monitoring app as an example, if the app can use the data generated 5 seconds ago, the relaxation is set to 5. A suitable relaxation value is application-dependent and data-dependent. By default, OFS sets relaxation to 0 in order to enforce strong consistency. In the cases where relaxation can be applied, OFS relies on application developers and users to decide suitable relaxation values and adjusts the values through an API provided by OFS. With a large relaxation value, delayed-update can update duplicates even less frequently to reduce resource consumption.

To reflect the current status of a block, the delayed-update algorithm keeps the following information on both the mobile device and the cloud, for each block that has been accessed by the app.

- A *shared* flag indicates if there are duplicates of the block cached in block buffers or saved in storage.
- A *valid* flag indicates if the block content is up-to-date.
- For each valid block, we also attach an *expiration time* to implement the relaxation feature. A valid block with a non-zero expiration time indicates that the block content is not up-to-date, but can still be used by the app until the expiration time. The block is invalidated when the expiration time is reached.
- The *location* of the latest update.
- An *overwritten threshold* indicates when remote duplicates should be updated.
- An *overwritten counter* counts how many times a block has been overwritten.

When a block is being read, its content is returned immediately if the block is valid; otherwise, the latest update is fetched remotely, and the status of the block is updated to valid and shared. When a block is being written, the block is updated immediately if it is not shared; otherwise, a message is sent to invalidate the duplicates before the block is updated and the "shared" flag is reset. When such an invalidation message is received on either the mobile device or the cloud, the corresponding block is invalidated (when the relaxation is zero) or marked with an expiration time (when the relaxation is greater than zero); at the same time, the location of the latest update is recorded in the mapping table maintained by the buffer management component.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

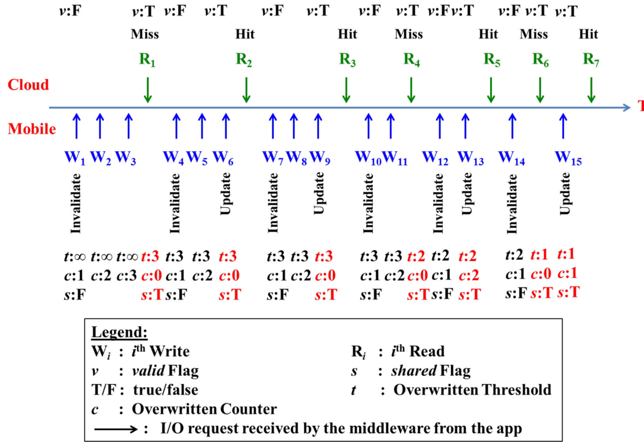


Fig. 3. Workflow of delayed-update algorithm.

The delayed-update algorithm tries to update remote duplicates when they are about to be read. To achieve this goal, the algorithm updates and uses the overwritten threshold as an indicator. When the number of block overwrites reaches this threshold, the remote duplicates are updated. The threshold is dynamically updated based on the history of accesses. Specifically, every time when a block is overwritten, the overwritten counter is incremented. When the content updated in the block is accessed somewhere else (i.e., the platform other than the one generating the content), the overwritten threshold is updated with the value of the overwritten counter, and the overwritten counter is reset. Thus, the threshold reflects how many times a block is overwritten before the content is used, and can be used to predict when remote duplicates should be updated.

In order to describe the basic idea of the delayed-update algorithm, we use an illustrative example with a series of reads ($R_1 \sim R_7$) and writes ($W_1 \sim W_{15}$) on the same block, as shown in Fig. 3. Writes are on the mobile device, and reads are in the cloud. The states of the valid flag, shared flag, overwritten threshold, and overwritten counter used in the algorithm are marked with v, s, t, and c in the figure.

When the block is being written for the first time on the mobile device (W_1 in Fig. 3), the shared flag shows that it has a duplicated copy, thus an invalidation message is sent to the cloud to invalidate the copy. On receiving the message, OFS in the cloud sets the valid flag to $false$ and acknowledges the message. On receiving the acknowledgement, OFS in mobile device sets the shared flag to $false$. Subsequent updates to the block, W_2 and W_3, can be performed directly since there is no duplicated copy. When the cloud tries to read the block, it checks the valid flag first. If the block is invalid (e.g., R_1 in Fig. 3), a miss occurs and the block is propagated. Thus, the shared flag on the mobile device is changed to $true$, and further updates (W_4) will result in an invalidation message. Until now, the algorithm performs exactly as a write-invalidation algorithm, except that the algorithm maintains an overwritten counter and an overwritten threshold for the block on each side (c and t in the figure for the mobile device). The counter is reset every time the block is propagated (e.g., R_1), and incremented every time the block is overwritten. The value of the overwritten counter is saved into the overwritten threshold before it is reset (e.g., the change of the t value corresponding to R_1). With more updates performed on the block (W_5 and W_6), the overwritten counter keeps increasing. When the value of the overwritten counter reaches the value of the overwritten threshold (3 when W_6 is performed), the mobile device propagates the new content in the block to the cloud before a read is issued in the cloud (R_2). This reduces the latency. This part shares a similar idea with the write-update algorithm. However, it only performs updates when it predicts that the updates are necessary. The prediction relies on the program maintaining a regular access pattern (e.g., the time period from W_1 to R_3). Misprediction occurs when the program changes its access pattern (e.g., W_{10}, W_{11}, and R_4). However, the algorithm can quickly adapt and adjust the prediction based on the new pattern, as it does for W_{12}, W_{13}, and R_5.

4 OFS Implementation

OFS sits between mobile apps and the offloading middleware and it is implemented at the application level rather than the OS level. This presents several challenges to the implementation, including the lack of root privilege and state loss when application is killed due to the short of resource. This section introduces the implementation details of OFS, particularly how these challenges are addressed.

4.1 Implementation Details

We have implemented an OFS prototype with Java on Android. Though the implementation is Android-based, the techniques used are generic and can be adapted to implement OFS on other mobile OSs.

At the application level, OFS can be implemented in two ways: as a library that is dynamically linked into each app, or as a set of services, which are independent threads running in the background without interaction with users. With the library implementation, the OFS code, system states, and block buffer are in the memory space of each app. Thus, the app can directly access OFS functionalities and data with high efficiency. However, this implementation incurs consistency issues, since mobile OSs, such as Android and iOS, may kill an app and reclaim its memory space when it is switched to the background. Inconsistency is caused if there are unsynchronized OFS system states or file data in the memory space, which are lost when the memory space is reclaimed. The other issue is that the library implementation does not support file sharing between apps.

