An Overlay File System for Cloud-Assisted Mobile Applications

Jianchen Shan*, Nafize R. Paiker*, Xiaoning Ding, Narain Gehani, Reza Curtmola, Cristian Borcea
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
{js622, nrp48, xiaoning.ding, narain.gehani, reza.curtmola, borcea}@njit.edu

***Abstract*—With cloud assistance, a mobile application can offload its resource-demanding computation tasks to the cloud (public cloud, cloudlet, or personal cloud, etc). This leads to a scenario where computation tasks in the same application run concurrently on both the mobile device and the cloud. These tasks need to save, read, and write files 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 in a manner that is efficient, consistent, and transparent to locations. The paper addresses this issue by designing 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 application delay when the latest data cannot be accessed locally. To guarantee location transparency, OFS creates an unified view of the data that is location independent and is accessible as local storage. Our experiments 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.**

I. Introduction

Mobile devices, such as smart phones and tablets, have become major personal computing devices. However, due to their compact size and mobility, mobile devices have limited computing resources (e.g., CPU power, energy supply, memory space, etc). Thus, to get the desired performance and energy conservation, various systems have been designed to allow a mobile application to use cloud resources (e.g., public cloud, personal cloud, or cloudlet) by offloading its resource-demanding tasks to the cloud in the form of threads, objects, or procedures [1]–[5]. For example, a mobile game 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 application 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 intensively studied, and a few programming models, along with the supporting middleware and system infrastructure, have been developed, e.g., Avatar [2], [6], MAUI [4], CloneCloud [5], Sapphire [1], and COMET [3]. However, supporting efficient file access, especially file sharing, with offloaded tasks in the same mobile application remains a challenging issue and has received little attention. Due to this issue, systems such as MAUI and COMET cannot offload application tasks if the tasks need to access files.

Existing file systems are not effective to handle remote file access for the offloaded tasks of mobile applications. Thus, they seriously limit the capability of mobile systems to freely offload tasks to the cloud. Network file systems and distributed file systems, such as NFS [7] and Dropbox [8], only support remote file access from the platforms where their client software is properly set up and configured. However, setting up and configuring 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 the access to the file. This requires that mobile applications 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 applications at low overhead. 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 (e.g., Coda [9], [10]), 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 applications.

To address these problems, we propose a 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 the tasks offloaded in the

* equal contribution in this work

form of threads, objects, or procedures. OFS is an application-level file system that manages file access and file sharing in a mobile application. It effectively hides the boundary between the mobile device and the cloud, and provides a unified environment for the tasks in the mobile application, such that the tasks can migrate freely between the mobile device and the cloud. 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 write-invalidate and write-update policies, to reduce file access latency and network traffic overhead, while ensuring strong consistency. 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 a few advantages for running cloud-assisted mobile applications. 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 application's user space, and thus can be duplicated and moved with the tasks to new locations. Third, it simplifies application development and system management. For example, with OFS, there is no need to set up clients and save the to-be-accessed files into a network/distributed file system before the application runs, and there is no need for handling different path names in the programs incurred by different mounting points on different devices. Programmers do not have to worry about whether a task is running on the mobile or offloaded to the cloud.

To the best of our knowledge, this is the first work that provides a system solution to support efficient file access in cloud-assisted mobile applications. We make the following contributions. First, we determine the requirements for a file system to effectively support offloading tasks to the cloud. Second, we propose and design OFS as a solution to meet these requirements. Third, we experimentally 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.

The rest of the paper is organized as follows. Section II outlines the background and motivation for designing OFS. Section III and IV present the design of OFS and consistency management techniques used by OFS, respectively. The evaluation of OFS is presented in section V. We then discuss related work in section VI. The conclusion and future work are presented in section VII.

II. Background and Motivation

This section introduces first several approaches to offload mobile application tasks to the cloud. Then, it presents a few mobile application examples to illustrate the demand for consistent and transparent file access and sharing. Finally, it analyzes and summarizes the requirements on file systems for cloud-assisted mobile applications, which underpin the design of OFS.

A. 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. To simplify programming, the tasks should not require modifications, and the program itself does not need to implement the offloading logic. Instead, the system software dynamically schedules and runs unmodified computation tasks of an application on the mobile device and the cloud.

To make scheduling decisions, the system uses a certain cost function, which balances the cost and the benefit of offloading a task based on a few factors, such as the workload of the task, dependencies on software and hardware resources, the state of the resources on the mobile device, network performance, and the overhead of transferring the task. To support the execution of unmodified tasks in the cloud, the system should recreate the execution contexts required by the tasks in the cloud, such as system support, supporting libraries, code, and all the required data sets. While system support, library, and sometimes application code can be pre-deployed, the data sets are usually transferred dynamically with the tasks or based on demand for a few reasons. For example, some data sets are generated/updated dynamically, and applications may use different data sets in different executions.

There are a few different methods to 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 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. However, migrating threads offers a few advantages over RPC, especially when distributed shared memory support is provided [3]. For example, a thread may be migrated at any time during the execution of the application, while with RPC only whole procedures can be offloaded.

Cloud-assistance can also be implemented with a VM-based approach (e.g., Cloudlet [11]). Since a VM is a complete running environment for an application, from memory state to storage, offloading tasks to the cloud can be achieved by migrating the VM containing the tasks. However, compared to moving a thread/object/procedure, migrating a VM inevitably incurs much higher overhead and sacrifices flexibility, since a VM has much more information (e.g., OS kernel state, buffered data, etc) than individual tasks and all the tasks in a VM must be moved together.

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

The collaborative tasks in an app run concurrently at the cloud and the mobile device, and they often need to access their data sets saved in files. However, existing file system designs cannot effectively support file access and file sharing across these platforms, as we will explain in this section. For this reason, systems supporting task offloading (such as COMET and MAUI) usually cannot migrate tasks if they need to access files. This seriously limits the capability of these systems to effectively offload tasks to the cloud.

This problem can be mitigated if the files to be accessed by a task are transferred before migrating the task. However, it is not easy to identify all these files, especially in cases when a task may need 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, and inconsistency may lead to incorrect results or application crashes.

