FogFS: A Fog File System For Hyper-Responsive Mobile Applications

Andreas Pamboris[1], Panayiotis Andreou[2], Irene Polycarpou[3], George Samaras[4]
[1]University of Central Lancashire and JARVIC Ltd, apamboris@uclan.ac.uk
[2]University of Central Lancashire and JARVIC Ltd, pgandreou@uclan.ac.uk
[3]University of Central Lancashire, ipolycarpou@uclan.ac.uk
[4]University of Cyprus, cssamara@cs.ucy.ac.cy

***Abstract*—Hyper-responsive mobile applications, such as augmented reality and online games, require ultra-low latency access to back-end services and data at runtime. While fog computing tries to meet such latency requirements by placing corresponding back-end services and data closer to clients, for e.g., within an access network, assuming a fixed back-end server throughout execution is problematic due to user mobility. A more flexible approach is thus required to allow for adapting to changes in network conditions when users roam, by relocating back-end services and data to closer available infrastructure. Support for real-time migration of software services exists, however, migrating associated disk state remains a bottleneck. This paper presents FOGFS, a fog file system that employs intelligent snapshotting, migration and synchronization mechanisms to speed up the migration of an application's disk state between different edge locations at runtime. The experimental evaluation of our prototype implementation reveals that the attainable speed-up is as much as $3.3\times$ compared to conventional migration approaches.**

***Index Terms*—fog file system, hyper-responsive mobile client applications, disk state migration, fog computing, edge computing**

I. INTRODUCTION

Smart-phones have become the predominant computing devices of choice nowadays, however, mobile applications are still constrained by their limited computational resources. Code-offloading approaches have been proposed in the past to leverage more powerful cloud-based resources for better performance [1]–[3]. Nevertheless, they often require transferring back and forth application data stored on mobile devices, which comes at a cost that may mask any attainable performance gains due to offloading. To address this problem, previous work focuses on statically partitioning applications and their state across a mobile device and a cloud-based server [4], [5]. Alternatively, applications may (by design) adhere to the client-server model, which ensures that back-end data and computations are collocated in the cloud. What is common in all such approaches is the fact that applications are ultimately split into two parts: the *client*, hosted on mobile devices, and the *back-end*, hosted on a particular cloud-based machine.

Hyper-responsive applications are a special category of clients that are adversely affected by high network latencies. For e.g., augmented-reality systems [6] require fast access to powerful servers in order to achieve seamless interactivity with the real world, while online gaming clients need to communicate in real time through centralized game services [7]. To satisfy such stringent latency requirements, fog-computing practices have emerged, which aim at placing back-end application components closer to the end user, for e.g., at the edge of an access network [8], [9]. Nevertheless, deciding the placement of back-end components a priori is often not as beneficial as one would expect due to user roaming. This has led to approaches that support switching between a predefined set of available *edge servers* based on the user's location during execution [10]. Nevertheless, while migrating back-end software processes from one edge server to another can be done efficiently, migrating an application's (potentially large) disk state in real time remains an open challenge.

This paper presents FOGFS, a fog file system that supports fast migration of an application's disk state between available edge servers to reduce the total handover time between them. A predefined set of edge servers is assumed, with each server initially containing the same base application file system. At any given point in time, only one server (*active*) is used. When users roam, if a more suitable server (*target*) is available, FOGFS caters to the migration of an application's disk state from *active* to *target*. To achieve this, it employs three main components: (i) a **checkpoint-based snapshotting mechanism** used to construct efficiently a minimal bundle of data for *target*, allowing it to reconstruct the application's disk state; (ii) a **migration mechanism** that parallelizes different steps involved with the migration process for better performance; and (iii) a decentralized **background synchronization mechanism** that tries to bring all available edge servers up to date with the most recent snapshotted version of an application's disk state, before a new migration is requested.

In summary, the contributions of this paper are: (i) it describes the design and implementation of FOGFS; and (ii) it demonstrates experimentally the potential of FOGFS to significantly speed up the migration of an application's disk state, compared to conventional approaches.