We choose to implement OFS into a set of application services (named *OFS middleware*) and keep application-specific states inside each app. The services start automatically when the system is on. They are marked as "sticky" services, so that they are less likely to be killed by the OS than normal application threads and other services. In some rare cases when these "sticky" services are killed by the OS, they will be restarted automatically by the OS, at a later time, before other services and apps. OFS has a simple checkpointing mechanism implemented to back-up OFS system states into storage before the services are killed. The checkpointing mechanism can also be used to handle network disconnection problems of offloaded tasks. If an offloaded task was disconnected due to network issues, OFS can roll back to the state before the task was offloaded.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

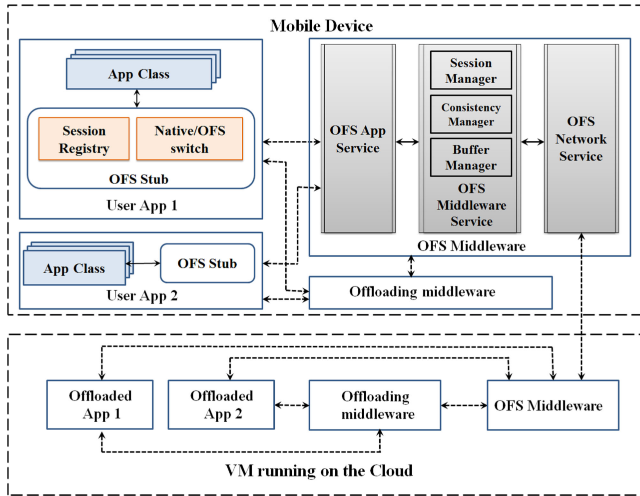


Fig. 4. Architecture of OFS implementation.

To facilitate the accesses to OFS services, we provide a component, named *OFS stub*, which is linked into each app process as the interface between the app process and OFS.

Fig. 4 shows the architecture of the implementation, which has two layers. The upper layer, named OFS stub, is mainly in charge of intercepting file I/O requests, maintaining app-specific information, and interacting with OFS middleware to satisfy file I/O requests. It consists of two major components, the native/OFS switch and the session registry. The native/OFS switch is as introduced in Section 3. The session registry, as a part of session management in Fig. 2, is in charge of maintaining file sessions by managing and updating the data structures used by the app for accessing open files, such as status of the file (location, open mode, etc), current offset, length and so on.

For the tasks offloaded into the cloud, the session registry provides the data structures for accessing files. For the tasks on the mobile device, the session registry mainly serves as a registration list of all the files that are currently accessed by offloaded tasks. The list is used by the native/OFS switch on the mobile device to filter requests.[2] The session registries in the cloud and on the mobile device are updated consistently when a file is accessed by an offloaded task and the information of the file cannot be found in the session registries. Specifically, before a task is offloaded, the session registry on the mobile device is empty, and thus all the file accesses of the app are handled by native file systems on the mobile device. Later, when a task is offloaded into the cloud and the task starts to access a file, the session registry in the cloud is searched. Since the required data structure for accessing the file cannot be found there, the access cannot proceed before the data structures are set up and registered as a file session. To set up and register the data structures, the OFS stub in the cloud generates a reopening request, which is forwarded to the OFS stub in the mobile device. On receiving the re-opening request, the OFS stub in the mobile device registers the file in its session registry. In this way, the file is marked as being accessed by an offloaded task, and later accesses to the file are forwarded to OFS by the native/OFS switch. Then, the OFS stub in the mobile device sends back the information required for accessing the file (e.g., file offset and open mode) to the OFS stub in the cloud, which then uses the information to update the session registry in the cloud. With the information, later accesses can be handled by OFS.

2. The native/OFS switch in the cloud determines that all the file accesses should be handled by OFS, except for the accesses to the files pre-configured to be accessed locally (e.g., read-only library files).

Using Filesystem in Userspace (FUSE) [15] may simplify the implementation. However, FUSE requires root permission and rooted systems. Android needs to be recompiled in order to implement OFS in the existing FUSE daemon. Thus, rather than using FUSE, we implemented the OFS middleware using app services, which run on both mobile device and the cloud. The main service, *middleware service*, implements the other three major components of OFS described in Section 3.2. Two supporting services assist the main service to interact with other system components. Specifically, the *app service* interacts with apps to receive requests and deliver responses; and the *network service* maintains the interaction between the OFS instances running in the cloud component and the mobile device.

In OFS, a large amount of data may be exchanged over network or locally across different OFS components, and some messages (e.g., events and updates on staled data) must be processed promptly. Thus, OFS must handle data communication with high efficiency. For network communication, we adopted a NIO-based TCP library named Kryonet [16], which is usually used by online games for high network throughput and low latency. For local communication, we used Android's Binder IPC mechanism. BroadcastReceiver mechanism is not used since it may reduce the communication throughput between the OFS stub and the OFS middleware by up to 3x based on our experiments.

We used the offloading service of the Avatar platform [17] as the offloading platform for our implementation. Avatar is a distributed mobile-cloud platform where each mobile device has a surrogate in the cloud. It also supports offloading Plain Old Java Objects (POJO) to the cloud. POJO is a software engineering term used to describe a Java object not bound by any special restriction or external class path. As the Avatar platform supports multi-threading programming, offloading an object only blocks the relevant threads in the mobile device instead of all of them. Unlike a regular offloading platform, offloading in Avatar aims to improve battery consumption, network bandwidth and latency for a group of users. It uses annotations to intercept the targeted code segment and uses a profiler to decide whether to offload based on QoS defined by the targeted user group. For the experiments conducted in Section 6, we hardcoded which operations are being offloaded to the cloud in order to ensure the intended task is always offloaded.

4.2 Implementation Challenges and Solutions

To implement OFS in userspace, we solve several issues. One issue is how OFS can interact with different apps to intercept their file I/O requests and satisfy them. To address this issue, our implementation intercepts library calls, instead of system calls. The interception of library calls does not require a system-level privilege and can be implemented with various approaches, e.g., manipulating symbol tables or binary weaving [12], [18]. Our current prototype

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

uses AspectJ [12] in the OFS stub to automatically link the required code to interact with the existing code of the app without additional effort from the app programmer. In this way, an app can be automatically enhanced with OFS support; and the app developers do not need to be aware of task offloading or implement the code that handles file I/O issues for offloaded tasks.

Specifically, the method interception mechanism in AspectJ is used to capture the calls related with file I/O requests. Then, the code to analyze the requests and to call the methods in OFS stub is injected using the weaving mechanism. While the capturing of file I/O requests and the injection of OFS code can be performed when the app is compiled or after the app is compiled (e.g., when the app is being loaded), the current prototype finishes this process at compile time to minimize runtime overhead.[3]

Another issue with a user-level implementation on Android is how to manage the accesses to app-private files. In android, an app can have two types of files. Private files are saved in the internal (private) storage space, and are only accessible from the apps that created the files. Public files are saved in the external (public) storage space, and can be accessed by any apps. Since OFS middleware runs as application services and cannot access private files of any app, if an offloaded task needs to access a private file saved on the mobile device, the accesses to the private file are forwarded back from OFS middleware to the OFS stub in the corresponding app and performed by the OFS stub. OFS does not buffer the data in private files. This rarely degrades performance, since private files are usually small files, such as settings, configurations, and cached data, and are infrequently accessed.