B. Motivating Examples

With the growth in the number of mobile devices, the amount of data (e.g., multimedia data) generated and operated by mobile applications also increases. Many of these operations (e.g., image/video recognition and augmentation) are too expensive for mobile devices and require the help of the cloud for optimized performance [12]. At the same time, most mobile applications interact with users. Their interactive tasks must run on mobile devices for desirable user experience and reduced overhead. Some mobile applications rely on the hardware resources (e.g., sensors) on mobile devices, and the related tasks must also be executed on mobile devices. This lead to scenarios in which an application has tasks on the mobile device and tasks in the cloud working collaboratively.

For example, enhanced camera apps on mobile devices can take photos or video clips, use the cloud to analyze (e.g., recognizing the people and landmarks in the files and tagging them properly) and improve them (e.g., removing red eyes and reducing blurring), and play back the improved photos or video clips on mobile devices. In such an app, a thread taking the photos/videos needs to save them. A processing thread may be migrated to the cloud when it is about to process some photos/videos and the system estimates that the benefit of offloading the tasks (e.g., better user experience with lower response time) exceeds the overhead (e.g., the cost to transfer the thread and the photos/videos). The system may migrate the thread back when the thread needs to process some other photos and it is not cost-effective to transfer these photos to the cloud. Thus, the thread may read the saved photos/videos from the cloud or the mobile device, and generates improved photos/videos where it runs. The generated photos/videos are then read out by a thread on the mobile device for playback. At the same time, the processing thread and other threads in the app may form a pipeline and run concurrently. For example, the processing thread first sends back an improved photo/video segment, and when the thread on mobile device plays back this photo/video segment, the processing thread may improve another photo/video segment in the cloud concurrently. Thus, the photos/videos must be well-managed to satisfy the concurrent accesses from both the mobile device and the cloud,

In another example, a video surveillance app on a mobile device may keep recording videos, which are analyzed in the cloud in real time to promptly detect, recognize, and tag moving objects. Other interactive apps (e.g., doodle clipboard apps and games) need to recognize and understand (in the cloud) complex user inputs collected on mobile device (e.g., doodles drawn by the users, gesture and eye movements of the users), and react to these inputs. In all these apps, a file system that supports the tasks running on the mobile device and the cloud to access and share the photos/videos/doodles and other data saved in files is critical to effectively leverage the computing power of the cloud.

C. Requirements on File System Design

To support remote file access and file sharing among the distributed tasks of cloud-assisted mobile applications, a file system should be able to locate and transfer data, and to manage data sharing. To accommodate features of mobile applications 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 application 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.
- **Consistency**: Reading stale data may lead to incorrect results or crash an application. 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 application 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. At the same time, mobile users often need to pay for the network traffic through cellular networks. Thus, 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).
- **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 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 applications and organize them accordingly, since the files to be accessed by mobile applications may be determined by user requests.

Existing file systems cannot satisfy all these requirements. The next section presents our application-level file system, which we call *Overlay File System* (OFS), and explains how its design satisfies the aforementioned requirements.

III. OFS Design

A. Overall System Architecture

OFS is built as a component of the middleware runtime that offloads and manages tasks. Figure 1 illustrates the position of OFS on the mobile device and the cloud platforms, and explains how OFS interacts with other components in the platform. The objective of OFS is to provide efficient, transparent, and consistent file accesses and file sharing for the computation tasks in a cloud-assisted mobile application. For this purpose, OFS intercepts and monitors the file access requests from the tasks in the application. It fulfills the requests for accessing local files by passing them to the OS and then 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. Let us notice that the file may be stored on the mobile and requested by a task from the cloud or vice versa.

OFS maintains the consistency between the blocks in the block buffer and their counterparts saved in remote files, such that the tasks can always see the latest updates no matter where they run. In addition to file accesses, OFS must also handle other file related requests, such as opening/closing files, creating/removing files, etc. OFS handles these requests by forwarding them to the platform storing the files and by updating the related metadata maintained on both the platform that opens the file and on the platform that stores the file.

Unlike conventional file systems, which are part of the operating system, OFS functions at the application level. A part of OFS is a library dynamically linked with the application, and the other part is a run-time service that can be integrated into a task-offloading middleware or work as an independent middleware. OFS maintains all its data structures (e.g., information about the files, file accesses, and the block buffer) in the virtual address spaces of mobile applications.

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 an individual mobile application. It does not provide system-wide management, e.g.,

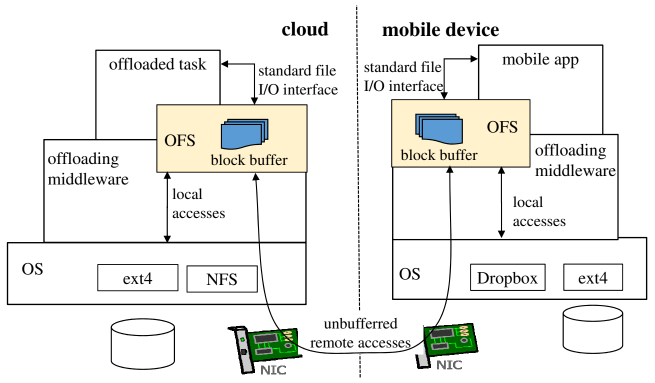
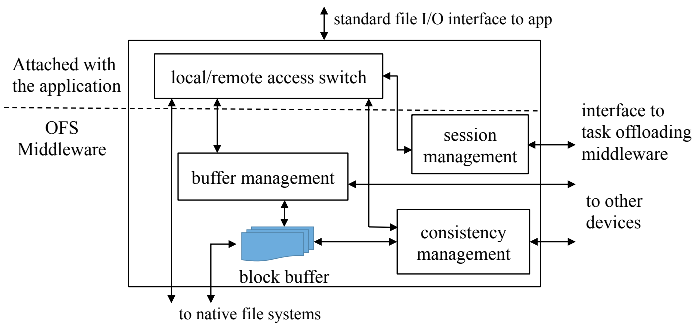


Fig. 1: Overall System Architecture



standard file I/O interface to app
Attached with the application
local/remote access switch
OFS Middleware
session management
interface to task offloading middleware
buffer management
to other devices
consistency management
block buffer
to native file systems

Fig. 2: OFS (Overlay File System) Architecture

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. Third, keeping all the functionality and data structures within the virtual space of the application 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 data blocks in virtual memory space does not incur costly kernel-application context switches.

