MobileSwap: Cross-Device Memory Swapping for Mobile Devices

Changlong Li[1,2,3], Liang Shi[1,2], Chun Jason Xue[3]
[1]Software/Hardware Co-design Engineering Research Center, Ministry of Education
[2]East China Normal University
[3]City University of Hong Kong
changlli@cityu.edu.hk, shi.liang.hk@gmail.com, jasonxue@cityu.edu.hk

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586108

***Abstract*—This paper presents MobileSwap, a cross-device memory swapping scheme for mobile devices. It exploits the unbalanced utilization of memory resources across devices. MobileSwap achieves comparable-to-local swapping performance based on existing network infrastructure. This is realized by two novel approaches: resource dedicated swapping for fast swapping among devices and app aware swapping for network connectivity considerations. MobileSwap is implemented and deployed on real mobile devices. Experimental results show that MobileSwap can enhance app caching capability by 2x compared with no swapping, and improve performance by 2.3x compared with state-of-the-art remote swapping. More importantly, local swapping induced read-write conflicts are largely removed.**

***Index Terms*—swapping, memory management, mobile devices**

I. Introduction

With the rapid development of mobile devices, more and more apps are installed and used by users. Enabling fast-switching and state-restoration is critical for user experience. App caching is an important method that caches the entire pages of apps in the main memory. This approach requires a large number of memory resources. As memory capacity is often constrained in mobile devices, only a few apps can be cached in the main memory at a time [1]. Modern solutions try to release memory pressure by evicting memory pages to the local storage [2]. However, local storage swapping introduces a large number of write requests to Flash storage, which leads to read-write conflicts [3], and seriously degrades system performance. A set of experiments show that the I/O latency of mobile apps can be increased by 2.4x on average when read operations are interfered by the swapping induced writes (Fig. 1). Furthermore, additional writes generated by local swapping are detrimental to Flash storage's already limited endurance. This paper explores swapping through remote I/O which can effectively overcome local storage swapping limitations.

There are several existing schemes proposed to enable remote swapping. InfiniSwap [4] opportunistically harvests and transparently exposes unused memory by dividing the swap space of each machine into many slabs and distributing them across machines' remote memory. To improve performance, the network stack is moved to the hardware. CloudSwap [5] is proposed to selectively swap data to the cloud. The schemes are designed for data centers and cloud systems. Different from existing works, this paper will propose the first cross-device swapping scheme among mobile devices.

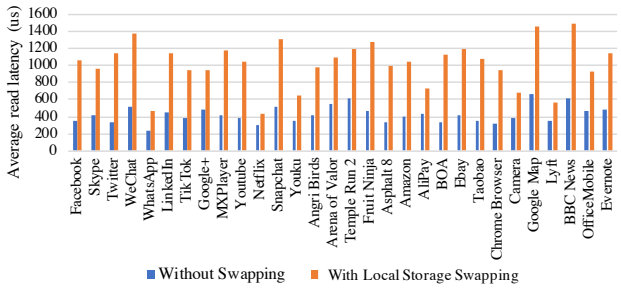

Fig. 1. The average read latency of mobile apps increased significantly when interfered by the swap induced writes in the background.

The basic idea of MobileSwap is motivated by two observations. First, with the development of wireless communication technologies, such as 5G [6] and WiFi-Direct [7], the performance of remote I/O has been significantly improved. Second, memory resource utilization is often unbalanced. For example, a smartphone's memory may be exhausted, while other smart devices around it, such as tablet, laptop, and smart TV, are idle. Under these two observations, memory pages in the heavy-loaded device can be swapped to the light-loaded devices around it. There are several challenges to enable the above scheme: First, even though the network infrastructure has been significantly improved, it is not designed for swapping. Our study shows that when conventional *swapping* works together with *networking*, the throughput is much lower than either one of them works alone. This is because existing swap and network subsystems were designed separately. Second, besides efficiency, network connectivity is another concern. Different from local storage I/O, remote I/O can be disconnected in many scenarios. For example, when one leaves home, the smartphone carried will be disconnect from the TV at home. Even if when one stays at home, remote I/O can also fail due to 'TV is powered off'. Once disconnection occurs, the swapped pages cannot be accessed again. Third, the overhead should be effectively controlled. To address the above issues, this paper makes the following contributions.