4.3 Interface with Task Offloading Systems

OFS must work synergistically with task offloading systems. However, it is challenging for an OFS implementation to be compatible to different offloading systems, which may be implemented in different ways at different system levels. As explained in Section 2, computation tasks may be offloaded in the form of objects, threads, or procedures. These different methods correspond to different ways of system implementation. If OFS is built inside an offloading system as a component, different OFS implementations are needed for different task offloading systems.

To increase compatibility and reduce development efforts, we decouple the implementation from specific task offloading systems, and keep a narrow interface between OFS and task offloading systems. With our implementation, the middleware service and the OFS stubs do not interact directly with task offloading systems. A simple utility, named *offloading service*, is developed to accept notifications from task offloading middlewares. The offloading service is notified when a cloud-assisted app is launched, when there is a task newly offloaded to the cloud, or when an offloaded task is about to be migrated back to the mobile device. Based on the notification, the offloading service instructs the OFS middleware to update system states and the related app threads to update application-specific states.

3. An alternate approach that does not need recompilation is to interpose the library function call paths. This can be done by instrumenting the binaries of the app with tools such as PIN [19] or ProbeDroid [20].

For example, when an object is migrated into the cloud by the Avatar offloading platform, the OFS offloading service in the cloud is notified about the migration with information, such as the ID of the offloaded object. The offloading service contacts the application thread responsible for the offloaded object in the cloud, such that the injected OFS code in the thread can re-establish existing file sessions by re-opening files and moving file pointers. Then, it notifies the OFS middleware about the offloaded object, such that subsequent file I/O requests from the offloaded thread can be serviced by the OFS middleware. Such interactions induce a one-time overhead which is included in the performance results presented in Section 6.

5 Case Study with a Real App

We implemented a photo enhancement app as a case study. It illustrates the demand for transparently supporting file accesses of cloud-assisted apps, and demonstrates the advantages of OFS. With the app, we explain how our OFS implementation efficiently supports the file accesses of the tasks distributed across the mobile device and the cloud.

The photo enhancement app processes the photos selected by the user. For each photo, it performs a few photo enhancement operations, including applying color reduction, adding salt noise, applying sharpening filters, and adding a watermark. The app displays the photo to the user after the operations. We implemented each photo enhancement operation in a Java class using OpenCV [21].

Based on the same source code implementation, we have built three versions of the app: 1) a conventional mobile app named *PE-Mobile* that executes all the operations locally on the mobile device, including the enhancement operations, 2) a cloud-assisted app named *PE-Offload* that can offload photo enhancement operations to the cloud and access the photos using a cloud storage system, and 3) a cloud-assisted app named *PE-OFS* that can offload photo enhancement operations to the cloud and access the photos using OFS.

For fair comparison, we hard-coded the task offloading part in the app to offload all the photo enhancement operations to the VM. We did not link *PE-Offload* with OFS stub, in order to test whether a cloud storage system (e.g., Dropbox or Google Drive) can be used to support the file accesses of the app. The last version, *PE-OFS*, was built with OFS support. Compared to *PE-Offload*, the enhancement with the OFS support in *PE-OFS* only requires the linking of OFS supporting library with the app, and does not incur additional efforts on programming or annotation.

Before running the app, we deployed the OFS middleware on a mobile device and an Android-x86 VM (detailed configuration in Section 6). Though OFS can be distributed and deployed through app stores, such as Google Play Store, currently the middleware is packed in Android application packages. Thus, we copied the packages (the apk files) into the mobile device and the virtual machine, and side-loaded the packages. Root privilege was not requested during the installation. However, we performed some simple configuration before *PE-OFS* could run: pair the mobile device and the VM, and allow access to library files.

We first run *PE-Mobile* to process a set of photos with different sizes to verify the functionalities of the app. Then, we

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

run PE-Offload to process the same set of photos. We want to justify the necessity to design a system to transparently support the file accesses of a cloud-assisted application. To make the photos accessible to the tasks offloaded to the VM, we installed the Dropbox client app on both the mobile device and the VM. Before the execution, we must first upload the photos into a Dropbox directory on the mobile device and mark them available for offline accesses in order to download them into the device. Only after the downloading is finished, can we launch *PE-Offload*. Even though the photos were accessible from the VM, we found that the photo enhancement tasks offloaded to the VM crashed during execution. This is because these tasks access each photo using the file handle created on the mobile device when the photo file is opened before any enhancement operations start, and the file handle is invalid in the VM. To solve the problem, we have to change the source code of the app, such that a photo must be re-opened before each enhancement operation and closed after the operation. With this improvement, the enhancement operations can be finished on the VM without crashing. But we find that the app may display the old versions of the photos on the mobiles, despite the fact that newer versions with enhancements exist in the cloud. This is caused by Dropbox being unable to promptly update the copies on the mobile device. Thus, we have to re-examine the photos after both the Dropbox instance on the VM and the instance on the mobile device finish the synchronization with Dropbox server. We have also tested PE-Offload by saving the photos into a Google Drive and experienced similar problems.

Despite the increased management and programming efforts, with existing cloud storage systems, a cloud-assisted program still may not be able to deliver correct results. This clearly shows that existing cloud storage systems cannot meet the requirements of cloud-assisted apps and a system must be designed to support the file accesses of these apps transparently and consistently.

We have also tested *PE-OFS* with the same set of photos. We run *PE-OFS* for two times. We first run *PE-OFS* completely on the mobile device without offloading any enhancement operations. Then we run it with the enhancement operations offloaded to the VM. With *PE-OFS*, the photos can be enhanced and correctly displayed after the enhancements no matter whether the enhancement operations are offloaded to the cloud or not. When the enhancement operations are performed on the mobile device, *PE-OFS* shows similar performance as *PE-Mobile*. The end-to-end latency for the enhancement operations on each photo is less than 0.6 percent higher than *PE-Mobile*. When the enhancement operations are offloaded to the cloud, compared to *PE-Mobile*, the end-to-end latency is reduced by 43 percent on average with *PE-OFS*, and the combined energy consumption of both the app and OFS middleware running on the mobile device is reduced by on average 3 percent.

The above experiments show that OFS has low overhead and can effectively support the seamless execution of cloud-assisted apps on the mobile device and in the cloud. We will present the detailed performance results in Section 6. In this section, we focus on explaining how OFS transparently supports the consistent file accesses of *PE-OFS* on both mobile device and in the cloud.