II. RELATED WORK

A. Edge/Fog Computing

The Mobile Edge (or Fog) Computing (MEC) paradigm [8] stemmed from the early work by Satyanarayanan et al. [9], who first envisioned a new system architecture for exploiting processing and storage capabilities at the edge of access networks. The idea of deploying micro data centers (referred to as *cloudlets*) in close proximity to mobile users resonated well with academic and industrial research communities, as it promised low-latency interactions between mobile clients and corresponding back-end services. As such, many have followed up on this vision, effectively identifying and addressing several challenges associated with MEC [11].

One such challenge relates to *user mobility*: mobile users that distance themselves from specific edge servers cause an increase in the network latency separating them from edge services. The follow-me cloud concept [10] proposes an architecture that allows edge services to migrate in unison with the user's movement patterns. To date, research has been done on related challenges, for e.g., mobility-aware online service placement frameworks [12], [13], hybrid edge deployment models for sharing edge infrastructure at scale [14], and platforms for migrating software services across edge servers [15]. However, the challenge of migrating an application's disk state fast enough remains open. Existing approaches either ignore the problem altogether, or employ standard migration mechanisms such as: (i) transferring the entire disk state at the destination edge location; or (ii) migrating entire Virtual Machines (VMs), with significant associated overheads involved. FOGFS tries to address this challenge through a more efficient approach that is tailored to the fog computing context.

B. VM Migration

The de facto standard for VM migration is *live migration* [16], which aims at minimizing *service down time* by allowing a VM instance to continue executing while migration is in progress. The process involves multiple iterations during which updates made to the VM state (during the previous iteration) are transmitted over to the destination host. Eventually, the source VM instance is suspended and all remaining modified state is sent to the destination host. As discussed in [17], live migration is inappropriate in the context of fog computing, since what matters the most is reducing total handover time (as opposed to service down time), which guarantees a faster switch to nearby edge infrastructure.

Existing VM migration approaches typically employ some form of incremental file synchronization mechanism, such as *rsync* [18], to encode and transmit only the difference between files residing on different hosts. Such mechanisms use a rolling hash function approach to compare different versions of the same file across hosts. FOGFS also employs a delta-encoding mechanism to infer the differences between two versions of an application's disk state. However, since, by design, both versions are available on the source host (*active*), it does so more efficiently, avoiding unnecessary network communication.

III. FOGFS DESIGN AND IMPLEMENTATION

FOGFS operates across a predefined set of edge servers, out of which only one is *active* at any given point in time. Activation of a different server (*target*) happens in order to adapt to changes in network conditions when corresponding users change their location at runtime. FOGFS handles the migration of the application's disk state. It comprises three main components that collectively aim at minimizing total handover time between edge servers, which are described next.

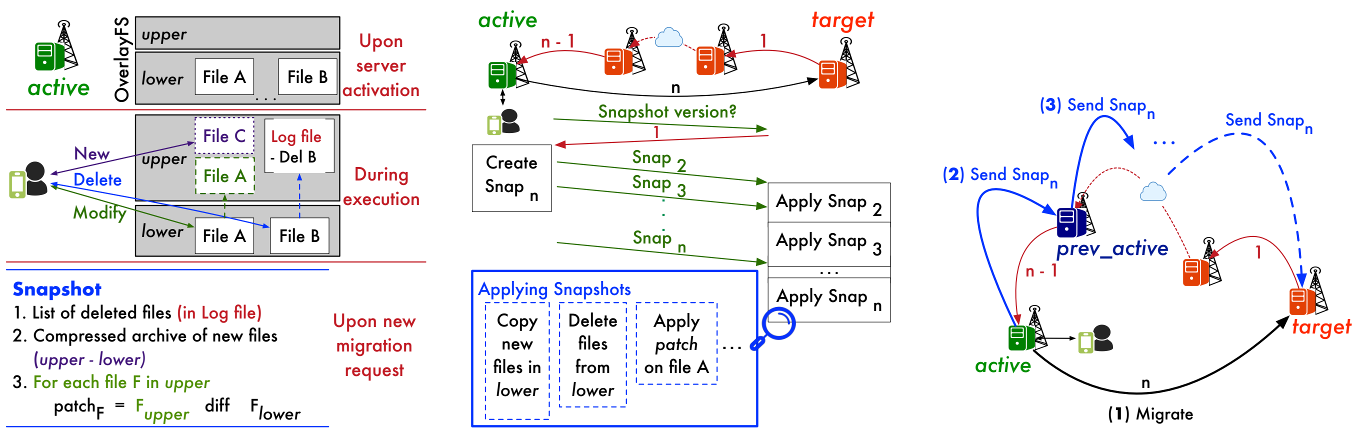
A. Checkpoint-based Snapshotting Mechanism

The first component is a *checkpoint-based snapshotting mechanism* used to create fast snapshots of an application's disk state (Fig. 1a). A *snapshot* is taken at well-defined checkpoints, i.e., right before a new migration begins, and contains only the differences between the up-to-date version of the application's disk state and the version at the previous checkpoint (i.e., when *active* last commenced operation).