B. OFS Architecture and Design

The overall structure of OFS is shown in Figure 2. OFS has two layers. The upper layer is implemented as a library, which consists of an interface to the application and the local/remote switch. The lower layer is implemented as a middleware runtime, which consists of three major components: (a) consistency management, (b) buffer management, (c) session management.

The *local/remote switch* intercepts the file I/O calls before they reach the system and decides for each call whether the call should be handled by a native file system or by OFS. OFS intercepts library calls, instead of system calls. The interception of library calls can be implemented with various approaches, e.g., manipulating symbol tables or binary weaving [13], [14]. Thus, OFS does not require a system-level privilege. Whether a file access is remote or local is determined based on the access history of task migrations; this information could also be pre-configured for certain files to improve the overall performance. For example, a task accesses remote files when it is offloaded, and accesses local files when migrated back. However, accesses to some files (e.g., libraries and some other files pre-distributed in the cloud) can be configured to be always local to reduce overhead. In OFS, a local file (e.g., a file on the mobile device) is relabeled as remote when it is accessed remotely (e.g., from the cloud) because the latest copy of the file may not be available locally.

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

The *buffer management* is in charge of managing and looking up the block buffer. For looking 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 can determine 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. The algorithm organizes all the buffered blocks into a linked list. When a file is closed, the algorithm moves all the blocks of the file to the LRU end of the list. When the content of a block becomes stale, the block is also moved to the LRU end. When a block is accessed, the algorithm moves it to the MRU (most-recently-used) end of the list. When space is needed, the algorithm selects and evicts the blocks at the LRU end.

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 in 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 applications. 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 mark the states of the files to be "remote" and establish the pointers to the location of the files, such that the local/remote file access switch can determine subsequent accesses in these sessions to be remote accesses. It also needs to copy the states (e.g., the current offset in each file, opening mode of the file, etc.) from the source platform.

Though buffering data improves efficiency, it incurs consistency issues. The *consistency management* component provides strong consistency, which is usually required by concurrent programming. For this purpose, it monitors all the accesses to the shared files, as well as the blocks cached in the block buffer. Enforcing strong consistency usually incurs a 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 applications. Thus, we use an adaptive algorithm named *delayed-update* combining both write-invalidate and write-update (Section IV) to reduce both latency and network traffic.

C. Workflow of OFS

In order to explain the detailed workflow of OFS, we will use the enhanced camera app, presented in Section II-B, as an example. As shown in Step 1 in Figure 3, a user starts the app, takes a photo, and attempts to store it on the mobile. The operation is intercepted by *local/remote file access switch*, which determines the operation to be local. Then, the app launches a thread to process the photo. Due to the heavy workload in the thread, it is offloaded to the cloud. As shown in Step 2 in the figure, in the cloud, the *session management* sets up the states required by the thread to access the photo and updates the configuration of the *local/remote file access switches*. Thus, upcoming accesses to the photo from the thread will be determined by the *local/remote file access switches* to be remote accesses and will be forwarded to the *buffer management*. In Step 3, when the thread processes the photo for the first time in the cloud (e.g., to detect faces), the photo is loaded into the *block buffer*. Then, the remaining processing of the photo in Step 4 (e.g., to recognize faces) can access the data cached in the *block buffer*. If the photo is changed (e.g., when it is be augmented), the *consistency management* in the cloud sends messages to its counterpart on the mobile device to invalidate and then update the changed blocks of the photo saved on the mobile device, as shown with Step 5 in the figure. Later on, the user interface thread of the app displays the photo on mobile device. The thread will automatically display the newly modified photo (Steps 6 and 7). The details about how consistency is maintained during the whole process are discussed in section IV. When the processing thread is migrated back, the remote file access sessions are destroyed (Step 8).

IV. Consistency Management in OFS

In this section, we first introduce several objectives for the consistency management design. We then describe the delayed-update algorithm used in our design.