6 Performance Evaluation

This section assesses the performance of OFS and evaluates its delayed-update consistency policy by comparing the performance with other consistency policies. We use the following metrics: 1) *Execution time* and *average file I/O latency* measure the performance of OFS and comparison solutions. 2) *Network overhead* quantifies the network overhead introduced by each solution. It practically represents the cost of achieving lower I/O latency. 3) *Number of overwrites per data transfer* measures how many overwrites are performed on a file block until it is transferred. Practically, it helps us estimate the benefits of delayed-update policy. The higher the values of this metric, the more reduction in network overhead. 4) *Power consumption* quantifies the power consumed by the OFS middleware and the app using OFS.

6.1 Experiment Setup

The experiments were conducted on a Nexus 6 mobile phone running Android 7 and a x86 VM running Android 6. The VM was hosted on an OpenStack-based cloud. It has 2 virtual CPUs, 3 GB memory, and 16 GB storage. The physical machine hosting the VM has an Intel Xeon (E5-2630) processor, 78 GB memory, and 2TB storage. We installed the OFS middleware on both the mobile phone and the VM. In the middleware, in addition to the delayed-update policy, we also implemented the write-invalidate and the write-update policies, which can replace the default delayed-update policy through OFS configuration. For our experiments, we set the block buffer size to be 64 MB. The replacement algorithm is run when block buffer is full and new data needs to be saved. With this size, hit ratios are above 95 percent for all workloads.

We first tested our implementation by running the aforementioned photo enhancement app. We use the app to enhance three sets of photos, 15 photos in each set. The resolutions of these three sets of photos are 2.1, 5.0 and 9.7 megapixel, respectively. For each photo, the app first displays the original photo. Then, it enhances the photo on the VM. When the enhancements finish, it displays the enhanced photo on the phone immediately.

Then, we tested OFS with the traces collected on the PhoneLab testbed [22] from six real mobile users, one trace for each user. The traces include the file I/O system calls captured on Android phones using boinic [23] when the users were actively using these phones for different amounts of time and executing different apps with different I/O patterns. To replay the traces, we first developed an app. The app creates some threads on the phone and some other threads on the VM. These threads read the records of file I/O operations in a trace and perform the corresponding operations on the corresponding files. To support the execution, we created a set of files based on the file names and paths in the traces. The contents in the files are randomly generated, since no computation is carried out on the contents.

To imitate the concurrent execution of the tasks offloaded to the cloud and the tasks on the phone, we divided the file operations in each trace into two sets, and re-played one set of operations with the threads on the phone and the other set with the threads on the VM. We divided the operations in two ways to imitate two different task offloading

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

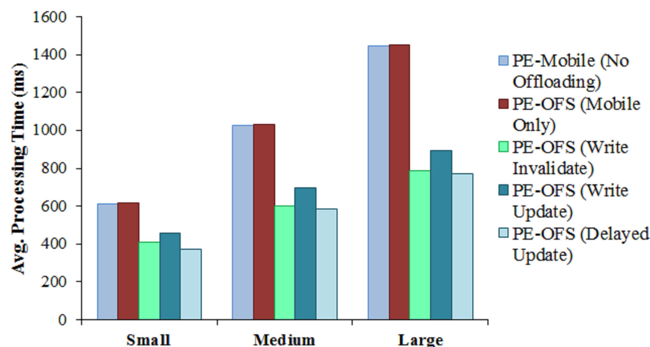
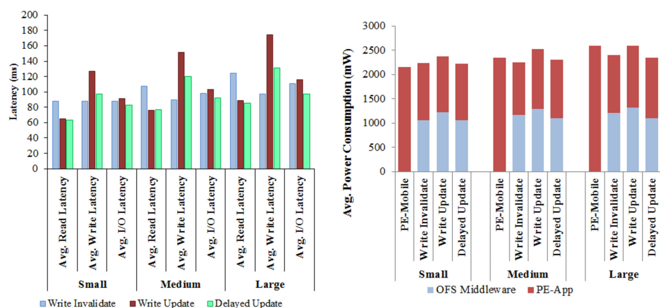


Fig. 5. Average processing time of the photo enhancement app.



(a) latency

(b) power consumption

Fig. 6. Average read latency, average write latency, average I/O, and average power consumption for photo enhancement app.

schemes: 1) thread offloading, and 2) procedure offloading. The traces have the IDs of the threads performing I/O operations. For *thread offloading*, we sorted the threads based on the number of I/O operations performed by them, then divided the threads into two sets, each set of threads having approximately 50 percent of the I/O operations. We replayed the I/O operations of one set of threads in the VM and the rest of the file operations one the phone. In the case of *procedure offloading*, for each thread, we first replayed 30 percent of its file operations on the phone, then replayed 50 percent of its file operations in the cloud, and finally replayed the rest (20 percent) of its file operations on the phone again. The 50 percent file operations replayed in the VM imitate those caused by procedure offloading. Thus, we obtained 12 workloads: one set of six traces for thread offloading, and one set of six traces for procedure offloading.

6.2 Results with Photo Enhancement App

This section evaluates the performance of the photo enhancement app *PE-OFS* we built for the case study (Section 5) to understand the advantages and overhead of OFS.

For each set of photos, we first run *PE-Mobile* on the phone; then we run *PE-OFS* under four scenarios: 1) *PE-OFS (mobile only)*: only on the phone without task offloading, 2) *PE-OFS (write-invalidate)*: on the phone with photo enhancement operations offloaded to the VM and the write-invalidate policy used to maintain consistency, 3) *PE-OFS (write-update)*: on the phone with photo enhancement operations offloaded to the VM and the write-update policy used to maintain consistency, and 4) *PE-OFS (delayed update)*: on the phone with photo enhancement operations offloaded to the VM and the delayed-update policy used to maintain consistency. When *PE-OFS* is launched, the photos are saved on the mobile device only.

Fig. 5 compares the end-to-end processing time for the above scenarios. First, it shows that, when running on the phone without task offloading, *PE-OFS* has similar performance with *PE-Mobile*, indicating the low overhead of OFS. On the VM, photo processing can be finished much faster than on the phone. Based on our measurement, the processing time is reduced by 86 percent on the VM on average for all the photos than on the phone. Therefore, despite that large network latency can offset the benefits of task offloading, when *PE-OFS* run on the phone with photo processing tasks offloaded to the VM, the average processing time with *PE-OFS* is still shorter than that with *PE-Mobile* by at least 31 percent. As shown in the figure, the performance advantage of *PE-OFS* is more prominent with larger photos. As shown with the last three bars in each group, for *PE-OFS* the average processing time is the longest with the write-update policy, and is the shortest with the delayed-update policy. Compared to the delayed update policy, the average processing times with write-invalidate and write-update policies are 5 and 20 percent higher than that with delayed-update.

