iSwap: A New Memory Page Swap Mechanism for Reducing Ineffective I/O Operations in Cloud Environments

ZHUOHAO WANG, Beihang University, China
LEI LIU*, Beihang University, China
LIMIN XIAO, Beihang University, China

This paper proposes iSwap, a new memory page swap mechanism that reduces the ineffective I/O swap operations and improves the QoS for applications with a high priority in the cloud environments. iSwap works in the OS kernel. iSwap accurately learns the reuse patterns for memory pages and makes the swap decisions accordingly to avoid ineffective operations. In the cases where memory pressure is high, iSwap compresses pages that belong to the latency-critical (LC) applications (or high-priority applications) and keeps them in main memory, avoiding I/O operations for these LC applications to ensure QoS; and iSwap evicts low-priority applications' pages out of main memory. iSwap has a low overhead and works well for cloud applications with large memory footprints. We evaluate iSwap on Intel x86 and ARM platforms. The experimental results show that iSwap can significantly reduce ineffective swap operations (8.0% - 19.2%) and improve the QoS for LC applications (36.8% - 91.3%) in cases where memory pressure is high, compared with the latest LRU-based approach widely used in modern OSes.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: cloud computing, page swapping, QoS

ACM Reference Format:
Zhuohao Wang, Lei Liu, and Limin Xiao. 2023. iSwap: A New Memory Page Swap Mechanism for Reducing Ineffective I/O Operations in Cloud Environments. 1, 1 (February 2023), 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Today's computer systems often have large-capacity memory and high-performance storage (e.g., HDD, NVMe). To tackle the performance gap between main memory and storage, the common wisdom is to improve the performance of I/O devices or to reduce disk I/O operations [31,32,40,41]. However, we find that the existing memory swap mechanisms (in the context of 4KB pages in this work) in modern OSes often blindly swap out hot/active pages (i.e., the frequently used pages), leading to unnecessary swapping in/out data pages across main memory and storage, hindering the overall system performance. These swap operations are ineffective. Moreover, in cloud, we further find that lots of pages belonging to the latency-critical (LC) services [14,24,38] or applications with a high priority are swapped out together with the pages belonging to best-efforts (BE)/low-priority applications [14,24,38] with no differences. The existing swap approach evicts the pages without the knowledge of the application features, severely impacting the QoS of LC applications. Taking the Linux kernel with version 5.10 as an example, we find that though the swap mechanism (e.g., LRU, CLOCK [4,20,23,27])

*Corresponding author (PI): lei.liu@zoho.com; liulei2010@buaa.edu.cn. Authors' address: Beihang University, No.37 Xueyuan Road Zhongguancun, Haidian District, Beijing, China, 100191. Sys-Inventor Lab - https://liulei-sys-inventor.github.io.

works in some cases, a large number of hot pages are frequently swapped out of the main memory at run time. Therefore, when applications want to use these pages again, OS has to load them from hard disks to main memory. These I/O overheads lead to performance/QoS slowdown for many applications [11,21,40], especially the cloud applications. For instance, in the cases where Redis in YCSB [15] has a memory footprint ranging from 2.5 GB to 30 GB on a cloud server with 128 GB main memory, 30.7% of the pages suffer swap out, and the hot pages that will be immediately reused account for 45.8% of all swapped pages. The LC services are widely used in cloud environments and are more sensitive to swap I/O operations [14,24,40]. In some cases where many applications are co-located on a specific cloud server, the existing swap mechanism swaps out memory pages belonging to LC services or applications with a high priority, but lots of the pages that belong to BE applications or applications with a low priority are remained in the memory, negatively affecting the QoS. The community is looking forward to a new design for the I/O swap mechanism.

To this end, we propose iSwap (intelligent Swap) in the OS kernel, a new mechanism that can conduct effective memory swaps and avoid incorrectly swapping the will-be-used memory pages. Besides, in the cases where memory pressure is high, iSwap compresses the pages belonging to the LC applications/high-priority applications and keeps them in the main memory, and evicts the pages that belong to the low-priority applications. By doing so, LC services or applications with a high priority can avoid performance loss caused by swap I/O operations, benefiting the system QoS. During the run time, iSwap monitors the candidate pages marked as to be swapped by using the present bit in PTEs (Page Table Entry) [4,5] and then makes the swap decisions according to the logic page-level reuse time, which has the identical reuse characteristic and behavior to the micro-architecture level page reuse information, obtained by monitoring the access bit in PTE and a learning-based approach. Then, iSwap doesn't use the LRU-based swap list, and it has a new design - a priority-based swap list, which reorganizes the swap list according to application priority (e.g., LC services have a higher priority). iSwap swaps pages in BE/low-priority list first when memory pressure is high, keeping pages belonging to LC services in memory and avoiding the unnecessary I/O thrashing for high-priority applications. Additionally, for LC application pages, iSwap has a compressing cache in main memory using zswap and compresses pages leveraging zbud [9]. Via this method, iSwap supports up to two compressed pages per page frame in memory, further reducing the number of pages that need to be swapped out. Compared with the existing swap mechanism in OSes (e.g., Linux, FreeBSD) [3,4,34,41], iSwap is more sensitive to memory pattern changes and can accurately learn the memory pages' reuse patterns, avoiding the ineffective memory swaps. Moreover, iSwap improves the QoS for LC applications and the overall system.