Liang Shi is the corresponding author. This work is supported by the NSFC 62072177, Shanghai Science and Technology Project (20ZR1417200), the Fundamental Research Funds for the Central Universities, and the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11219319).

- Resource Dedicated Swapping (RDS): To avoid performance degradation caused by swapping-networking incompatibility, the required resources are dedicated and the data in the channel are organized in the 'right-size'. The slowdown caused by the uncoordinated collaboration of networking and swapping is significantly improved.
- App Aware Swapping (AAS): Different from conventional page granularity memory reclaiming, app is treated as the basic unit of MobileSwap. Pages belonging to one app are swapped to the same remote device. With AAS, the impact of I/O disconnection is effectively minimized.
- MobileSwap is implemented and deployed on real-world mobile devices. Experimental results show that MobileSwap outperforms state-of-the-art remote swapping by 2.3x. The number of cached apps is doubled compared to no swapping when enabling MobileSwap, while no app crash occurs when the network is disconnected.

II. Background and Related Work

A. App Caching on Mobile Devices

Mobile systems such as Android prefer to cache apps in memory when switching them to the background, rather than releasing their pages directly. When launching an app without caching, the mobile platform first creates the process for the app, then launches UI (user interface) and lays out the screen. This launch of an app is referred to as $cold\ launch$. With app caching, several apps reside in memory. The system switches the cached app to the foreground when necessary. These launches of apps are referred to as $warm\ launch$. App caching and warm launching are important to mobile devices. First, launch speed is significantly improved. As investigated, cold launching takes 4-17x longer than directly restarting from the memory. Second, app caching makes it possible to maintain app states, such as play progress of YouTube or navigation record of Google Map. As a result, the app caching capability has become an important feature for mobile systems.

Due to the importance of app caching, the swapping technique plays an important role in mobile systems. By swapping some pages out, more apps can be cached in the background, thus the user experience can be significantly improved.

B. Related Work

Swapping is a widely studied topic. Due to the recent rapid development of mobile devices, this technique becomes even more important. Fig. 2 shows the architecture for swapping. There are two types of swapping in general, local swapping and remote swapping. The design and related work on these two types of swapping are presented as follows.

Local swapping is designed to evict pages from memory to a local swap area (SA). An SA could be a partition on the Flash or a virtual block device on DRAM. In this architecture, memory pages are evicted to SA through swap stack and storage stack. Swap stack includes the condition to perform swapping, identifies inactive pages, and determines the swapping scale. Storage stack converts the evicted memory pages to block I/O requests. Then dispatching these requests to

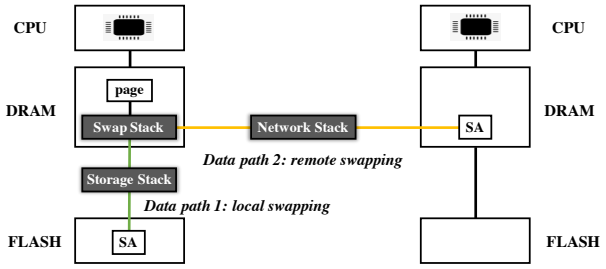

Fig. 2. The architecture for memory swapping.

SA. Local swapping is the most widely studied scheme. For instance, MARS [9] is designed to speed up app launching through flash-aware swapping. ZRAM [10] compresses pages to a RAM disk based swap area in DRAM. Based on that, Li et al. [8] proposed a two-level swapping framework for user experience improvement, and Zhong et al. [1] built a swap partition with nonvolatile memory (NVM) to improve the performance of smartphones. However, constrained by the hardware in mobile devices, including capacity and lifetime, these schemes have limited effects.