To better understand the performance difference, we measure latencies of file read operations and the latencies of file write operations, and show the average latencies of reads, writes, and all the file operations in Fig. 6a. Generally, average latencies are higher with bigger photos than with smaller photos, because the app reads/writes a whole photo in one I/O operation. As shown in the figure, among three policies, the write-invalidate policy incurs the highest average read latency due to the long latencies caused by reading invalidated duplicates; and the write-update policy incurs the highest write latency, since it must update all the duplicates on each write. The delayed-update policy updates duplicates only when they are predicted to be read soon. Compared to write-update, the average write latency with delayed-update is 23 percent lower, since it does not need to update duplicate on every write with delayed-update. Compared to write-invalidate, the average read latency is 29 percent lower with delayed-update, which may have validate duplicates before they are read. On average, the average latency of file I/O operations are 8 and 12 percent lower with delayed-update policy than with write-invalidate and write-update policies, respectively.

The last experiment measures the power consumption (i.e., energy consumed every second) incurred by the app using Trepn Power Profiler [24]. For *PE-OFS*, the power consumption consists of two parts: the power consumption of the app itself and the power consumption of OFS middleware. Fig. 6b shows the average power consumption. From the figure, it is clear that with the increase of image size, the average power consumption increases. For small photos, the average power consumption with *PE-Mobile* is lower than that with *PE-OFS*. For medium and large photos, the average power consumption with *PE-Mobile* is higher than that with *PE-OFS* if write-invalidate or delayed update policy is used. When write-update is used, due to the large amount of energy consumed for updating duplicates frequently, the power consumption with *PE-OFS* is higher than *PE-Mobile*. With similar or lower power consumption to *PE-Mobile*, the

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

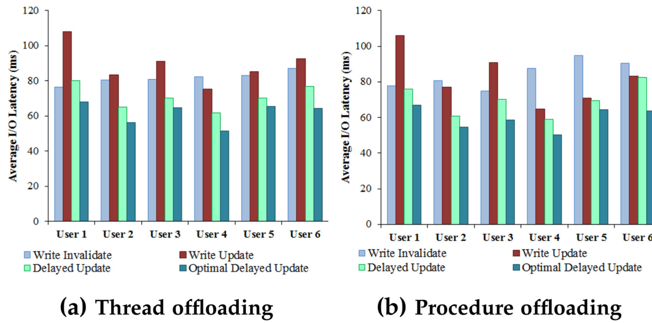


Fig. 7. Average I/O latency for six mobile users.

reduced processing time with *PE-OFS* is also translated into reduced energy consumption, particularly when the consistency policy used in OFS is properly chosen.

6.3 Results with the Real Mobile User Traces

This section tests the performance of OFS using the file operations in the traces. We compare the performance of delayed-update policy with three alternative consistency policies: write invalidate, write update, and optimal delayed-update. The original delayed-update policy relies on the prediction of future accesses to make decision on whether remote duplicates should be updated. The optimal delayed-update policy can be considered as an improved delayed-update policy, in which the prediction is 100 percent correct, with the knowledge on the file accesses issued in the future. Though it is not possible to make 100 percent correct prediction in practice, by comparing the performance between the delayed update policy and the optimal delayed-update policy, we can estimate the potential to further improve the delayed update policy. We implemented the optimal delayed-update policy by modifying the OFS middleware to accept the hints passed from the trace-replaying app.

Fig. 7 compares the average latency of file I/O operations with these policies for thread offloading and procedure offloading. The I/O latency mainly consists of network latency, the time to access the local storage, and the time spent on IPC between the user app and the OFS middleware. As shown in the figure, the average I/O latency with delayed-update policy is lower than that with the write-update policy across all the workloads. Compared to the write-update policy, with the delayed-update policy, OFS can reduce I/O latency by 3∼28 percent for different workloads (21 percent on average). The delayed-update policy also incurs lower I/O latency than the write-invalidate policy for these workloads (6∼33 percent lower), except for the trace of user 1 in the thread offloading scenario (4 percent higher). Compared to the delayed-update policy, with the optimal delayed-update policy, the average I/O latency can be reduced by 7∼24 percent. This shows that the performance of delayed-update policy can be further improved if sophisticated algorithms can be used to improve prediction accuracy.

The results also show that, in general, procedure offloading benefits more from OFS than thread offloading. This is because in procedure offloading, a bulk of I/O operations are migrated to the cloud together, where in thread offloading, different threads can run on mobile device and cloud simultaneously while accessing same file. This causes thread offloading to be more expensive in order to maintain consistency. Therefore, we conclude that offloading systems should implement procedure offloading in order to take full advantage of OFS.

To gain further insights into the behavior of OFS, Fig. 8 shows the average latency for read operations and write operations for the two sets of workloads. As expected, write-update achieves the lowest read latency and the highest write latency due to its design of updating blocks for every write, while write-invalidate achieves the lowest write latency and the highest read latency due to its design of updating blocks upon read operations. The optimal delayed-update policy combines the advantage of write-update on low read latency and the advantage of write-invalidate on low write latency, with read latency and write latency close to those of write-update and write-invalidate respectively. Though the delayed-update policy cannot achieve such low latency limited by its prediction accuracy, it balances read latency and write latency well to improve overall performance. For these workloads, compared to write-update, on average it reduces write latency by 34 percent, at the cost of 18 percent higher read latency; compared to write-invalidate, on average it reduces read latency by 38 percent, at the cost of slightly increased write latency (11 percent higher). Relative to the optimal delayed-update policy, the write latency with the current delay-update policy is 7 percent higher and the read latency is 19 percent higher, indicating that the delayed-update policy tends to over-predict the arrival time of read operations.

We also measured the amount of network traffic with the two sets of workloads. Fig. 9 shows that, as expected, the write-update policy incurs the most network traffic, while the write-invalidate policy incurs the least. Generally, the network traffic incurred by OFS (the delayed-update policy) is much less than that of write-update (67 percent less on average), and only slightly higher (3∼14 percent)