To sum up, this paper makes the following contributions.

(1) We show that the widely used page swap approach blindly swaps many to-be-used hot pages out of main memory, especially in cloud computing, where the cold and hot memory regions in small sizes are interleaved in the address space. We further show that these ineffective I/O operations frequently happen, leading to performance losses and QoS slowdown. We find that the reasons behind this phenomenon are multi-fold. First, the features of memory pages are not effectively identified; thus, the swap decisions are often based on inaccurate information. Second, the core logic (e.g., second chance LRU approach [3,4,20]) for managing active/inactive pages might be ineffective; therefore, the promotion/demotion routine for active pages often incurs errors.

(2) We show that the candidate to-be-swapped pages belonging to applications are interleaved in the swap cache in OS. And the existing swap approach does not know which application a specific page belongs to. So, it often blindly swaps some pages from the LC or high-priority applications in practice out of the memory, but the pages that belong to low-priority or BE applications remain in memory. This behavior negatively affects the QoS, as the required pages for LC or high-priority applications have to suffer I/O swaps.

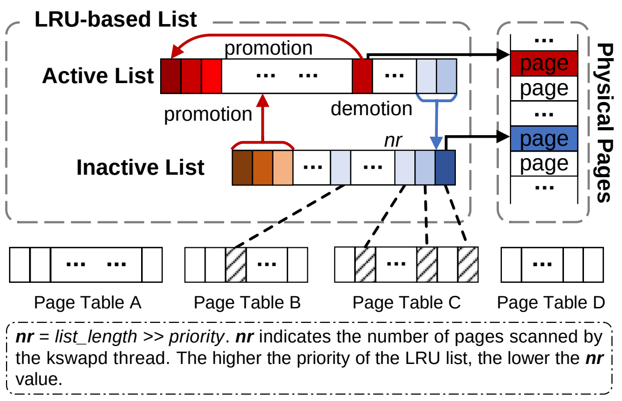

LRU-based List
Active List
promotion
promotion
demotion
nr
Inactive List
Physical Pages
page
page
page
page
Page Table A
Page Table B
Page Table C
Page Table D
nr = list_length >> priority. nr indicates the number of pages scanned by the kswapd thread. The higher the priority of the LRU list, the lower the nr value.

Fig. 1. The LRU-based page reclamation/swap mechanism [4].

(3) We design iSwap, a new swap mechanism in the OS kernel that conducts effective memory swap and avoids incorrect swap operations. iSwap has a new two-phased monitoring mechanism that locates the memory swap regions and learns the logic reuse patterns for the swap candidate pages. iSwap conducts effective swaps based on learned reuse patterns. Furthermore, using memory compression technology (i.e., compressing cache in memory), iSwap makes the best effort to ensure that high-priority/LC services' pages remain in memory, improving the system QoS in cloud environments.

(4) We implement iSwap in the Linux kernel with version 5.10, and we show that iSwap performs well and exhibits a low overhead for cloud applications. The experimental results show that iSwap outperforms the widely used swap approach in Linux. iSwap reduces the ineffective I/O swaps by 48.9% (up to 66.8%) on average, and reduces the 99th percentile response latency by 70.6% (up to 91.3%) for LC cloud services, on average.

2 BACKGROUND

2.1 Swap Operations

The main memory and storage systems are interactive. When a page fault occurs, the handler in the OS kernel moves the required pages from the hard disk (storage) into the main memory. During this process, the application is stalled until the OS's page fault handler completes the swaps and updates the page table accordingly. In terms of swap out, the memory capacity is not infinite, so OS needs to evoke the page reclamation routine to free pages until the number of available pages reaches the high watermark [8]. For example, in Linux, *kswapd* scans the processes and tries to swap out some cold/inactive pages in the cases where the watermark is low. However, when these pages are required again, OS has to move them from the hard disk to memory (i.e., I/O happens), leading to performance losses in many cases (especially for LC applications) [11,13,14,34].