1) Tracking disk state changes at runtime: FOGFS leverages OverlayFS [19], a layered union file system to distinguish between modified and unmodified application data files. OverlayFS supports overlaying the contents of one directory onto another and uses Copy-on-Write semantics to keep track of modifications at runtime. Behind the scenes, a file system is split into two main directories, *upper* and *lower*. Initially, all of an application's files are included in *lower*, which is a read-only directory. File access through OverlayFS first attempts to retrieve data from *upper*, and only defaults to *lower* if a file is not found. Any attempt to modify a file in *lower* automatically creates a copy in *upper*, which is the one that is actually modified. As a result, the base files remain unmodified in *lower*. Deleted files are removed from *upper*, if present, and their deletion is transparently logged.

FOGFS uses the above-mentioned mechanism as follows: (i) when a new server is activated, an OverlayFS instance is mounted, which contains all of an application's (up-to-date) files in *lower*; (ii) throughout execution, changes to the application's disk state are tracked (as described above); (iii) once a new server activation is requested, a snapshot of the application's disk state is taken (see §III-A2); and finally (iv) upon migrating to *target*, all of *active*'s *upper* files are copied into *lower* (replacing previous versions accordingly), deleted files are also deleted from *lower*, and the OverlayFS instance is re-mounted.

2) Creating snapshots: To create a snapshot of an application's disk state, first all new files on *active* are bundled together using the tape archiving (*tar*) utility [20], which also supports compression of multiple files. Furthermore, FOGFS creates a list of deleted files, which are referenced by their corresponding full-path file names. Finally, it uses the *diff* utility [21], which is a data comparison tool, to calculate the difference between the two versions of modified files, included in *lower* and *upper*, respectively. *diff* is line-oriented and attempts to determine the smallest set of deletions and insertions for creating one file from the other. Its output is called a *patch* and can be applied to the previous version of a given file in *target* in order to bring it up to date. FOGFS creates a mapping between *patches* and unique file names, which is included in the snapshot along with the corresponding set of *patches*.



(a) Checkpoint-based snapshotting mechanism (b) Migration mechanism (c) Background synchronization mechanism

Fig. 1: FOGFS Components

All aforementioned data and meta-data constitute a snapshot of an application's disk state, which is sent to *target* as part of the migration process. Each snapshot is assigned a unique identifier that denotes the chronological order of corresponding checkpoints, i.e., the identifier of the most recent snapshot taken is set to the identifier of the last snapshot received by *active* incremented by one. Provided that *target*'s version of an application's disk state is the same as the one of the previous checkpoint, reconstructing the most up-to-date version on *target* is trivial: all new files are copied into *target*'s *lower* directory; all deleted files are removed from *lower*; and all modified files are updated in *lower* by applying all corresponding *patches*.

B. Migration Mechanism

FOGFS's migration mechanism is described in Fig. 1b. While *active* always has access to all preceding snapshots of the application's disk state, used to reconstruct it when last activated, this may not be the case for *target*. Since server activations are arbitrary, *active* may have been preceded by a server other than *target*. For e.g., consider the following sequence of server activations: servers S_1, S_2 and S_3. When the handover between corresponding pairs of the aforementioned servers takes place, S_1 contains snapshot $Snap_1$, S_2 contains $Snap_1$ and $Snap_2$, while S_3 contains $Snap_1$, $Snap_2$ and $Snap_3$. If a migration from S_3 to S_1 is next in line, the former would have to provide the latter with $Snap_2$ and $Snap_3$, to be applied (in order) to S_1's version of the disk state.