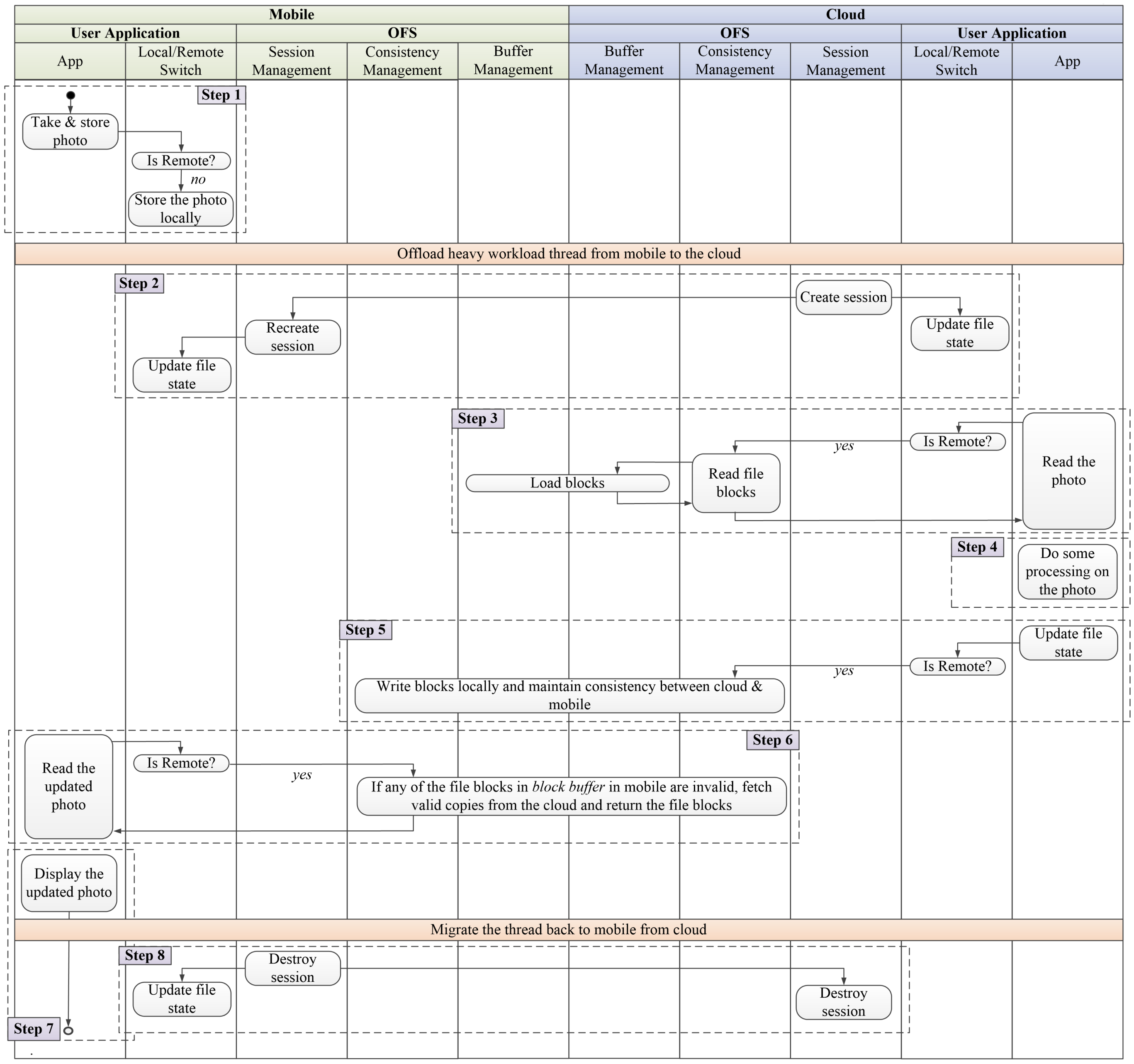


Fig. 3: The Workflow of OFS in an Enhanced Camera App

A. Design Objectives

The main goal of OFS is to provide an environment in which the tasks of a mobile application 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 applications, but will also simplify application development, because programmers will not be concerned with getting stale data in applications. 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 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.

B. 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. On a write operation, delayed-update invalidates duplicates first to ensure consistency. Then, it validates and updates the duplicates only when they are about to be 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 decide when the duplicates should be validated and updated. We address this issue by monitoring the file access patterns of mobile applications.

To satisfy the consistency demands other than strong consistency, we extend the delayed-update approach with a tunable knob called *relaxation* to relax the requirement on enforcing consistency. Using the same health monitoring app example, if the data analysis can use the data generated 5 seconds ago, relaxation is set to 5. To enforce strong consistency, relaxation should be set to 0. With a large relaxation value, delayed-update can update duplicates even less frequently to reduce resource consumption. In our implementation, we set the default value of relaxation to 0 and allow the application to adjust it.

The delayed-update algorithm keeps information to reflect the current status of a block. The following information is kept on both the mobile device and the cloud, for each block of data in the block buffer or in local storage that has been accessed by the application:

- A *shared* flag indicates if there are duplicates of the block cached in block buffers or saved in storages.
- 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 application 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 should be 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.

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 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 based on 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. If the threshold is 1, there is not any benefit to wait for more updates on the block. Thus, the algorithm does not delay the update to the remote duplicates any more. Instead, when the block is overwritten, the remote duplicate is immediately updated. In this case, the algorithm essentially begins to enforce a write-update policy. But, the algorithm still keeps monitoring the access patterns and updates the overwrite threshold accordingly. It resumes the normal delayed-update policy when the threshold increases.

V. Performance Evaluation

This section evaluates the performance of OFS using traces from real mobile users. The goal of the evaluation is three-fold: (1) compare OFS, and especially its delayed-update consistency policy, with NSF (i.e., close-to-open consistency policy) as well as write-update and write-invalidate consistency policies; (2) assess the performance of OFS and the comparison methods for both thread offloading and procedure offloading; and (3) understand the factors that lead to the benefits observed in OFS.

For the experiment, we use the following metrics:

- *Average I/O latency*: this metric measures the efficiency of OFS and comparison systems. We consider the the latency for all read and write operations, the latency for all reads, and the latency for all writes.
- *Network overhead*: this metric quantifies the network overhead introduced by each solution. It practically represents the cost of achieving lower I/O latency.
- *Number of overwrites per transfer*: this metric represents how many times a block is overwritten on average before it is transferred over the network. We use this metric to investigate the factors that lead to the OFS benefits
- *Hit ratio*: this metric evaluates the average rate of finding a block in the block buffer. Similar with the previous metric, this metric is used to gain insights into the benefits of OFS.
- *Mobile device active time*: this metric estimates the time the device is active, based on the frequency of I/O requests. Practically, it helps us estimate the benefits of offloading. The lower the values of this metric, the more beneficial the offloading (i.e., lower execution time on the mobile and implicitly lower battery consumption).
- *The ratio between I/O latency and active time*: this metric estimates the I/O sensitivity of a workload. Applications with a lower ratio benefit more from offloading.

Using these metrics, we show that with OFS most remote file accesses can be absorbed by block buffers. We also show that the delayed-update method can effectively reduce both network overhead and I/O latency. Therefore, offloading tasks to the cloud can effectively reduce the active time of mobile devices, leading to faster app execution and lower battery consumption on the mobiles.

A. Experiment Setup

We use a trace-driven emulation for our experiments. In the emulator, a mobile device is connected to a VM (an Amazon EC2 instance in US-East region) through a cellular network (LTE) with a latency of 35 milliseconds and a bandwidth of 5Mbps. In addition to the delayed-update policy in OFS, we also implemented a few alternative consistency policies in the emulator for comparison: write-update, write-invalidate, and the consistency policy in NFS [7]. NFS implements a close-to-open policy: when an NFS client closes a file, it flushes all the modified data back to the server; later, when another NFS client opens the file, the client can read the latest data from the server. For consistency, clients need to use file locks or shared reservations to avoid concurrent file sessions. This reduces the flexibility of accessing files concurrently.

Most traces used in the experiments were derived from those collected on the PhoneLab testbed [15] from real mobile users. Specifically, we used traces from six users, who were actively using their Android phones for different amounts of time. Each user executed different apps and had a different I/O pattern. The file I/O system calls were captured using boinic [16].