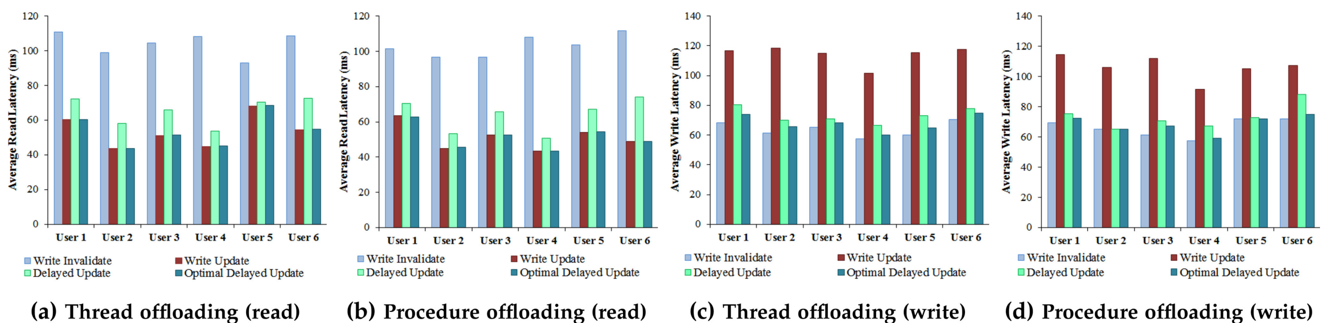


Fig. 8. Average latency of read operations and write operations.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

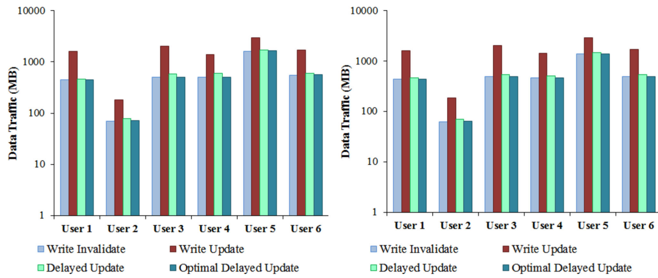
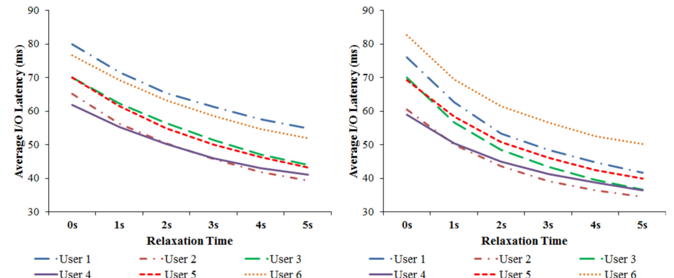


Fig. 9. The amount of network overhead incurred by the workloads.



(a) Thread offloading (b) Procedure offloading

Fig. 11. Average I/O latency when the value of relaxation time is increased from 0 to 5 sec.

than that of write-invalidate. Note that, with write-invalidate, updates are transferred only when they must be propagated to satisfy the requests for data. Thus, the network overhead can hardly be further reduced. This is mirrored by the network traffic incurred by the optimal delayed-update policy, which is also slightly higher than that with the write-invalidate policy by 1.2 percent. Let us also note that similar to the results for file I/O latency, procedure offloading leads to lower network overhead.

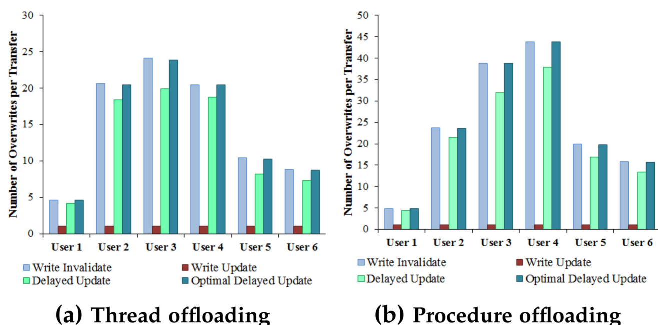
The results with file I/O latency and network traffic clearly demonstrate the advantages of OFS. It reduces the file I/O latency substantially compared to the write-invalidate policy, while maintaining a similar network overhead. The write-update policy performs poorly in terms of both average file I/O latency and network overhead.

To gain insights into the factors that lead to the OFS benefits, we collected the number of overwrites per transferred data block. As shown in Fig. 10, a block may be overwritten multiple times before it is transferred. This is the reason why the policies except for write-update can effectively reduce the latency of write operations (Fig. 8) and network overhead (Fig. 9). The figure also shows that with OFS the average number of overwrites per transfer (4∼37 times across different users) is only slightly lower that with the write-invalidate policy and the optimal delayed-update (5∼43 times across different users). This explains why the latency and network traffic of write operations with OFS is slightly higher to that with the write-invalidate and the optimal delayed-update policy (Figs. 8 and 9).

Fig. 10 also shows that the number of overwrites is significantly higher for procedure offloading than for thread offloading. This result explains why procedure offloading performs better than thread offloading in terms of write latency and network overhead.

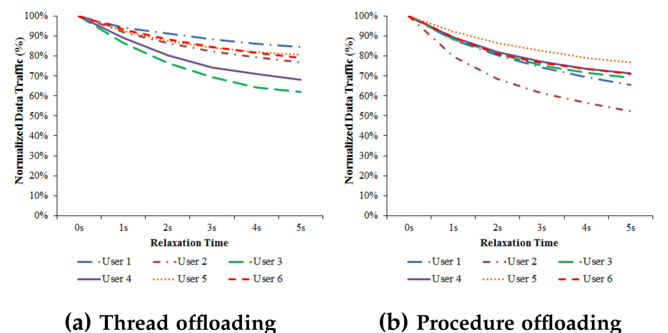
To understand the impact of relaxation time on performance, we changed the length of relaxation time from 0 seconds (i.e., regular delayed-update with no relaxation) to 5 seconds, and measured the average I/O latency and total network overhead. Figs. 11 and 12 show the decrease of I/O latency and network overhead with relaxation time for different workloads. Since the network overhead varies significantly for different users, we normalized the overhead to the one without relaxation for each user, and show in Fig. 12 the normalized network overhead. When the relaxation time is increased to 5 seconds, the average I/O latency is reduced by 36 percent on average for the traces of the 6 users with thread offloading and 43 percent on average with procedure offloading; the amount of network overhead is reduced by 25 percent on average with thread offloading and 32 percent on average with procedure offloading. The average I/O latency is reduced because the overhead associated with the latest update to each block across the Internet is amortized by a larger number of read operations before the relaxation time expires. The amount of network overhead is reduced because multiple updates to the same file block on a device can be consolidated and propagated together with one network transfer when the relaxation time expires.

The figures also show that, with the increase of relaxation time, though average I/O latency and the amount of network overhead keep getting reduced, the reduction becomes less prominent. The reasons are as follows. With the increase of relaxation time, the cost of propagating an update is amortized by an increasingly larger number of read operations, and thus the benefit of amortization diminishes. At the same time, there are a limited number of updates to the same file block in a period; thus, the number of updates that can be consolidated before the relaxation time expires may not keep increasing.



(a) Thread offloading (b) Procedure offloading

Fig. 10. The average number of overwrites per data transfer.