FOGFS's migration mechanism involves a two-level concurrent operation of processes, which aim at reducing, to the extent possible, the overall migration time. Before commencing a migration, a first round of communication between *active* and *target* informs the former of the latter's most recent snapshot, say X. Thereafter, *active* starts sending all snapshots whose identifier is greater than X in ascending order. In parallel, *active* starts preparing the last snapshot, which contains the most recent changes made to the application's disk state. *target* on the other hand starts receiving consecutive snapshots, which are immediately applied to its current version of the disk state, in the order that they are received. For each snapshot received by *target*, multiple independent operations happen in parallel: (i) all new files are copied into *lower*; (ii) all deleted files are removed from *lower*; and (iii) *patches* are applied concurrently to all of *lower*'s affected files.

C. Background Synchronization Mechanism

FOGFS's *background synchronization mechanism* tries to bridge the gap between edge servers before the next migration takes place, by bringing their copy of the application's disk state "closer" to the one last snapshotted. This improves migration efficiency by removing the need for several rounds of snapshot transmission/processing from the critical path of the migration work-flow.

This mechanism is described in Fig. 1c. All edge servers that were activated at some point maintain a reference to their predecessor *active* server, coined *prev_active*. When a migration from *active* to *target* completes, the former attempts to send its most recent snapshot to *prev_active*. This carries on recursively in an attempt to propagate the most recent disk state changes to as many edge servers as possible. The process terminates when either: (i) the edge server contacted already has the corresponding snapshot (for e.g., when *prev_active* is equal to *target*); or (ii) *prev_active* references no other server (first server activated).

FOGFS's background synchronization mechanism does not guarantee that all available edge servers will eventually obtain the most recent snapshot. Instead, it is a fully decentralized, best effort approach, which avoids all associated challenges faced by centralized alternatives, namely with regards to complex management, fault-tolerance and scalability.

IV. Evaluation

This section demonstrates experimentally the potential of FOGFS to significantly speed up the migration of an application's disk state across different edge servers.

TABLE I: Disk state version difference per checkpoint

Checkpoint	S_1	S_2	S_3	S_4	S_5	**Diff.**
Initially	base	base	base	base	base	–
(S_1, S_2)	**Snap₁**	**base**	base	base	base	1
(S_2, S_3)	*Snap₁*	**Snap₂**	**base**	base	base	2
(S_3, S_4)	*Snap₁*	*Snap₂*	**Snap₃**	**base**	base	3
(S_4, S_5)	*Snap₁*	*Snap₂*	*Snap₃*	**Snap₄**	**base**	4
(S_5, S_1)	**Snap₁**	*Snap₂*	*Snap₃*	*Snap₄*	**Snap₅**	4

A. Experimental Methodology

1) Emulation Environment: Experiments were carried out in an emulated network environment using the CORE network emulator [22]. In particular, the topology used consists of five edge servers (referred to as S_1–S_5), which are interconnected through links that support data transfer rates of up to 100 Mbps, which is what is supported by ethernet-speed backhaul networks [23]. The hardware resources shared by the server nodes are: 8192 MB of RAM; a quad-core Intel i7-6700K (@4.00 GHz) CPU; and 500 Gb of SSD storage.

2) Workload Generator: A workload generator was implemented using the Python scripting language to emulate back-end services making changes to an application's disk state *DS* of size *DSSize*. The three types of changes possible are: (i) addition of new files; (ii) deletion of existing files; and (iii) modification of files, which entails adding and deleting data within existing files. The workload generator accepts as input the overall percentage of differences (*DiffRate*) between two consecutive checkpoints. Assuming a total size of: (i) new files (*new*); (ii) deleted files (*deleted*); (iii) additions within existing files ($file\_additions$); and (iv) deletions within existing files ($file\_deletions$), *DiffRate* is defined as follows:

$$DiffRate = \frac{new + deleted + file\_additions + file\_deletions}{DSSize}$$

Files in *DS* ranged in size between 50–100 MB. Changes to *DS* were such that: (i) *DSSize* remained the same; and (ii) the specified *DiffRate* was attained. Furthermore, the workload generator ensured an even spread of the different types of changes possible. It was found empirically that the actual number of new and deleted files had a negligible impact on overall migration efficiency, as long as their contribution to *DiffRate* remained the same. For e.g., adding one new file of size S, or two new files of size $S/2$ does not affect FogFS's migration performance. Given the above, the workload generator was designed as follows:

i) Add a new file of size $\frac{DiffRate}{3} \times DSSize$.
ii) Delete files until $\frac{DiffRate}{3} \times DSSize$ data are deleted.
iii) Delete $\frac{DiffRate}{6} \times DSSize$ data from files at random.
iv) Add $\frac{DiffRate}{6} \times DSSize$ data to files at random.