To imitate the concurrent execution of the tasks offloaded to the cloud and the tasks on the mobile device, we divided the file operations in the trace of each mobile phone user based on the threads performing the operations (i.e., thread IDs included in the trace), such that a portion of the I/O operations can be replayed in the cloud and the rest of the I/O operations can be replayed on the mobile device. We divided the file operations in each trace in the following two ways to imitate two different task offloading schemes:

- *Thread offloading*: we randomly selected 50% of the threads and replayed their file operations in the cloud, while the rest of the file operations are executed on the mobile device.
- *Procedure offloading*: for each thread, we first replayed 30% of its file operations on the mobile device, then replayed 50% of its file operations in the cloud, and finally replayed the rest (20%) of its file operations on the mobile device again. The 50% file operations replayed in the cloud emulate procedure offloading.

Thus, we obtained 12 workloads: one set of six traces for thread offloading, and one set of six traces for procedure offloading.

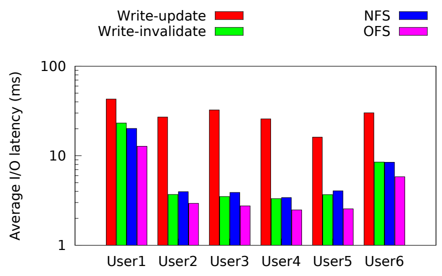
To better understand the performance of OFS, we also generated a synthetic trace, which is arguably an adversarial workload for OFS. In the trace, a thread running on the mobile device generates and saves data into files; and at the same time, another thread offloaded to the cloud reads the data stored in the files on the mobile by accessing the files remotely. Since each data block is written only once by the thread on the mobile device and then read once by the thread in the cloud, there is no potential for OFS to optimize performance by accumulating multiple overwrites and reusing the data in the block buffers.

B. Results with the Real Mobile User Traces

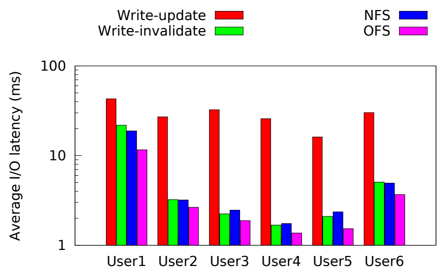
Figure 4 shows the average latency of file I/O operations for thread offloading and procedure offloading. The I/O latency mainly consists of network latency and the time to access the local storage. As shown in the figure, the average I/O latency with OFS is the lowest across all the workloads. The figure also shows that the workloads suffer the highest I/O latency with the write-update policy due to the overhead paid to update duplicates in block buffers on every overwrite. Compared to the write-update policy, OFS can reduce I/O latency from 70% to 95% for different workloads. The write-invalidate policy and NFS incur similar I/O latency. Compared to these policies, OFS can reduce the I/O latency by 16% $\sim$ 47% for the workloads.

The results also show that procedure offloading benefits more from OFS than thread offloading. The same behavior is observed for write-invalidate and NSF. This is because every thread offloads some procedures in the procedure offloading scenario, and thus all three methods have a higher potential for improvement. 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, Figure 5 shows the average latency for read operations and write



(a) Thread offloading

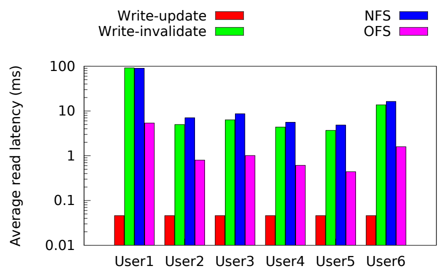


(b) Procedure offloading

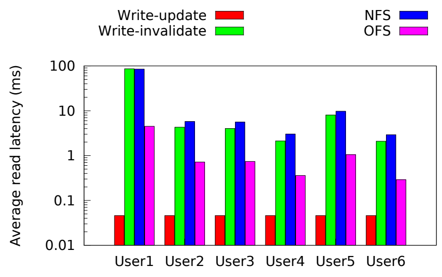
Fig. 4: Average I/O Latency for six mobile users. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS). The Y axis is in log scale.

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. The results show that OFS achieves as much as 14 times lower read latency than write-invalidate policy and NFS. This is due to the delayed-update policy in OFS, which adapts better to the workload characteristics. The cost for this large improvement in read latency is a higher write latency for OFS when compared to write-invalidate policy (3 times) and NFS (2 times). We also notice in Figure 5 that OFS performs better for procedure offloading than for thread offloading for both read and write operations (except for one user workload). These results are in line with those in the overall I/O latency experiment. Furthermore, the improvement in OFS read latency is larger than the improvement in OFS write latency for procedure offloading vs. thread offloading. Thus, from these results, we learn that OFS is expected to perform best for read intensive applications (i.e., with fewer writes) and systems based on procedure offloading.

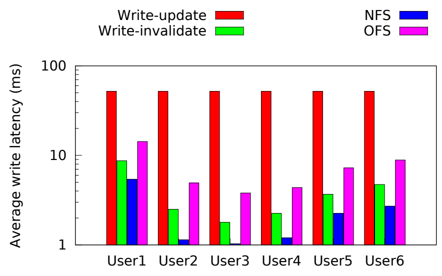
We also measured the amount of network overhead under the two sets of workloads. Figure 6 shows, as expected, that the largest network overhead is incurred by the write-update policy. Generally, the network overhead of OFS is much less than the overhead of write-update, and it is only a slightly higher than the overhead of write-invalidate and NFS (by 6% on average). 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. Therefore, these results clearly



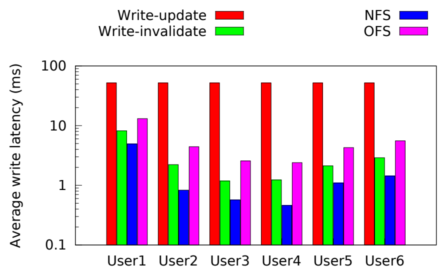
(a) Thread offloading (read)



(b) Procedure offloading (read)



(c) Thread offloading (write)