(a) Thread offloading (b) Procedure offloading

Fig. 12. Normalized network overhead incurred when the value of relaxation time is increased from 0 to 5 sec.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

7 Related Work

OFS is an easily deployable file system that supports seamless, transparent, consistent file I/O of mobile apps with concurrent tasks running on mobile devices and in the cloud. This section first presents methods used by existing mobile-to-cloud offloading systems to handle file I/O. Then it compares existing distributed and network file systems with OFS. Finally, the section discusses existing consistency policies.

7.1 File I/O in Existing Cloud Offloading Systems

A few systems that offload computation from the mobile to the cloud have been developed [2], [4], [5], [6], [13], [25], [26]. However, none of them is able to handle the file I/O of offloaded tasks efficiently, if at all. Some of them, such as MAUI [5], and ThinkAir [25] assume that the to-be-accessed files are already available in the cloud when tasks are migrated. They do not have mechanisms to support consistent remote file accesses. On the other hand, the offloading tool for Android applications based on autonomous method selection [26] does not offload methods with file I/O. It should be noted that all of these offloading systems, like OFS, work at user-level.

CloneCloud [6] migrates threads in application-level VMs. It supports access to local files. But, accessing and updating the same file from both the mobile device and the cloud simultaneously is not supported. COMET [4] provides distributed shared memory support for migrating threads between mobile devices and cloud. However, it does not support offloading threads that perform file operations. Sapphire [2] is a distributed programming platform for developing and deploying apps spanning mobile devices and clouds. Tasks are distributed using Sapphire Objects (SO) that encapsulate both data and code. Sapphire SOs may access remote files with a simple RPC-based mechanism. But the design lacks transparency and efficiency. For example, SOs accessing files cannot move, and all the file accesses have to go through network. Just-in-time (JIT) provisioning in cloudlets [13] uses a synthesis server to help prepare virtual disks for the tasks offloaded to cloudlets. Since the files to be accessed by the tasks are included in the virtual disks, JIT provisioning and cloudlets can satisfy file I/O requests of offloaded tasks. This design is for VM-based task offloading, which usually incurs a high overhead. OFS targets offloading tasks in the context of threads, objects, or procedures.

7.2 Distributed and Network File Systems

Various distributed and network file systems were developed for different purposes [8], [10], [11], [27], [28], [29], [30], [31]. Most distributed and network file systems (e.g., NFS [8], AFS [32], Coda [10], [11], and BlueFS [31]) are for users accessing their files from different devices or sharing files. Some of them (e.g., Coda and BlueFS) target mobile users and take into consideration the characteristics of mobile devices (e.g., limited resources and network connection). OFS is designed mainly to support the file accesses of the tasks offloaded to the cloud from mobile devices.

OFS differs from existing distributed and network file systems from the following perspectives. First, conventional distributed and network file systems usually require that the client software be installed and configured before they can access files, making them cumbersome to use in task-offloading scenarios. OFS works at the application level and can be established on demand when a task in an app is offloaded to the cloud. Second, unlike OFS, conventional distributed and network file systems do not provide support for tasks that have opened files at the time of offloading. Last but not least, OFS supports efficient and consistent file sharing in task-offloading scenarios, as we will explain in detail in the next subsection.

7.3 Consistency Policies

Different policies are adopted in distributed and network file systems to enforce consistency. For example, Coda [10], [11] supports disconnected operations, which allow users to update files when network is disconnected. However, this leads to consistency issues that need to be solved by users. BlueFS [31] cannot avoid conflicts either, and it requires users to manually resolve conflicts. This is not practical for mobile apps that offload tasks to the cloud—any benefits in performance will be lost if the users are asked to help solve consistency issues through conflict resolution.

NFS [8] supports close-to-open consistency. To guarantee file consistency, applications need to use either file locks or shared reservations to avoid interleaving file sessions. This model does not fit task-offloading scenarios, where tasks running in parallel at the mobile and the cloud may need to update/read a file concurrently.

Mobile File System (MFS) [30] is a cache manager for adapting data accesses in collaborative applications to network variability when they access a distributed file system. MFS supports consistent accesses to shared files. But the consistency scheme is designed to target network bandwidth variation and network latency is not a major concern. The scheme may cause high file I/O latency, which is not desirable in task-offloading scenarios.

Raindrop File System (RFS) [27] aims at mobile devices accessing files saved in cloud. It implements a client-centric management scheme, in which clients decide synchronization points to manage consistency. However, how to select appropriate synchronization points is a challenging and unsolved problem. When used in task-offloading scenarios, RFS increases the difficulty of programming and cannot guarantee the required file consistency.

Simba [28], [29] provides a reliable and consistent synchronization service for mobile devices. With Simba, mobile apps can always see a consistent view of their data, and the data can be stored locally on the mobile device, in the cloud, and/or on other mobile devices. In addition to calling Simba API to access/update data, it is also the app's responsibility to call Simba API to register data, synchronize updates, and resolve conflicts. OFS, on the other hand, does not require apps to handle these operations, and can be used when apps do not have offloading logic.

8 Conclusion and Future Work

Research described in this paper has been driven by the demand for offloading mobile app tasks to the cloud. The paper has identified one major obstacle to satisfying this demand, namely the lack of effective support to allow the offloaded tasks to access and share files with the rest of the

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

app on the mobile device. To remove this obstacle, we have presented and implemented an overlay file system (OFS), which provides efficient, consistent, and location transparent access to files in a mobile cloud environment where app tasks could be executed at either platform. The experimental results based on real app and real mobile user traces have demonstrated that OFS can effectively support consistent file accesses from both the mobile device and the cloud and achieve substantially lower file access latency than competing methods. Furthermore, OFS is able to reduce the response time and energy consumption of mobile apps by speeding up the app execution through offloading support. As a result, the battery life of the mobile devices can be extended. Finally, we have learned that OFS works best for read-intensive apps, with few writes, and for systems that implement procedure offloading.

OFS has two limitations. First, due to the application level implementation, OFS is not aware of the physical locations of the files. Thus, even if a file is physically saved in a cloud storage, OFS still needs to fetch the file data from the mobile device before the data can be accessed by the tasks offloaded in the cloud. This increases the overhead of accessing files in a cloud storage. Second, OFS only supports the scenario, in which the mobile app on a single mobile device offloads tasks to the cloud. It cannot support mobile distributed apps offloading tasks from multiple mobile devices. Our future work on OFS will focus on removing these limitations.

Acknowledgments

This research was supported by the US National Science Foundation (NSF) under Grants No. CNS 1409523, SHF 1617749, CNS 1054754, and DGE 1565478, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No.A8650-15-C-7521. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, and AFRL. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein. A preliminary version of this paper appeared on MSST 2016 [1]

References

[1] J. Shan, N. R. Paiker, X. Ding, N. Gehani, R. Curtmola, and C. Borcea, “An overlay file system for cloud-assisted mobile applications,” in *Proc. 32nd Symp. Mass Storage Syst. Technol.*, 2016, pp. 1–14.