3) Experiments: We compare the performance of FogFS against two conventional alternatives: (i) Baseline that simply copies over the entire *DS* from *active* to *target*; and (ii) Diff&Patch, a simple delta-encoding approach that encodes and transmits only the changes made to *DS*. Diff&Patch uses the same techniques with FogFS for tracking changes and producing corresponding patches, albeit without FogFS's cleverness with regards to scheduling migration tasks and synchronizing *DS* snapshots in the background.

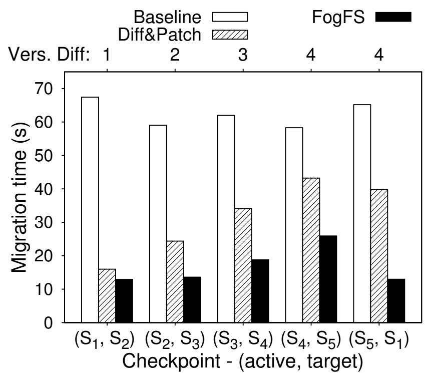
All three approaches are compared on the basis of *end-to-end migration time*, which is measured as the time spent from when a new migration initiates until the most updated version of *DS* is ready for use on *target*. For each experiment, a total of five migrations were triggered between all edge servers (S_1–S_5), the sequence of which was chosen to cover a range of *DS* version differences between *active* and *target* (not accounting for FogFS's background synchronization mechanism). More precisely, S_1 was initially set to be *active*. Migrations from *active* to *target*, which correspond to checkpoints denoted by (*active*, *target*), happened in the following order: (S_1, S_2), (S_2, S_3), (S_3, S_4), (S_4, S_5) and (S_5, S_1). All edge servers started off with the same *base DS*. Thereafter, for each successive checkpoint, the difference between the *DS* versions of *active* and *target* is shown in Table I.

A variable-sized application disk state was considered. Since for different values of *DSSize*, the difference in relative performance gains achieved using FogFS was negligible, and in the interest of space, the remainder of this section presents only the results obtained for small and medium-sized *DS* (containing files totaling 500 MB and 2.5 GB worth of data, respectively). Finally, the workload generator was configured appropriately to produce a variable amount of updates to *active*'s *DS*, which is specified through *DiffRate*. More specifically, we varied *DiffRate* from 10–40%, which empirically proved to be enough to stretch FogFS to its limit, as discussed in §IV-B4.

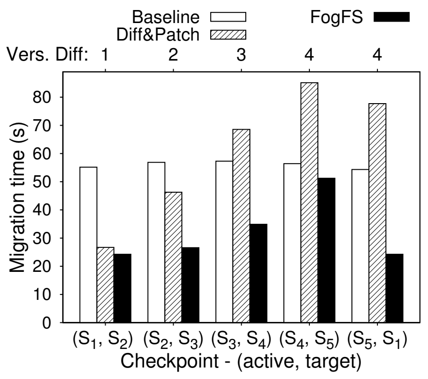
B. FogFS Migration Efficiency

FogFS's performance is compared against that of Baseline and Diff&Patch for disk state sizes of 500 MB (Fig. 2) and 2.5 GB (Fig. 3). Each figure includes three graphs, each corresponding to a different $DiffRate$ (ranging from 10–30%). The bottom x-axis shows the different checkpoints, while the top x-axis shows the corresponding *DS* version difference between *active* and *target* at each checkpoint. Since the relative performance gains achieved by FogFS follow similar trends for both *DSSizes* considered, our analysis primarily references the results for the 500 MB *DS* (Fig. 2).