2.2 The LRU-based Page Swaps

Modern OSes often use LRU-based page swap algorithms [4,20,34]. Taking Linux as an example, it has a hot/cold page classification mechanism, and it has two LRU-based lists, i.e., the active_list and inactive_list. The active_list contains pointers to active pages (i.e., hot pages) for all processes, and the inactive_list contains pointers to cold pages that might be reclaimed and swapped out of memory. OS uses PG_active bit in the page to show whether the page is in active_list or inactive_list [4,5] (the value of 1 indicates the page is in active_list). OS promotes and demotes pages according to the pages' hot/cold features as shown in Figure 1. OS determines whether the page is hot or cold based

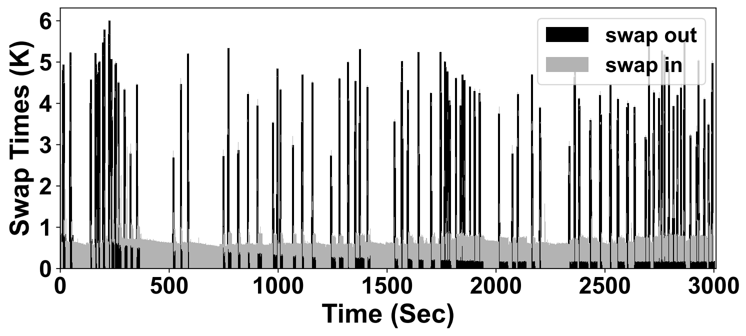


Fig. 2. Page swap in/out times during 3000s for Redis (30 GB).

on its current value of access_bit (the value of 1 indicates the page is accessed) in the PTE [4,5]. In Linux, a page is marked as cold if the access_bit is 0 in two consecutive samplings (second chance algorithm [3,4]). Figure 1 shows a typical LRU-based design in Linux. When the memory pressure of the system is high, the kernel calls kswapd to scan LRU lists and reclaim the cold pages. kswapd reclaims the pages from the bottom of the inactive_list, as illustrated in Figure 1. This reclamation routine uses reverse mapping to find the PTE of the page in a specific list. If a page is considered cold (i.e., the access_bit is 0 in two consecutive samplings), OS will swap it out of the main memory. This mechanism might work well in some cases, but it does not consider the reuse patterns for memory pages and, therefore, cannot handle the cases when memory behaviors are random or exhibit frequent changes in practice. For instance, as long as the access_bit is 1, the pages accessed 1 time and 1,000 times are considered as active pages without any difference. So, some hot pages, like the page reused many times, might be put to the bottom half of the list and then demoted to the inactive_list. Therefore, they become the candidates for swapping out. Notably, the page swap thread only scans a portion of the address space for a specific application [4] instead of the global address space. And, the existing approach uses a second chance algorithm, which only has pages' access patterns in a short history. Therefore, the existing approach inevitably increases the possibility of swapping out the hot pages.

2.3 Swap Cache

Modern OS often has a memory pool between the LRU swap list in Figure 1 and the hard disk, e.g., the swap cache in Linux [4,9,41,42]. When swap operations happen, the to-be-swapped pages in inactive_list in the LRU list will be first moved to the swap cache and then evicted into the hard disk via the I/O system. The LRU-based swap list in Figure 1 and the swap cache have the memory pages from all applications without knowledge of applications' features, e.g., the priority and which applications are more sensitive to I/O swaps. For example, on a specific cloud server using Linux, we find that both LC and BE applications' pages are in the swap cache; a page belonging to the LC application and a page belonging to BE application are considered swap-out candidates without any difference, i.e., these pages will be swapped out of main memory to disk with equal probability. Thus, in practice, we often find that the existing OS blindly swaps LC applications' pages to disk but keeps pages belonging to BE applications in memory, leading to severe QoS slowdown for LC applications.

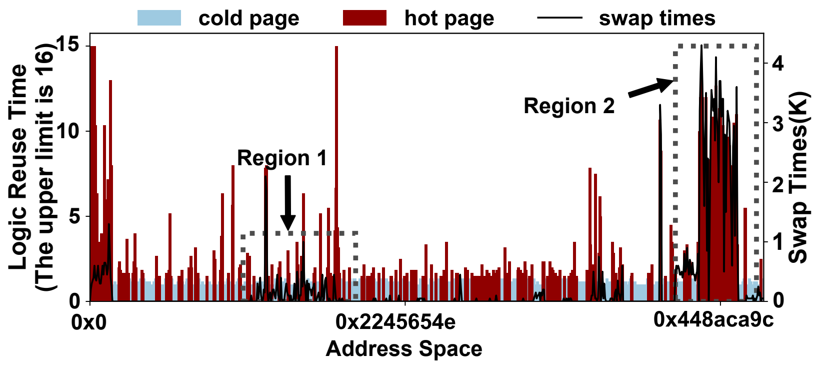


Fig. 3. Page reuse and swap times. Cold pages' logic reuse time is 0 or 1 in this experiment based on empirical studies [4,20,25,36].