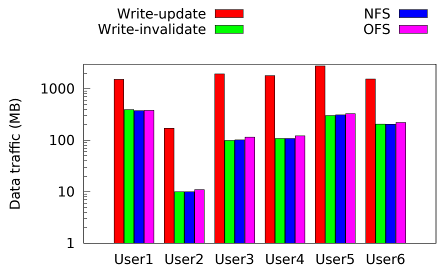


(d) Procedure offloading (write)

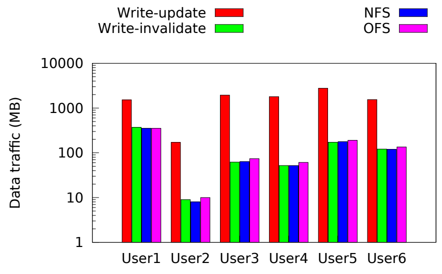
Fig. 5: Average latency of read operations and write operations. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS). The Y axis is in log scale.

demonstrate the advantages of OFS. It reduces the file I/O latency substantially compared to write-invalidate and NFS, while maintaining a similar network overhead. Write-update performs poorly in terms of both average file I/O latency and network overhead. Let us also note that similar to the results for file I/O latency, procedure offloading leads to lower network overhead.

To gain insights into the factors that lead to the OFS benefits, we collected the number of overwrites per transferred

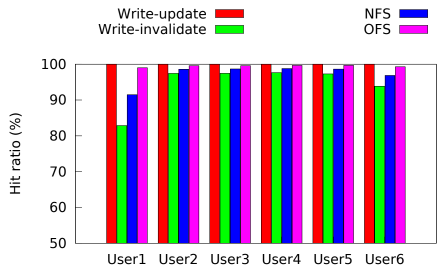


(a) Thread offloading

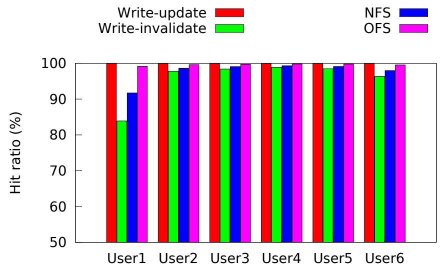


(b) Procedure offloading

Fig. 6: The amount of network overhead incurred by the workloads. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS). The Y axis is in log scale.

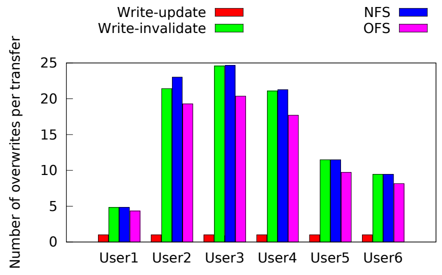


(a) Thread offloading

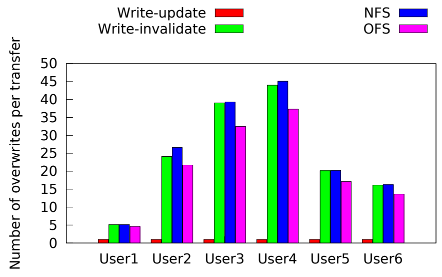


(b) Procedure offloading

Fig. 8: Block buffer hit ratios. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS).



(a) Thread offloading



(b) Procedure offloading

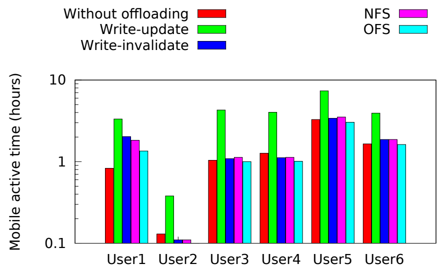
Fig. 7: The average number of overwrites per data transfer. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS).

data block and the hit ratios for the block buffers. OFS, write-invalidate policy, and NFS transfer data lazily. As shown in Figure 7, they transfer a data block when it is overwritten multiple times. This is the reason why these policies can effectively reduce the latency of write operations (Figure 5) and network overhead (Figure 6). The figure also shows that with OFS the average number of overwrites per transfer is not as high as that with the write-invalidate policy or NFS. This is because, with OFS, when the overwrite threshold is not accurately predicted, some blocks may be transferred too soon. This explains why the latency of write operations is slightly higher with OFS than that with the write-invalidate or NFS (Figure 5). This also explains why OFS incurs slightly higher network overhead than write-invalidate and NFS (Figure 6).

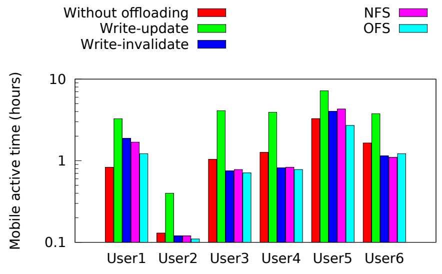
Figure 7 also shows that the number of overwrites is significantly higher for procedure offloading than for thread offloading. This result explains why procedure offloading performs better than thread offloading in terms of write latency and network overhead.

Figure 8 shows that block buffers have higher hit ratios with OFS than they do with write-invalidate and NFS. With OFS, the average hit ratio is over 99% for all the workloads. Such high hit ratios are achieved because updates in OFS may be transferred and saved in block buffers before subsequent accesses, and thus turn these accesses into "buffer hits". This explains why the latency of read operations is so much lower in OFS than that in write-invalidate and NFS (Figure 5). The high hit ratios with OFS also indicate that the delayed-update algorithm in OFS can effectively adjust the overwrite threshold and update duplicates to minimize buffer misses.

The results in Figure 8 also show that procedure offloading has slightly higher hit ratios than thread offloading. These results explain why the read latency is lower for procedure offloading than for thread offloading.