[2] I. Zhang, et al., “Customizable and extensible deployment for mobile/cloud applications,” in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 97–112.

[3] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, “Avatar: Mobile distributed computing in the cloud,” in *Proc. 3rd IEEE Int. Conf. Mobile Cloud Comput. Services Eng.*, 2015, pp. 151–156.

[4] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, “COMET: Code offload by migrating execution transparently,” in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 93–106.

[5] E. Cuervo, et al., “MAUI: Making smartphones last longer with code offload,” in *Proc. 8th Int. Conf. Mobile Syst. Appl. Services*, 2010, pp. 49–62.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic execution between mobile device and cloud,” in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.

[7] M. A. Khan, et al., “Moitree: A middleware for cloud-assisted mobile distributed apps,” in *Proc. 4th IEEE Int. Conf. Mobile Cloud Comput. Services Eng.*, 2016, pp. 21–30.

[8] B. Pawlowski, et al., “The NFS version 4 protocol,” in *Proc. 2nd Int. SANE Conf.*, 2000, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.7599&rank=2

[9] Dropbox. [Online]. Available: https://www.dropbox.com/, Accessed on: Jan. 12, 2017.

[10] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 3–25, 1992.

[11] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, “Exploiting weak connectivity for mobile file access,” *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 143–155, 1995.

[12] AspectJ. [Online]. Available: https://eclipse.org/aspectj/, Accessed on: Jan. 12, 2017.

[13] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, “Just-in-time provisioning for cyber foraging,” in *Proc. 11th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2013, pp. 153–166.

[14] O. Kirch, “Why NFS sucks,” in *Proc. Linux Symp.*, 2006, vol. 2, pp. 51–64.

[15] Filesystem in Userspace (FUSE), [Online]. Available: https://github.com/libfuse/libfuse, Accessed on: Jun. 12, 2017.

[16] Kryonet. [Online]. Available: https://github.com/EsotericSoftware/kryonet/, Accessed on: Jan. 12, 2017.

[17] G. Gezzi, “Smart execution of distributed application by balancing resources in mobile devices and cloud-based avatars,” Department of Computer Science and Engineering, Master’s thesis, University of Bologna, Bologna, Italy, 2016.

[18] W. R. Dieter and J. E. Lumpp Jr, “User-level checkpointing for linuxthreads programs,” in *Proc. USENIX Annu. Tech. Conf.*, 2001, pp. 81–92.

[19] Pin - A dynamic binary instrumentation tool. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, Accessed on: Jun. 14, 2017.

[20] ProbeDroid: A dynamic binary instrumentation kit for android app analysis. [Online]. Available: http://www.zsshen.org/program-analysis-and-binary-instrument, Accessed on: Jun. 12, 2017.

[21] OpenCV: Open source computer vision. [Online]. Available: http://opencv.org/, Accessed on: Jan. 12, 2017.

[22] PhoneLab: A smartphone platform testbed. [Online]. Available: https://www.phone-lab.org/, Accessed on: Jan. 02, 2017.

[23] Bionic sources (official repository). [Online]. Available: https://android.googlesource.com/platform/bionic/, Accessed on: Mar. 5, 2016.

[24] Trepn power profiler. [Online]. Available: https://developer.qualcomm.com/software/trepn-power-profiler, Accessed on: Jan. 12, 2017.

[25] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *Proc. IEEE INFOCOM*, 2012, pp. 945–953.

[26] A. Zanni, S. young Yu, S. Secci, R. Langer, P. Bellavista, and D. Macedo, “Automated offloading of android applications for computation/energy optimizations,” in *Proc. INFOCOM*, 2017, demo paper, http://infocom2017.ieee-infocom.org/program/demos-posters

[27] Y. Dong, et al., “RFS: A network file system for mobile devices and the cloud,” *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 101–111, 2011.

[28] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, “Reliable, consistent, and efficient data sync for mobile apps,” in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 359–372.

[29] D. Perkins, et al., “Simba: Tunable end-to-end data consistency for mobile apps,” in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 7:1–7:16.

[30] B. Atkin and K. P. Birman, “MFS: An adaptive distributed file system for mobile hosts,” in Cornell Computer Science Department Technical Report., Department of Computer Science Cornell University, Ithaca, 2003, http://www.cs.cornell.edu/batkin/docs/mfs.pdf

[31] E. B. Nightingale and J. Flinn, “Energy-efficiency and storage flexibility in the blue file system,” in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 363–378.

[32] R. Többicke, “Distributed file systems: Focus on andrew file system/distributed file service (AFS/DFS),” in *Proc. 13th IEEE Symp. Mass Storage Syst. Toward Distrib. Storage Data Manag. Syst.*, 1994, pp. 23–26.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.

Nafize Rabbani Paiker received the BS degrees from Military Institute of Science and Technology, Bangladesh, in 2008. He is working toward the PhD degree in the Department of Computer Science at New Jersey Institute of Technology. His research interests include mobile computing, cloud computing, parallel and distributed computing.

Jianchen Shan received the BS and MS degrees from Shanghai University, China, in 2008 and 2011. He is currently working toward the PhD degree in the Department of Computer Science, New Jersey Institute of Technology. His research interests include parallel and distributed computing and cloud computing.

Cristian Borcea received the PhD degree from rutgers university, in 2004. He is currently a professor in the Department of Computer Science, New Jersey Institute of Technology. He is also a visiting professor in the National Institute of Informatics, Tokyo, Japan. His research interests include mobile computing and sensing, ad hoc and vehicular networks, distributed systems, and cloud computing. He is a member of the ACM, the IEEE, and the USENIX.

Narain Gehaini received the PhD degree in computer science from Cornell University. He is currently a professor of computer science, New Jersey Institute of Technology. Previously, he was with Bell Labs. He has worked extensively in programming languages, software, and databases. He has authored several software systems, holds several patents, and has written many books and numerous papers in computer science.

Reza Curtmola received the BSc degree in computer science from the Politehnica University of Bucharest, Romania, in 2001, the MS degree in security informatics in 2003, and the PhD degree in computer science, in 2007, both from The Johns Hopkins University. He is an associate professor in the Department of Computer Science, NJIT. He spent one year as a postdoctoral research associate at Purdue University. He is the recipient of the NSF CAREER award. His research focuses on storage security, applied cryptography, and security aspects of wireless networks. He is a member of the ACM and the IEEE Computer Society.

Xiaoning Ding received the PhD degree in computer science and engineering from the Ohio State University. He is an assistant professor with New Jersey Institute of Technology. His interests include in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Hofstra University. Downloaded on March 18,2023 at 20:24:17 UTC from IEEE Xplore. Restrictions apply.