3 MOTIVATION

3.1 Ineffective Swap Operations are Happening

To study how the swap mechanism performs in practice, we run Redis (30 GB), Memcached (30 GB), and MySQL (30 GB) in YCSB [15,16] on a typical off-the-shelf cloud server with 128 GB DDR4 main memory and use *vmstat* [33] to monitor the swap in/out pages. The OS is Linux with kernel version 5.10. Figure 2 shows that the page swap in/out times within 3,000 seconds for Redis. This figure shows that page swaps always happen, and we further find that the swapped-out pages contain many pages that will be used pretty soon, so OS has to swap them into the main memory repeatedly. MySQL and Memcached exhibit similar phenomena. More details are in the following paragraphs.

Figure 3 further shows Redis' average page-level reuse and swap times within 30 minutes. Reuse can reflect the page's hot/cold features, and the pages with higher reuse times can be hotter (i.e., more active) [25,35,37]. To obtain the page-level reuse times, we use a loop to check the pages' access_bits [25,37] (details are in Sec. 4). As the loop's control number is 16 in each sampling point (refer to Sec. 4), the upper bound for reuse times is 16. The x-axis shows the address space for Redis. The left y-axis shows the logic reuse times and the right y-axis shows the times of swap operations for memory pages. We can see that many frequently used hot pages always suffer swaps out, and these swap operations happen frequently. As illustrated in Region 2 in Figure 3, the active pages that will be reused are swapped out thousands of times (at the peak). Moreover, we can see that swaps frequently happen in the regions where many hot pages and cold pages are interleaved with each other (e.g., Region 1 in Figure 3).

To study the underlying reasons, we track the page-level reuse patterns for these pages that are frequently swapped out. For a specific page, we monitor the access_bit in PTE with a loop manner (clear and sample access_bit repeatedly) to obtain the page-level logic reuse pattern [25,35,37]. In a specific sampling period, a page exhibiting 11011 reuse pattern is more active (hotter) than the page that has 01000 reuse pattern. Figure 4 shows the reuse pattern for a specific page in Redis' address space. We can see that reuse pattern always changes tremendously. Therefore, the page swap decisions based on the momentary state of the access_bits (e.g., Linux, Free BSD) are unreasonable. The widely used LRU-based swap mechanisms (e.g., second chance algorithm in Linux, Free BSD [3,4]) for swapping out cold pages clear and check the access_bits at the present moment. However, by doing so, as long as the recent value of access_bit is 0, the page is marked as cold (swap candidate) without the consideration of its real footprints. Therefore, some will-be-used hot pages that are with diverse reuse patterns are considered as cold ones and swapped out of memory frequently. However, they are needed in pretty soon, and thus these ineffective swap out/in operations happen all the time as a

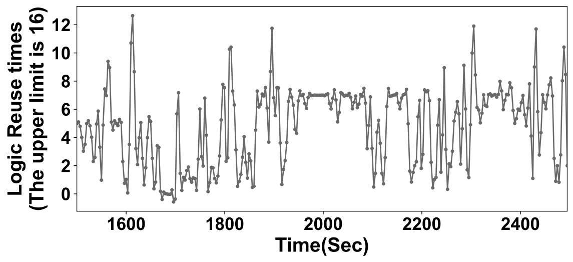


Fig. 4. The reuse times of a specific page in Redis' address space.

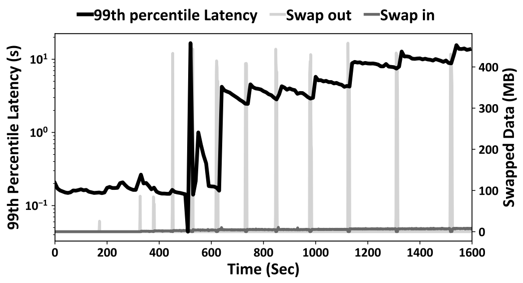


Fig. 5. Page swap times and Redis' 99th percentile latency.

result. Besides, even using the design like swap cache in OS [4,26], the ineffective swaps cannot be eliminated. This design employs lazy reclaiming technology to avoid ineffective page swaps. Yet, it cannot avoid them at the root, as the pages in the swap cache are from the LRU-based swap mechanism (without the considerations of reuse patterns) and will be flushed into the disk periodically.