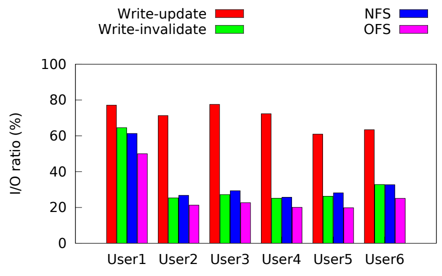


(a) Thread offloading

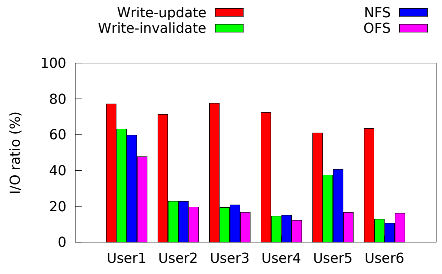


(b) Procedure offloading

Fig. 9: Total mobile device active time. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS). In addition, a scenario without offloading is presented. Y axis is in log scale.



(a) Thread offloading



(b) Procedure offloading

Fig. 10: The ratio between I/O latency and mobile device active time. Four consistency policies are compared: write-update, write-invalidate, close-to-open (NFS), and delayed-update (OFS).

To understand the overall benefits of OFS for apps (e.g., lower execution time) and mobile devices (e.g., lower battery consumption), we collected the active time of the mobile device under each workload. The intuition behind this metric is that longer active time means longer execution time for apps and higher battery consumption of the device. In the emulation, a mobile device is assumed to become inactive if there is no file I/O operation within 10 seconds; in other words, we assume that the short periods between I/O file operations are filled with computation. We also assume that the cloud has higher computing power than the mobile device, such that offloading computation tasks does not delay the file I/O operations.

Figure 9 shows the active time of the mobile device for the two sets of workloads and for the scenario in which task offloading is disabled. Except for the workloads of User 1, using OFS in conjunctions with offloading tasks to the cloud reduces the active time of the mobile device. OFS can reduce the active time by larger percentages (23% on average, and up to 39%) under procedure offloading workloads than under thread offloading (11% on average, and up to 23%). In the emulation, we offload 50% of threads (thread offloading) or 50% of computation tasks (with procedure offloading) to the cloud. The active time can be further reduced if more threads/computation tasks are offloaded.

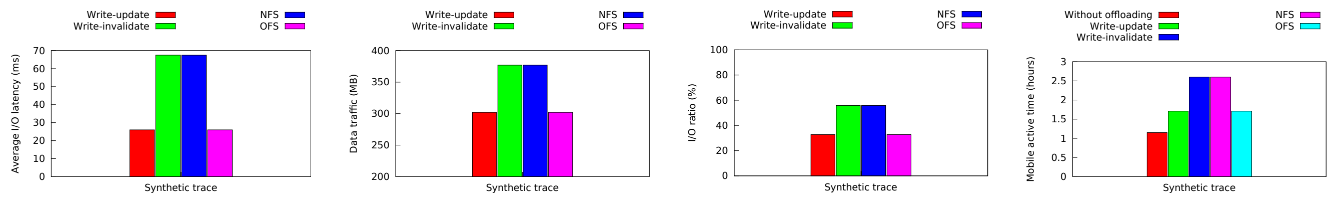
A major factor affecting the benefit of task-offloading is the ratio between the I/O latency and the active time of the mobile device, which is as shown in Figure 10. This metric estimates the I/O sensitivity of a workload in the sense that applications with a lower ratio benefit more from offloading. Among all the workloads, the active time is reduced by the largest percentage under the procedure offloading workloads of User 3, User 4, and User 6. This is because their I/O overhead accounts for a small percentage in these workloads. However, due to high latency, task-offloading cannot reduce the active time for the workloads of User 1 when compared to the scenario with task-offloading disabled.

The figure also shows that OFS can reduce the ratios more effectively than the other polices. Particularly, there are some workloads (e.g., the procedure offloading workload of User 5), under which the active time of the mobile device cannot be reduced by task-offloading with write-invalidate or NFS, but it can be reduced with OFS.

C. *Results with the Synthetic Trace*

Under the thread-offloading and procedure-offloading workloads generated with the traces from real mobile users, the write-update policy shows the lowest performance due to the high overhead paid to update the duplicates on all the data overwrites. However, there are applications in which data writes lack temporal locality, and the write-update policy may show its advantages.

To test the performance of OFS under such workloads, we repeat the emulation with the synthetic trace, in which each data block is written only once by the thread on the mobile device, and then is read once by the thread offloaded in the cloud. Figures 11(a) and 11(b) compare the average I/O latency and network overhead incurred by this workload. OFS



(a) Average I/O latency (b) Amount of network overhead (c) Ratio between I/O latency and mobile device active time (d) Mobile device active time

Fig. 11: Performance of OFS, write-update, write-invalidate, and NFS with the synthetic trace. The rightmost figure also contains a no offloading scenario.

and write-update perform best. This is because, with write-invalidate and NFS, in addition to transferring all the updates over network, extra costs must be paid to invalidate duplicates or to open/close files. OFS shows similar performance as write-update because the overwrite threshold is adjusted based on the access pattern, which effectively turns the delayed-update policy into a write-update policy.

Figure 11(c) shows that, even with OFS and write-update policy, the overall I/O latency is still high. Task-offloading cannot help reducing mobile active time in this case, as shown in Figure 11(d), because the thread running on the mobile device keeps the device active. However, in real applications, running all the threads on the mobile device may overload the device and lead to undesirable user experience and large energy consumption.

VI. Related Work

OFS is an easily deployable file system that supports seamless, transparent, consistent file I/O of mobile applications 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.

A. File I/O in Existing Cloud Offloading Systems

A few systems that offload computation from the mobile to the cloud have been developed [1], [3]–[5], [11], [17]. However, none of them is able to handle file I/O efficiently, if at all. Some of them, such as MAUI [4] and ThinkAir [17], 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.

CloneCloud [5] migrates threads in application-level VMs. To handle file I/O, CloneCloud punches through the abstract machine to the process system call interface. However, CloneCloud places all methods that share the same native state in either mobile device or all of them in the VM. In other words, if more than one method accesses the same file, either all of them have to be offloaded or none of them can be offloaded. Therefore, accessing and updating the same file from both the mobile device and the cloud simultaneously is not supported.

COMET [3] 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 [1] is a distributed programming platform for developing and deploying applications spanning mobile devices and clouds. Computation tasks are distributed using Sapphire Objects (SO) that encapsulate both data and code. Sapphire does not have a clear design on how to provide support for SOs to access remote files efficiently and consistently.