The other type of swapping is remote swapping. Remote swapping is designed by maintaining SA in the remote device memory. During remote swapping, evicted pages are converted to network packets and then delivered to the remote devices. In this architecture, network stack, as well as swap stack are utilized for the swapping process. There are several works proposed to design remote swapping. CloudSwap [5] makes use of the cloud resource to release the memory pressure of local devices. InfiniSwap [4] is designed to realize remote swapping based on InfiniBand in data centers. They are designed to fully exploit the fast bandwidth of RDMA. These works target cloud systems and data centers. This work targets the swapping scheme among mobile devices.

III. MobileSwap Design

A. Overview

This paper proposes MobileSwap, a cross-device memory swapping scheme for mobile devices. The basic idea of MobileSwap is to enhance the app caching capability of a busy device by swapping memory pages to the remote idle devices. However, there are several challenges to realize the above idea. First, the swapping efficiency should be ensured based on existing network infrastructure. Second, network connectivity issues should be considered. Third, since mobile devices are power sensitive, the overhead should be optimized and controlled. Inspired by these challenges, MobileSwap proposes two novel techniques: resource dedicated swapping (RDS) and app aware swapping (AAS).

Fig. 3 shows an example for MobileSwap. Assume there are four mobile devices with unbalanced resource utilization. To enhance the app caching capability of device A, MobileSwap detects idle devices (B, C) and moves A's data to them. A heartbeat is maintained to check the resource utilization states, specifically, the available memory capacity of devices. First, for RDS, resources in the swapping process are

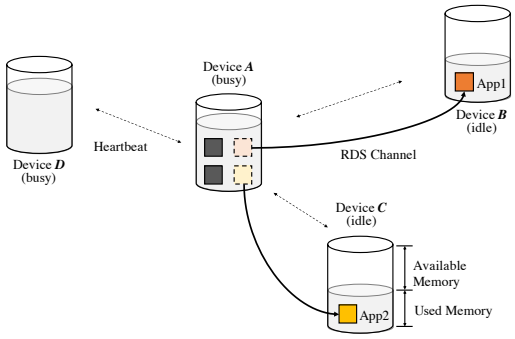

Fig. 3. MobileSwap overview. Four mobile devices with unbalanced resource utilization. Data at the heavy-loaded device A are swapped to the light-loaded devices B and C.

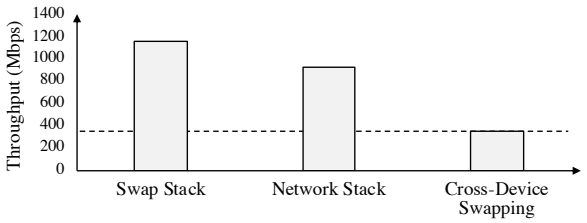

Fig. 4. Throughput comparison among swapping, networking, and cross-device swapping.

dedicated. These resources constitute a fast channel between A and each remote idle device. With this approach, the cost between swapping and network can be minimized for swapping performance. Second, for AAS, MobileSwap swaps the data with app granularity. Pages belonging to the same app are swapped to the same destination. When the swapped app is switched to the foreground again, all its pages will be swapped back. If the environment changes, for example, the network disconnected, the impacted apps can easily be traced. MobileSwap kills the impacted apps in the background and cold launch them when they are switched to the foreground the next time. In this way, the impact of disconnection becomes controllable and transparent to users. In the following, this paper introduces the details of the two proposed techniques.