3.2 Swap Operations Negatively Affect QoS for LC Services

A cloud server may have many co-located LC (latency-critical) and BE (best-efforts) services (e.g., apps in Table 1) at the same time [14,24,38]. The existing OS swap approach does not leverage the knowledge of the application features. As a result, it often blindly swaps pages from the LC or high-priority applications in practice. These applications are more sensitive to I/O operations. Yet, pages belonging to BE or low-priority applications remain in the main memory. These applications are not that sensitive to I/O swaps. In a sentence, the existing swap approach leads to QoS slowdown for LC applications. To show this phenomenon, we run a workload (i.e., hybrid workload 1 in Table 2) that has LC and BE applications on a typical cloud server with 128 GB DDR4 memory (Sec. 6). The LC application is Redis with 40 GB memory footprint. The BE applications involve Fluidanimate (5 threads) [6] with 20 GB memory footprint and Streamcluster [6] (10 threads) with 10 GB memory footprint. We monitor *VmSwap* in *proc* filesystem [7] to obtain the swap in/out information for each application. Figure 5 shows how existing Linux swap mechanism performs on Redis. Similar to the previous studies, the y-axis shows its 99th percentile response latency that reflects the QoS of LC application, and the y-scale of latency is logarithmic [14,24,38].

In Figure 5, from time point 0 to 500, as the memory pressure is low, OS swap operations rarely occur. During this period, Redis' response latency is low and relative stable (i.e., QoS is acceptable). After time point 500, Redis memory footprint increases, and OS starts to swap out pages (including the pages belonging to both LC and BE apps as mentioned before) to the hard disk. As discussed before, the swap operations blindly evict many to-be-used hot pages to disk, and the LC service's pages (with high priority) are evicted together with BE applications' pages. LC services are more sensitive to I/O swaps, so the swaps should be used carefully. When these swapped-out pages are needed again, the OS kernel has to migrate the required pages from the hard disk into the main memory. Redis service is stalled during this period until the kernel completes the swap operations. Therefore, we observe clearly that the response latency of Redis increases significantly. In Figure 5, the 99th percentile response latency increases from 0.15s to 1.2s.

We conclude that existing OS swap mechanisms do not distinguish if the to-be-swapped pages belong to LC/high-priority or BE/low-priority applications, leading to some pages that belong to LC/high-priority applications being swapped to disk. Such swap behaviors cause the QoS of LC/high-priority applications, which are more sensitive to I/O

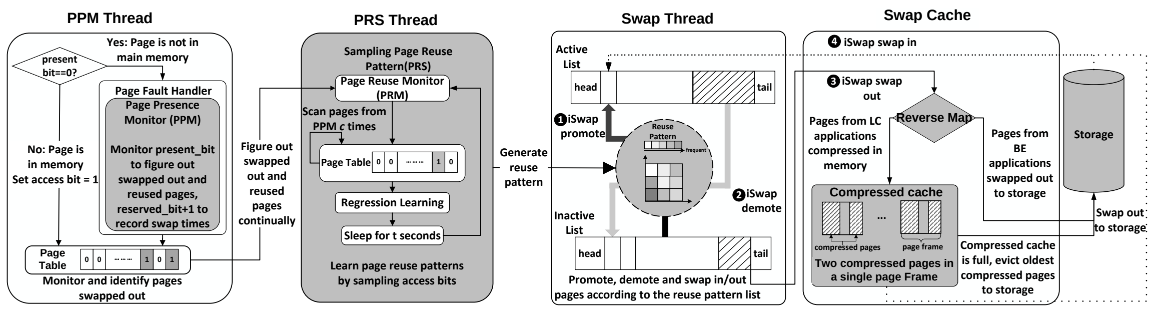

PPM Thread
present bit==0?
Yes: Page is not in main memory
Page Fault Handler
Page Presence Monitor (PPM)
Monitor present_bit to figure out swapped out and reused pages, reserved_bit+1 to record swap times
No: Page is in memory Set access bit = 1
Page Table
Monitor and identify pages swapped out
Figure out swapped out and reused pages continually
PRS Thread
Sampling Page Reuse Pattern(PRS)
Page Reuse Monitor (PRM)
Scan pages from PPM c times
Page Table
Regression Learning
Sleep for t seconds
Learn page reuse patterns by sampling access bits
Generate reuse pattern
Swap Thread
Active List
head
tail
1 iSwap promote
Reuse Pattern
frequent
2 iSwap demote
Inactive List
Promote, demote and swap in/out pages according to the reuse pattern list
Swap Cache
4 iSwap swap in
3 iSwap swap out
Reverse Map
Pages from LC applications compressed in memory
Pages from BE applications swapped out to storage
Compressed cache
compressed pages
page frame
Two compressed pages in a single page Frame
Compressed cache is full, evict oldest compressed pages to storage
Storage
Swap out to storage