Just-in-time (JIT) provisioning in cloudlets [11] 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.

B. Distributed and Network File Systems

Various distributed and network file systems have been developed for different purposes and application scenarios [7], [9], [10], [18]–[22]. Most distributed and network file systems (e.g., NFS [7], AFS [23], Coda [9], [10], and BlueFS [22]) 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 application is offloaded to the cloud. Second, unlike OFS, conventional distributed and network file system 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.

C. Consistency Policies

Different policies are adopted in distributed and network file systems to enforce consistency. For example, Coda [9], [10] supports disconnected operations, which allow users to update files in Coda when network is disconnected. However, this leads to consistency issues that need to be solved by users. BlueFS [22] cannot avoid conflicts either, and it requires users to manually resolve conflicts. This is not practical for mobile applications that offload tasks to the cloud – any benefits in performance will be lost when users asked to help solve consistency issues through conflict resolution.

NFS [7] 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) [21] 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) [18] 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 [19], [20] provides a reliable and consistent synchronization service for mobile devices. With Simba, mobile applications 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 application's responsibility to call Simba API to register data, synchronize updates, and resolve conflicts. OFS, on the other hand, does not require applications to handle these operations, and can be used when applications does not have offloading logic.

Maintaining data consistency has been intensively studied. In addition to the consistency methods/policies discussed above, a large number of other solutions have been proposed for various specific parallel and distributed system scenarios [24]–[29]. OFS targets the scenario, in which concurrent and collaborative tasks run both on the mobile device and in the cloud, and may access the same file(s) concurrently. We have not found other work providing a consistency solution similar to that provided by OFS.

VII. Conclusion and Future Work

The study 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 app on the mobile device. To remove this obstacle, we have presented 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 mobile user traces have demonstrated that OFS can achieve substantially lower file access latency than competing methods with a similar network overhead. Furthermore, OFS is able to reduce the active time of mobile devices 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.

As future work, we plan to integrate OFS in our Moitree [6] middleware for mobile distributed apps supported by the cloud. In addition, we will improve the design of OFS to support file sharing between multiple applications. We also plan to allow OFS to share files saved in network file systems or distributed file systems in a more efficient way.

VIII. Acknowledgment

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1409523, CNS 1054754, DGE 1565478, and DUE 1241976, the National Security Agency (NSA) under Grant H98230-15-1-0274, 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, NSA, DARPA, and AFRL. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

References

[1] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 97–112.

[2] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in *The 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud '15)*, March 2015.

[3] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 93–106.

[4] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10)*, June 2010, pp. 49–62.

[5] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the 6th EuroSys Conference (EuroSys 2011)*, April 2011, pp. 301–314.

[6] M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, “Moitree: A middleware for cloud-assisted mobile distributed apps,” in *The 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud ’16)*, March 2016.

[7] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, “The nfs version 4 protocol,” in *Proceedings of the 2nd International SANE Conference*, 2000.

[8] “Dropbox,” https://www.dropbox.com/, [Online; accessed 12-February-2015].

[9] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, 1992.

[10] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, “Exploiting weak connectivity for mobile file access,” *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 143–155, 1995.

[11] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, “Just-in-time provisioning for cyber foraging,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’13. New York, NY, USA: ACM, 2013, pp. 153–166. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464451

[12] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, “A survey of computation offloading for mobile systems,” *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1007/s11036-012-0368-0

[13] W. R. Dieter and J. E. Lumpp Jr, “User-level checkpointing for linux-threads programs.” in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 81–92.

[14] R. Ramachandran, D. J. Pearce, and I. Welch, “AspectJ for multilevel security,” in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2006, pp. 13–17.

[15] “Phonelab: A smartphone platform testbed,” https://www.phone-lab.org/, [Online; accessed 02-Feb-2016].

[16] “Bionic sources (official repository),” https://android.googlesource.com/platform/bionic/, [Online; accessed 5-Mar-2016].

[17] 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 *Proceedings of the IEEE Infocom 2012*, March 2012, pp. 945–953.

[18] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, “Rfs: A network file system for mobile devices and the cloud,” *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 101–111, 2011.

[19] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, “Reliable, consistent, and efficient data sync for mobile apps,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST’15, 2015, pp. 359–372.

[20] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, “Simba: Tunable end-to-end data consistency for mobile apps,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15, 2015, pp. 7:1–7:16.

[21] B. Atkin and K. P. Birman, “Mfs: an adaptive distributed file system for mobile hosts,” in *Cornell University Technical Report*, 2003.

[22] E. B. Nightingale and J. Flinn, “Energy-efficiency and storage flexibility in the blue file system,” in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, ser. OSDI’04, 2004, pp. 363–378.

[23] R. Többicke, “Distributed file systems: Focus on andrew file system/distributed file service (afs/dfs),” in *Mass Storage Systems, 1994.’Towards Distributed Storage and Data Management Systems.’First International Symposium. Proceedings., Thirteenth IEEE Symposium on*. IEEE, 1994, pp. 23–26.

[24] J. Protic, M. Tomasevic, and V. Milutinovic, “Distributed shared memory: concepts and systems,” *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 4, no. 2, pp. 63–71, Summer 1996.

[25] J. B. Carter, “Distributed shared memory: concepts and systems,” *J. Parallel Distrib. Comput.*, vol. 29, no. 2, pp. 219–227, Sep. 1995. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1995.1119

[26] L. I. Kontothanassis, M. L. Scott, and R. Bianchini, “Lazy release consistency for hardware-coherent multiprocessors,” in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224170.224398

[27] L. Guangchun, Z. Jun, L. Xianliang, and L. Jun, “Hccm: A novel cache consistence mechanism,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 2, pp. 25–36, Apr. 2003. [Online]. Available: http://doi.acm.org/10.1145/769782.769786

[28] P. Bzoch and J. afark, “Maintaining cache consistency for mobile clients in distributed file system,” in *Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the*, Aug 2013, pp. 55–62.

[29] K. Fawaz and H. Artail, “Dcim: Distributed cache invalidation method for maintaining cache consistency in wireless mobile networks,” *IEEE Transactions on Mobile Computing*, vol. 12, no. 4, pp. 680–693, April 2013.