B. RDS: Resource Dedicated Swapping

The design of RDS is motivated by our preliminary study: The peak throughput of cross-device swapping is much less than either swapping or networking alone. As shown in Fig. 4, the capability is evaluated (setup detail is shown in section IV) by collecting the number of pages processed by these processes. First, the swap stack can sustain 1100 Mbps, which relies on the host systems. Second, the network stack can sustain 920 Mbps, which relies on network bandwidth. However, when these two stacks are combined, the sustained throughput is reduced to only 355 Mbps. The reason comes from that when swapping and networking mechanisms coordinate, several issues happen: (1) The context switch overhead is high. The data path to achieve a cross-device swapping is long and complicated. Log analysis shows that resource preempting or interrupts happen frequently during the whole process. (2) The swapped data is not in the right size. In Linux kernel, memory is managed at page granularity. However, page-grained handling and transmission is not an efficient approach to networking. To solve these issues, RDS is designed with two schemes, fast channel and page batching, detailed as follows.

1) RDS channel: MobileSwap avoids redundant context switches by allocating dedicated resources for swapping. There are also many efforts using dedicated resources to improve system efficiency [11]. However, as presented, the performance deterioration is caused by incompatibility between two independent subsystems. In RDS, both swapping and networking required resources are dedicated. Specifically, MobileSwap uses three sets of dedicated resources to build a channel, including the network link, swap queue, and computing unit. First, a link is dedicated to each remote device. MobileSwap maintains one TCP socket instance and establishes a long-lived TCP link with that device. Second, a queue is dedicated to the swapped pages. All pages swapping to the same destination are inserted into the same queue. Third, a per-core worker thread is dedicated to moving swapped pages bidirectionally on the dedicated resources. Fig. 5 shows an example of this process. In this example, device A builds RDS channel with devices B, C, and D. Note that each channel's worker thread runs on one single core, then context switch is avoided during execution. Core and channel is not one-to-one mapped, two channels can be maintained by one core. Suppose device A has only two cores, Cr_1 and Cr_2. If channel (A to B) is running on Cr_1, only one of the other two channels (A to C) and (A to D) are allowed to run at the same time on Cr_2. The new channel must wait until the dedicated core is released by others.

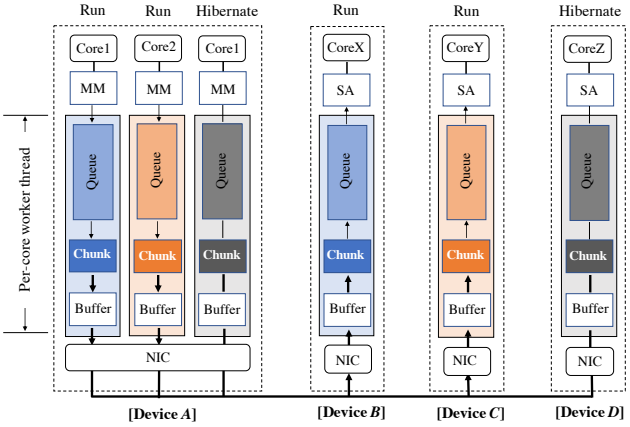

Fig. 5. Example of RDS: data from device A is swapping through RDS channel to device B or C. Resources are dedicated and data is batch processed in the channel.

Even though resource dedication improves swapping efficiency, unreasonable dedication can also bring new problems. For example, the performance of the local tasks may be affected if resources are occupied all the time. What is worse, the long-lived network link is unfriendly to energy efficiency. To solve these issues, MobileSwap hibernates and releases the dedicated resources if one swapping cycle is finished. Besides, MobileSwap evicts data in app granularity, as presented in the following subsections. A swapping cycle is defined as the

swapping process of one app. Thus, RDS channel goes into sleep mode until a new app swapping starts.

2) Page batching: Since pages delivered to one destination are maintained in the same queue, it is possible to batch them. Also shown in Fig. 5, the worker thread merges the pages in the queue to a huge page, which is named *chunk* in this paper, instead of sending the small pages to remote device one by one. The chunk size (Cs) is set as 64KB to align with the maximum packet size supported by common network protocols such as TSO (TCP Segmentation Offload). When arrived, the chunks are stored in the swap area. Pages are batch processed between swap and network stack for two reasons. First, if performing at the network stack, it would require MobileSwap to process each swap request individually to send to TCP buffer. Such a process method results in high CPU processing overheads [12]. Second, the operation is not performed in the swap stack either, since it requires fairly invasive kernel modification. In the process of page batching as well as chunk transmission, the memory page is maintained in the memory and indexed by a radix tree, which is supported by the original swap-cache mechanism [13]. The address space and PTE (Page Table Entry) information of the pages will not be updated before the swapping process is finished. If a page is demanded during the process, required pages can be directly obtained in the local memory. In this case, the interrupted swapping operation will be withdrawn.