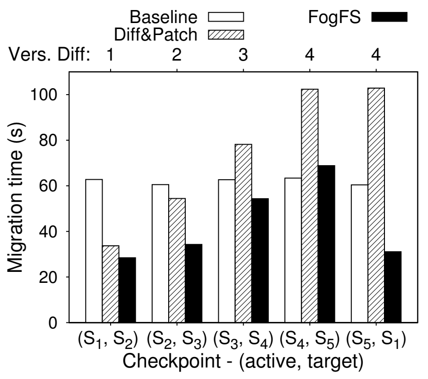
1) Baseline Vs. Diff&Patch: As expected, the performance of Baseline is roughly the same in all experiments, since it simply copies *active*'s version of *DS* over to *target*—*DSSize* does not change. The Diff&Patch approach, however, manages to significantly reduce migration time by as much as 4.2×—when the difference between *active*'s and *target*'s *DS* versions is equal to one (Fig. 2a). The reason for this is that Diff&Patch only transmits the changes between the two versions. Nevertheless, as the version difference increases from 1–4, the speed-up factor gradually decreases to approximately 1.4× due to the fact that more changes between the corresponding *DS* versions need to be transmitted/processed. For e.g., while a version difference of one implies changes of the amount of $DiffRate \times DSSize$, a higher *DS* version



(a) $DiffRate$ =10% changes

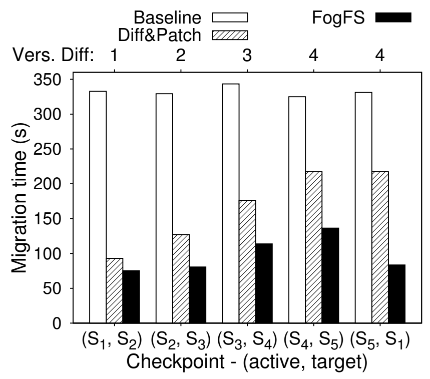


(b) $DiffRate$ =20% changes

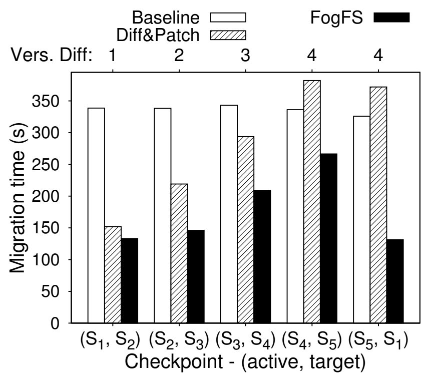


(c) $DiffRate$ =30% changes

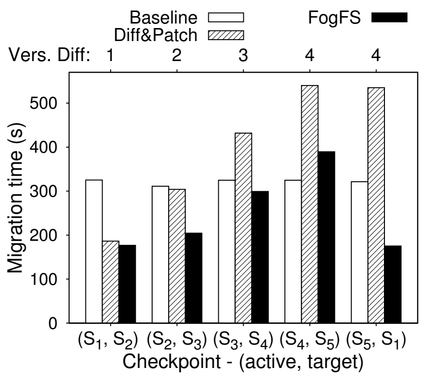
Fig. 2: FOGFS vs. BASELINE and DIFF&PATCH (migration time for *DSSize*=500 MB)



(a) $DiffRate$ =10% changes



(b) $DiffRate$ =20% changes



(c) $DiffRate$ =30% changes

Fig. 3: FOGFS vs. BASELINE and DIFF&PATCH (migration time for *DSSize* =2.5 GB)

difference, of say X, roughly translates to changes of as much as $DiffRate \times DSSize \times X$.

The same applies when the workload generator is configured to make more changes to *DS* between two consecutive checkpoints, which is controlled through the $DiffRate$ parameter (Fig. 2b and 2c). As $DiffRate$ increases from 10–30%, more changes to *DS* occur, which leads to a decrease in performance gains. For e.g., for a *DS* version difference of one, the corresponding speed-up factors when $DiffRate$ increases to 20% and 30% are 2.2× and 1.8×, respectively.

Beyond a certain amount of changes between *active*'s and *target*'s *DS* versions, DIFF&PATCH starts under-performing compared to BASELINE. The reason is that any gains in data transmission are no longer enough to mask the cost of additional processing required by DIFF&PATCH (calculating and applying *patches* to *target's DS*). In particular, for $DiffRate \geq 20\%$ and *DS* version differences of three and above, BASELINE performs better than DIFF&PATCH.