Fig. 6. iSwap in a nutshell. The four components are kernel threads and they work in parallel. According to our experiments, the loop control number c is 16 and the sleep time t is 3 seconds in PRS. These empirical thresholds can be adjusted if needed. The highlighted parts in the dark are new designs in the OS kernel.

operations – low-speed disk, to be negatively affected.

To sum up, the current OS swap mechanism we use daily on cloud servers has two challenges to address carefully. *(1) Ineffective swaps are happening, leading to hot pages that will be used soon being swapped to disk frequently. (2) Swap operations hinder QoS for LC services because pages belonging to diverse applications are blindly swapped without distinction.*

4 THE DESIGN OF ISWAP

4.1 iSwap in a Nutshell

To this end, we propose iSwap (intelligent Swap), a new memory swap mechanism in OS that learns to avoid ineffective swaps that evict to-be-used hot pages and high-priority pages out of main memory. As illustrated in Figure 6, iSwap has four key components that work in a pipe-lined way. (1) A Page Presence Monitor (PPM) that captures the page-swapping information. (2) A Page-level logic Reuse pattern Sampling (PRS) component that learns whether a specific page will be used. (3) A new page-swapping component that manages the pages based on the reuse patterns and conducts swaps that avoid moving hot and will-be-used pages out. (4) A new swap cache component that distinguishes applications' pages and applies different swap strategies – compress LC applications' pages in the main memory and swap out BE applications' pages to hard disk. iSwap considers the page-level logic reuse patterns and applications' priorities when making swap decisions.

iSwap can effectively handle cases where the applications exhibit random memory patterns, e.g., the address space has many interleaved small segments mixed with hot and cold pages. Moreover, iSwap can make the best efforts to ensure high-priority/LC services' pages are kept in memory and swaps out low-priority/BE applications' pages to free memory when available memory is insufficient. iSwap improves the system performance and QoS in cloud environments. More details of our design are in the following.

4.2 Monitoring the Page Presence (PPM)

We design PPM to monitor the page presence and to locate the memory regions that suffer swaps. PPM is an online memory page presence monitoring module in OS kernel (Figure 6). For a specific page, it monitors the present bit in PTE. The value of 1 indicates the page is in the main memory, and 0 indicates the page is not in the main memory. When a program accesses a page, OS tries to find its PTE in the page table and then checks the value of the present bit in the

PTE. If the present bit is 0 (not in main memory), the page fault handler in OS will be triggered. PPM is in the page fault handler, and PPM starts to work when the handler is triggered. The page fault handler calls the do_swap_page() function to move data pages into the main memory and creates the PTEs for these pages. At this time, PPM records that this page has been swapped in/out, adds 1 to the value kept in reserved bits 52-55 in its PTE and sets the present bit to 1 simultaneously. By doing so, during the run time, PPM marks all of the pages that have been swapped. Notably, the value in the reserved bits indicates how many times a specific page is swapped. Using PPM, OS can learn the memory regions that are suffering swaps in the application's address space. We will investigate the reuse patterns for the pages in these memory regions.

4.3 Learning Page-level Reuse Patterns (PRS)

We design PRS to learn the pages' reuse patterns in the memory regions produced by PPM that suffer swaps. PRS is also in the OS kernel and works cooperatively with PPM (Figure 6). Rather than random sampling approach, iSwap locates the memory regions where swap-in/out operations happen, avoiding blind scan and reducing sampling overhead. Pages in these memory regions are more likely to be swapped again. iSwap uses an auxiliary hash table to store the page reuse pattern. Each page has an entry in the auxiliary hash table. The auxiliary hash table has minor impact on the memory usage (about 9 MB data for every 1GB swapped pages). After having the swap memory regions, PRS monitors the access_bits in PTEs for these pages in swap regions and counts the reuse patterns for the pages in a loop manner (clear and examine access_bit in a loop). PRS clears the access_bits for the pages in the memory regions (i.e., set the access_bits to 0 using pte_mkold()), then examines these PTEs. The pages whose access_bits are reset to 1 are marked as reused. Using a loop, PRS can have these pages' logic reuse times (reuse patterns). PRS sets the accumulative values in the corresponding hash item within hash table. For example, if the loop control number is 200 in PRS, a page that exhibits 150 on the accumulative value of access_bit has more possibilities for reuse than the page with 15 on that value.