In summary, based on RDS, data can be efficiently swapped across devices. It works well if the network is in good condition. However, cross-device I/O may not always be stable. The network could be disconnected. App crash could take place when disconnection occurs, caused by the failed response of swapped pages. Thus, this paper further proposes AAS, which is responsible to notify the impacted apps and minimizes app crashes.

C. AAS: App Aware Swapping

The basic idea of AAS is to swap the pages owned by one app to the same destination. MobileSwap indexes these pages based on the process virtual address space and corresponding virtual memory areas (VMA). The indexed pages are batch processed in the RDS channel and finally delivered to a remote idle device. In our design, MobileSwap prefers to swap the app to the device which has the most available memory space. When the remote device is disconnected, MobileSwap will kill the impacted apps (whose data were swapped to the disconnected device) proactively so that they can be cold launched without crashes. As a result, earlier swapped pages will not be demanded again. By sacrificing part of the app caching benefit, AAS gets rid of disconnection induced app crashes. Note that disconnection may not only occur when a swapped app is in the background, it may also occur when the app is running in the foreground. App can also crash in this phase if some pages are left in the disconnected device. And obviously, killing a foreground app and coldly launching it is unacceptable. To avoid this problem, MobileSwap swaps all pages of that app back when it is launched.

MobileSwap maintains an LRU to help determine which app to swap. The least-recently-used app will be selected as a victim. Thanks to RDS, the throughput is significantly improved. An app's pages can be swapped back in a short time. As evaluated, it takes 7%-16% longer to warm launch a swapped app. The speed is much faster than the cold launch (around 9x faster). In the case of huge apps, MobileSwap further sets a peak size Ps for each app. The swapped data of each app is not allowed to exceed the peak size. Besides network disconnection, MobileSwap ends apps' life cycle in three other conditions. First, the network has a connection problem. It could be caused by a weak wireless signal. Second, the network is occupied by user's other applications. MobileSwap avoids competing for the network resource. Third, the destination device becomes busy. In these cases, MobileSwap releases the rented memory on the destination device.

Based on AAS, MobileSwap benefits from renting memory from idle neighbor devices and returns the memory without negative impact to other applications.

D. Implementation and Discussion

In this section, the implementation of MobileSwap is presented with consideration of security and overhead issues.

MobileSwap is designed for multiple devices of the same owner, but can also be deployed on shared devices of multiple owners. Thus, MobileSwap provides a security interface, `ls_encrypt_fn`, in the implementation to protect user privacy. The data can be encrypted by calling the interface before swapping out. The private key is stored locally, and remote devices cannot access the swapped data without authorization. MobileSwap supports various encryption algorithms for mobile devices, including RSA and AES (Advanced Encryption Standard). To enable the security module, developers just need to register the algorithm in the interface. Then the registered algorithm can be executed. The encryption feature is turned off by default.

MobileSwap makes efforts to reduce overhead. The most critical overhead comes from energy consumption. Both CPU and network consume power, especially when these resources are dedicated. Thus, MobileSwap releases them and hibernates corresponding threads when no app swapping is performed. As evaluated, it consumes 0.3W-0.6W to swap an app out. The value fluctuated slightly depends on the data size of the swapped app and the signal intensity of the network. In comparison with the persistent energy consumption of touchscreen [14], MobileSwap induced consumption is negligible. Meanwhile, the memory consumption of MobileSwap is optimized in the implementation. Most of the MobileSwap induced memory consumption comes from the dedicated TCP buffers in the channel. The buffer should have enough space for receiving chunks but should ensure that most space of the buffer is utilized. Instead of applying for a fix-size buffer, MobileSwap improved the TCP's auto-tuning mechanism, so that the buffer space could be dynamically allocated and released with chunk granularity. As measured, sub-10MB space is required by MobileSwap in total. Meanwhile, the swap area

is dynamically allocated. MobileSwap adopts ZRAM's [10] method on RAM disk management. The swap partition will not occupy physical memory until there is actual data to store.