2) Effect of FOGFS *migration mechanism:* Without accounting for the background synchronization mechanism, FOGFS's performance follows similar patterns as those described for DIFF&PATCH. The performance gains over BASELINE gradually decrease as the amount of changes between *active*'s and *target*'s *DS* versions increases, either due to a larger difference between the *DS* versions of the two, or due to an increased $DiffRate$. Nevertheless, FOGFS consistently improves over DIFF&PATCH due to its efficient migration mechanism, which employs a two-level concurrent approach (see §III-B). When the *DS* version difference between *active* and *target* is one, the improvement is negligible since there is not enough scope for parallelism—only the most recent snapshot needs to be transmitted and processed. However, for larger version differences, FOGFS significantly outperforms DIFF&PATCH by approximately 1.8×. As a result, the conditions for which BASELINE outperforms FOGFS are less conservative: for $DiffRate \geq 30\%$ and *DS* version differences of four and above.

3) Effect of FOGFS *background synchronization mechanism:* In the best case scenario, FOGFS's background synchronization mechanism manages to reduce the theoretical *DS* version difference between *active* and *target* (captured by the top x-axis) to one, which is the minimum possible value. This is indeed the case for the migration that takes place at checkpoint (S_5, S_1), since at the end of all preceding migrations, the most recent snapshot was propagated to all other edge servers. When this occurs, FOGFS essentially incurs the minimum possible migration cost for a given $DiffRate$. For the configuration used, FOGFS achieves a speed-up of approximately 3.3× over DIFF&PATCH, which would increase with higher theoretical version differences between *active* and *target*. In the general case, assuming a theoretical version difference of four, FOGFS's

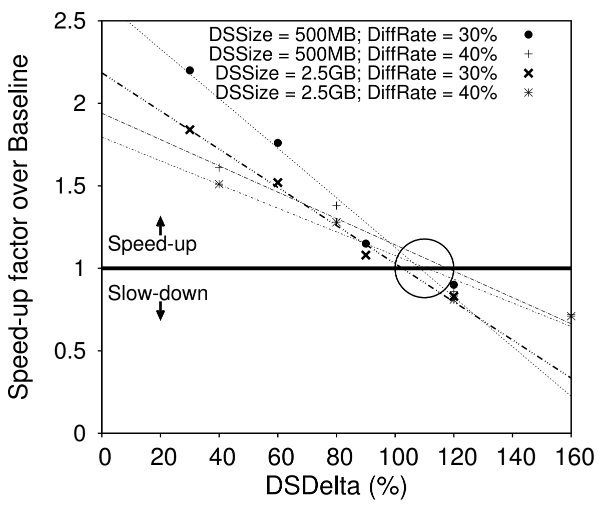


Fig. 4: FOGFS performance for increasing *DSDeltas*

migration time is expected to range between the corresponding values reported for checkpoints (S_4, S_5) and (S_5, S_1).

4) Discussion: Overall, the relative amount of *DS* changes required by *target* for a given migration, coined *DSDelta*, determines whether FOGFS performs better than BASELINE. Based on the configuration of the experiments ran, *DSDelta* can be approximated by multiplying the *DS* version difference between *active* and *target* by $DiffRate$. To identify the threshold *DSDelta* beyond which BASELINE outperforms FOGFS, we ran experiments that varied *DSDelta* and measured FOGFS's speed-up over BASELINE. We considered file systems of sizes 500 MB and 2.5 GB, while *DSDiff* was set to relatively high values, namely, 30% and 40%. This yielded four sets of results, which are shown in Fig. 4. While increasing $DSDelta$, the speed-up achieved by FOGFS gradually decreases in all cases, whereas beyond a certain point (around 100–120%), BASELINE starts performing better than FOGFS.

This information can be used to devise a migration policy that switches at runtime between FOGFS and BASELINE in order to maximize performance. Knowing *target*'s most recent *DS* version (say X), *active* can quickly calculate *DSDelta* by summing the sizes of subsequent snapshots ($> X$) and dividing the sum by *DSSize*. Based on the *DSDelta* threshold identified, *active* could then decide dynamically whether to use BASELINE instead of FOGFS. However, reverting to BASELINE at runtime requires additional handling of snapshots, since for future migrations, *target* will not have access to *DS* snapshots $> X$. Realizing such a hybrid solution is left for future work.