Previous studies [25,26,37] show that 200 loops can accurately reflect the logic page-level reuse times (R) for applications, including Memcached, Redis, and MySQL, with a large memory footprint in practice, but the sampling overhead cannot be ignored [26,37]. Reducing the learning overheads is key to making this technology practical. This paper aims to reduce the sampling overheads without losing accuracy. We reduce the number of loops step by step and finally find that using a regression-based estimation approach when the loop control number (l) is 16, PRS can have approximate results (R) to the cases where the loop control number is 200. We define the regression function as $R/200 = r/l$ (l is the number of loops, and r is the reuse times for a specific page after l loops). So, we can get R when r and l (16) can be obtained with low overhead. After reducing the number of loops to 16, the time overhead of sampling is significantly reduced by 79.6%. For instance, scanning 30 GB of memory that suffers swaps and getting their reuse pattern, iSwap only takes around 2 seconds. Moreover, as modern cloud servers often have tens of computing cores, OS occupies a few of them for learning/sampling during the run time is practical.

4.4 New Page Swap Mechanism

We introduce a new swap system with two-tired reuse-based page active/inactive lists. Figure 6 shows the details. The active list contains the active pages (i.e., pages that might be touched in pretty soon according to their reuse patterns), and the inactive list contains the candidate pages for swapping. Compared with previous swap mechanisms based on the LRU strategy, iSwap re-organizes memory pages according to the reuse patterns obtained by PRS and finds the swap candidates accordingly. The pages that might not be used according to reuse patterns will be swapped out of the main memory. Note that though the page is swapped out, iSwap still keeps its reuse information in its PTE. A page that might be accessed soon will be placed in the active list. Otherwise, it will be in the inactive list. Pages in each list are sorted by their reuse values.

The active list and inactive list are reorganized according to reuse patterns. When a program attempts to use some pages that are not in the main memory, the page fault handler is triggered to swap in these pages (step ❹ in Figure 6). At this time, PPM is evoked to locate the memory regions that suffer swaps, and PRS learns the pages' reuse patterns. If the page's reuse patterns show that the pages will be used (hot), it will be kept in the active list. If the page is in the inactive list, it will be promoted to the active list and be placed in an appropriate position of the active list according to its reuse value (i.e., step ❶). In terms of page eviction in inactive list, as shown in ❸, when memory reclamation is evoked, iSwap takes the nr pages from the tail of the inactive list (i.e. nr is the number of pages to be swapped to swap cache, calculated as $nr = list\_length(LRU) >> p$, where p is priority of list [4]). In our design, the reuse value > 1 in 16 loops (Sec. 4.3) can be considered as will-be-used pages and should not be swapped out. Otherwise, the pages are considered cold pages that will not be used in the near future. Finally, iSwap migrates these pages to the swap cache (step ❸ in Figure 6). The length of the inactive list is about 2/3 of the length of the active list (i.e., 2/3 is an empirical value, consistent with the Linux OS), and iSwap keeps the ratio of two lists during run time. So as shown in step ❷ in Figure 6, iSwap demotes n pages from the active list to the inactive list according to reuse patterns (i.e., n is the number of pages to be demoted to the inactive list, calculated as $n = inactive\_length - 2/3 \times active\_length$).

4.5 Swapping vs. Compressing

After the reuse-based swap mechanism, we design a new swap cache mechanism (the last part in Figure 6) that caches pages before finally moving to the hard disk. After the swap thread migrates inactive pages out of the main memory (Sec. 4.4), these pages will be first moved to this new swap cache, as illustrated in Figure 6. The new swap cache has different policies on handling pages from LC/high-priority applications or BE/low-priority applications, respectively. Since LC/high-priority applications are more sensitive to I/O operations, while BE/low-priority applications (e.g., batch services.) are not, blindly swapping out pages belonging to LC applications together with BE applications' pages like existing swap mechanisms leads to QoS slowdown for the LC applications (Sec. 3.2). iSwap works intelligently and handles multiplicity by using Multi-policy. iSwap keeps LC/high-priority applications' pages in memory and compresses them when memory pressure is high, and iSwap evicts pages belonging to BE/low-priority applications instead of other high-priority applications' pages.