IV. Experiments

A. Experimental Platforms and Workloads

In the experiment, five Android-based mobile devices are used as the evaluation platform. One client and four servers. Client refers to the device obtaining swapping service, and servers refer to the devices providing remote swap area. The setup detail is presented in Table I. All devices are connected to the same WiFi network with a bandwidth of 1200 Mbps in dual-band (WS5200). To avoid bias, network functions except WiFi are closed. There are thirty workloads deployed on the client, as shown in Table II.

TABLE I
Experimental setup.

	Client	Servers
OS	Android 9.0 (kernel 4.9.0)	Android 9.0 (kernel 4.9.0)
CPU	Hisi Kirin 970 @ 8 Cores	4*cortex-A73 + 4*cortex-A53
Network	Wifi IEEE 802.11 b/g/n/ac	1000 Mbps Network Interface
Memory	6GB DRAM	6GB LPDDR4

TABLE II
Workload applications.

Category	Application
Social Network	Facebook, Skype, Twitter, WeChat, WhatsApp, LinkedIn, TikTok, Google+
Multimedia	MXPlayer, Youtube, Netflix, Snapchat, Youku
Mobile Game	Angry Birds, Arena of Valor, Temple Run 2, Fruit Ninja, Asphalt 8
Commerce	Amazon, AliPay, BOA, Ebay, Taobao
Business Utility	Chrome Browser, Camera, Google Map, Lyft, BBC News, OfficeMobile, Evernote

In the experiments, five schemes are implemented and measured as a comparison.

NoSwap: This is the baseline of the evaluation. It disables all swap features in the system.

LocalSwap: A swap partition in local is built. The memory data is written to Flash when memory under pressure.

RemoteSwap: The swapping scheme is similar to LocalSwap, except that partitions (Linux swap files) are built remotely.

CloudSwap: This work realizes CloudSwap [5]. Some pages are swapped to the cloud, which is simulated by a server in this experiment, while some pages are swapped to local.

MobileSwap: This is the proposed cross-device swapping system, which includes RDS and AAS.

B. User Experience

As aforementioned, local swapping induces a lot of read-write conflicts. It significantly increases apps' read latency. Unlike write requests, read requests are latency critical, and slowing down the read access is likely to have a significant performance impact [15]. In mobile system, the user experience will deteriorate when suffering read-write interference. So it is critical to minimize the swap induced access interference. The read latency of each app is measured when enabling MobileSwap. As shown in Fig. 6, the average read latency when enabling MobileSwap is 469us, which is 1.1x of NoSwap, and reduced by 53.7% on average compared to LocalSwap.

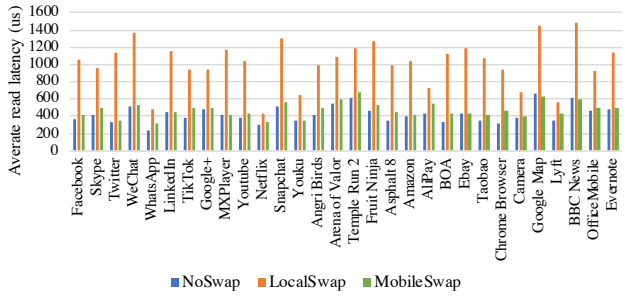

Fig. 6. The read latency of mobile apps when enabling LocalSwap and MobileSwap. The read-write conflict is significantly reduced via remote I/O.