V. CONCLUSIONS

FOGFS supports the fast migration of an application's disk state between available edge servers to facilitate hyper-responsive mobile clients that require ultra-low latency access to this data throughout execution. It comprises three main components: (i) a checkpoint-based snapshotting mechanism that creates fast snapshots of changes made to an application's disk state, which are required by newly activated edge servers; (ii) a migration mechanism designed to speed up migrations by taking advantage of opportunities for parallel processing; and (iii) a decentralized background synchronization mechanism, which further improves migration efficiency by bridging the gap between edge servers containing different versions of the application's disk state, before a new migration is requested. We experimentally evaluated our prototype implementation of FOGFS and have shown its potential to reduce migration time by as much as 3.3× compared to conventional approaches.

ACKNOWLEDGMENT

We thank Jiayi Fu and Michail Christou for contributing significantly to the implementation of FOGFS. This paper is dedicated to the memory of our loving colleague and friend, Prof. George Samaras, who recently passed away.

REFERENCES

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *MobiSys*, 2010.
[2] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution between Mobile device and Cloud," in *EuroSys*, 2011.
[3] A. Pamboris and P. Pietzuch, "EdgeReduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies," in *MobileSoft*, 2015.
[4] ——, "C-RAM: Breaking Mobile Device Memory Barriers Using the Cloud," in *Transactions on Mobile Computing*, 2015.
[5] A. Pamboris, "Mobile code offloading for multiple resources," Ph.D. dissertation, Imperial College London, UK, 2013.
[6] S. Alshaal, S. Michael, A. Pamboris, H. Herodotou, G. Samaras, and P. Andreou, "Enhancing Virtual Reality Systems with Smart Wearable Devices," in *MDM*, 2016.
[7] M. Báguena, A. Pamboris, P. Pietzuch, M. Sichitiu, and P. Manzoni, "Better Performance in LTE Networks with Edge Assistance: The World of Warcraft Case," in *MOBIQUITOUS*, 2015.
[8] ETSI ISG MEC., *Mobile-Edge Computing - Introductory Technical White Paper*, 2014.
[9] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," in *PerCom*, 2009.
[10] T. Taleb, P. Hasselmeyer, and F. G. Mir, "Follow-Me Cloud: An OpenFlow-Based Implementation," in *GREENCOM-ITHINGS-CPSCOM*, 2013.
[11] U. Shaukat, E. Ahmed, Z. Anwar, and F. Xia, "Cloudlet Deployment in Local Wireless Networks: Motivation, Architectures, Applications, and Open Challenges," *Journal of Network and Computer Applications*, 2016.
[12] A. Ksentini, T. Taleb, and M. Chen, "A Markov Decision Process-based Service Migration Procedure for Follow Me Cloud," in *ICC*, 2014.
[13] T. Ouyang, Z. Zhou, and X. Chen, "Follow Me at the Edge: Mobility-Aware Dynamic Service Placement for Mobile Edge Computing," in *IWQoS*, 2018.
[14] M. Baguena, A. Pamboris, P. Pietzuch, G. Samaras, M. Sichitiu, and P. Manzoni, "Towards Enabling Hyper-Responsive Mobile Apps at Scale Using Edge Assistance," in *CCNC*, 2016.
[15] A. Pamboris, M. Báguena, A. L. Wolf, P. Manzoni, and P. Pietzuch, "Demo:: NOMAD: An Edge Cloud Platform for Hyper-Responsive Mobile Apps," in *MobiSys*, 2015.
[16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI*, 2005.
[17] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan, "Adaptive VM Handoff Across Cloudlets," in *Technical Report CMU-CS-15-113*, 2015.
[18] rsync, https://rsync.samba.org.
[19] N. Brown, *Overlay Filesystem*, https://goo.gl/D1mEQo.
[20] *FreeBSD Manual Pages*, https://goo.gl/QLeWP2.
[21] *GNU Diffutils*, https://www.gnu.org/software/diffutils/.
[22] Networks and Communications Systems Branch, *CORE*, http://goo.gl/pXIZsV.
[23] HSI, *What is a Backhaul?*, https://www.highspeedinternet.com/resources/what-is-a-backhaul/.