C. Efficiency and Stability Improvement

The app caching benefit is further evaluated. In the evaluation, the pre-installed apps are repeated launched for ten rounds to create memory pressure. The order of app launch changes randomly in each round. The upper bound of swap partition size for LocalSwap, RemoteSwap, and MobileSwap is set to 2GB. For fairness, CloudSwap maintains a 1GB local partition and a 1GB remote partition.

As shown in Fig. 7-(a), the number of cached apps with MobileSwap is 14 on average. It is 1.4x and 1.17x of RemoteSwap and CloudSwap respectively. The benefit is different for different schemes even with the same swap partition size. This is because many pages are not swapped to the partition efficiently. When the speed of memory reclaiming is slower than memory needs, the memory pressure keeps increasing. The system will finally relieve memory pressure by killing apps. Thus, the app caching benefit is directly affected by swapping efficiency. The efficiency of memory swapping is further measured. As shown in Fig. 7-(b), the average throughput of MobileSwap is 828Mbps. It is 2.3x of RemoteSwap. The throughput of MobileSwap is even higher than LocalSwap (781Mbps/s). It is due to the coordination of RDS and AAS, which avoids a large quantity of small I/Os and performs swapping more aggressively. In comparison with CloudSwap, MobileSwap improves by 1.72x. CloudSwap's hybrid architecture weakens the batch processing capability of the system to the swapped data.

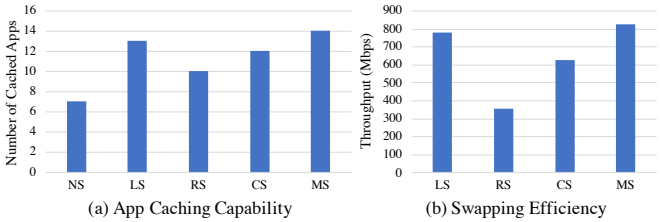

Fig. 7. Swapping efficiency and app caching capability comparison between NoSwap (NS), LocalSwap (LS), RemoteSwap (RS), CloudSwap (CS), and MobileSwap (MS).

Based on the above experiments, we disconnect the network manually. Then switching the apps to the foreground. To be close to the real usage scenario, each switched app is used for two minutes. During the experiments, six apps crashed with RemoteSwap, and four apps crashed with CloudSwap. Zero app crashing is detected with the proposed system. It is because all apps impacted by this network outage are proactively killed ahead of time. After the network is disconnected, six apps can be warm launched, which is close to the app caching capability before enabling MobileSwap.

D. Sensitive Study

To further understand MobileSwap, several sensitive studies are conducted. The app caching benefit is measured with the change of swap partition's size and number. As illustrated in Fig. 8, the number of cached apps when enabling MobileSwap with two servers, each share 1GB memory to the client, is equal to the [1 server, 2GB] and [4 servers, 512MB] case. Then, the servers are brought off-line one by one to observe the robustness of MobileSwap. Results show that the number of cached apps decreased gradually with the reduction in servers. More apps cold launched, but no app crash occurs.

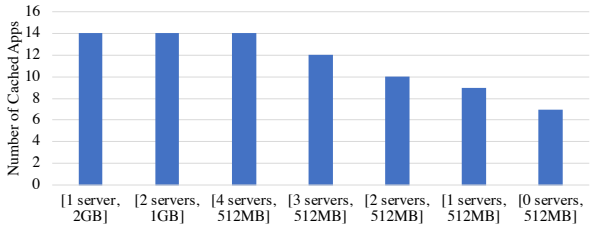

Fig. 8. The number of cached apps changes with the changing of partition size and number.

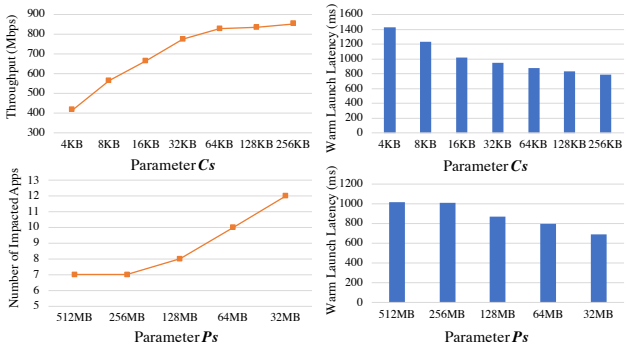

Fig. 9. Parameters tuning. With the increase of Cs, the swapping throughput and launch speed increased. With the decrease of Ps, more apps are impacted by disconnection, but launch speed increased.

The value of two parameters is tuned in the evaluation. First, the chunk size Cs in RDS. Second, the allowed peak size Ps of each swapped app in AAS. Cs affects efficiency. Ps affects the number of disconnection impacted apps. And both parameters affect the warm launching speed. As shown in Fig. 9, with the increase of Cs, the swapping throughput increased. As a result, the timing cost to warm launch a swapped app decreases. On the other hand, the number of impacted apps increased with the reduction of Ps. More apps are swapped out to relieve memory pressure. The positive impact is that app launching speeds up. An interesting phenomenon is that the slope smooths out with the increase of Cs and Ps. Their values are set as 64KB and 128MB by default. These parameters of MobileSwap are configurable as the optimal value may differ on different hardware infrastructure.

V. Conclusion

This paper presents MobileSwap, a cross-device memory swapping scheme for mobile devices. By making use of remote I/O, the memory pressure can be relieved without interfering with the local I/Os. MobileSwap demonstrates significant performance gain compares with state-of-the-art techniques.

References

[1] K. Zhong, D. Liu, L. Liang, X. Zhu, L. Long, Y. Wang, and E.H. Sha. Energy-efficient in-memory paging for smartphones. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10): pp.1577-1590, 2015.

[2] X. Zhu, D. Liu, K. Zhong, J. Ren, and T. Li. SmartSwap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference*, pp.1-6, 2017.

[3] R. Liu, X. Chen, Y. Tan, R. Zhang, L. Liang, and D. Liu. SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp.966-975, 2020.

[4] G. Juncheng, L. Youngmoon, Z. Yiwen, C. Mosharaf, and G. S. Kang. Efficient Memory Disaggregation with INFINISWAP. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp.649-667, 2017.

[5] D. Chae, J. Kim, Y. Kim, J. Kim, K. A. Chang, S. B. Suh, and H. Lee. CloudSwap: A cloud-assisted swap mechanism for mobile devices. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp.462-472, 2016.

[6] M. Shafi, A.F. Molisch, P.J. Smith, T. Haustein, P. Zhu, P.D. Silva, and G. Wunder. 5G: A tutorial overview of standards, trials, challenges, deployment, and practice. *IEEE journal on selected areas in communications*, 35(6): pp.1201-1221, 2017.

[7] C. Funai, C. Tapparello, and W. Heinzelman. Enabling multi-hop ad hoc networks through WiFi Direct multi-group networking. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pp.491-497, 2017.

[8] C. Li, L. Shi, L. Yu, and C.J. Xue. SEAL: User Experience-Aware Two-Level Swap for Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.4102-4114, 2020.

[9] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng. MARS: Mobile Application Relaunching Speed-Up through Flash-Aware Page Swapping. In *IEEE Transactions on Computers*, 65(3): pp.916-928, 2015.

[10] Nitin Gupta. ZRAM Project. Linux Foundation, San Francisco, CA, USA. https://www.kernel.org/doc/ Documentation/blockdev/zram.txt.

[11] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, TCP = RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp.127-140, 2020.

[12] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[13] Understanding the Linux Kernel, https://www.oreilly.com/library/view/understanding-the-linux/0596002130/ch16s03.html, 2020.

[14] A. Carroll. Understanding and Reducing Smartphone Energy Consumption. University of New South Wales. 2017.

[15] M.K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 the Sixteenth International Symposium on High-Performance Computer Architecture*. pp.1-11, 2010.