For a specific page, iSwap uses reverse mapping to identify the priority of the application to which the page belongs, and then swaps pages accordingly. In our design (referring to Linux), each process has a value indicating the priority. The lower value indicates a higher process priority. We use a priority threshold for running applications. Applications with a priority below the threshold are considered high-priority applications. The priority threshold can be adjusted if needed. iSwap identifies applications' priority in the swap cache before moving pages to the hard disk.

Memory Compression. We have two considerations for the pages belonging to LC/high-priority applications. (1) LC applications are more sensitive to swap operations. Swapping their pages to the hard disk will negatively affect the QoS. So, we want to keep them in memory. (2) When the swap thread starts to reclaim pages, it indicates the memory pressure of the system is high, and OS needs more free memory for running applications, especially for the LC applications. Using zswap [9], we create a compressing cache in the main memory for LC and high-priority applications. The capacity of the compressing cache is 10% of the total memory, and it can be adjusted manually. The compressing cache uses zbud allocator, which can store two compressed pages in a specific page frame [9]. Figure 6 illustrates how it works. When iSwap starts to evict pages, it compresses the pages belonging to LC applications, saving memory space for the system and avoiding I/O swaps. When LC applications need these pages again, iSwap decompresses them without involving I/O operations. In addition, if the compressing cache is full, iSwap decompresses and evicts the oldest compressed pages to the hard disk. Compressing cache is only available for LC/high-priority applications. iSwap evicts other applications' pages according to the reuse patterns only as mentioned before.

4.6 iSwap Works in OS Kernel

In summary, the four components in iSwap mentioned above work in parallel as independent threads in the kernel. Algorithm 1 shows the overall logic of iSwap. In practice, iSwap can obtain the page-level logic reuse patterns with low overhead and effectively avoid unnecessary swapping of pages across main memory and storage systems. Furthermore, as I/O operations can be time-consuming in computer systems, iSwap can improve overall system performance. Meanwhile, iSwap handles pages from diverse applications by multi-policy. Thus, in cloud environments, iSwap can effectively reduce swap operations for LC applications and thus improve QoS. The implementation details are in the following.

Algorithm 1: iSwap's Working Procedure

```
   Input: Page Tables
1  do in parallel
2      Procedure PPM //Locate memory regions suffering swap
           Input: Page Tables
3          for each page fault do
4              Call do_swap_page() to swap in required pages;
5              Adds 1 to reserved bits in PTE to locate the swapped memory regions;
6      Procedure PRS //Learn page-level reuse pattern in memory regions produced by PPM
           Input: Memory region produced by PPM
           Output: Page hash table contains page-level reuse pattern
7          for every 3 seconds do
8              Add pages suffering swaps to page hash table;
9              for $i \leftarrow 0$ to 16 do
10                 for page in page hash table do
11                     if access_bits = 1 then
12                         Reuse time adds 1;
13                     Clear access_bits;
14         Return page list;
15     Procedure SWAP //Swap pages according to reuse pattern
           Input: Swapped page list
16         Sort pages in active/inactive list by their reuse times;
17         for each reclamation do
18             Migrate nr pages from the tail of the inactive list to swap cache;
19             Demote n pages from active list to inactive list;
20             if page in swap cache belongs to LC apps then
21                 if compressing cache is full then
22                     Decompress and evict the oldest page in compressing cache to disk;
23                 Compress page to compressing cache;
24             else
25                 Evict page to disk;
```

Table 1. LC and BE workloads used in evaluations

Latency-Critical (LC) Workloads	
Redis (v3.0.6)	In-memory key-value database [19]
MySQL (v1.4.25)	SQL database engine [39]
Memcached (v5.7.33)	Memory object caching system [17]
Best-Effort (BE) Workloads	
Fluidanimate (FA)	Fluid dynamics for animation with Smoothed [6,28]
Streamcluster (SC)	Online clustering of an input stream [6,30]
Canneal (CN)	Simulated cache-aware annealing [6,12]
Freqmine (FM)	Data Mining [6,18]

Table 2. iSwap evaluation with hybrid workloads

Workloads	LC Application	BE Applications
Hybrid workload 1	Redis	FA, SC
Hybrid workload 2	MySQL	FA, SC
Hybrid workload 3	Memcached	FA, SC
Hybrid workload 4	Redis	FA, SC, CN, FM
Hybrid workload 5	MySQL	FA, SC, CN, FM
Hybrid workload 6	Memcached	FA, SC, CN, FM

5 ISWAP IMPLEMENTATION

We implement iSwap in Linux kernel with version 5.10 with 895 LOC. PPM (Sec. 4.2) is in the kernel functions do_swap_page (). When the page fault handler calls do_swap_page () to move data pages into the main memory, PPM has a new function do_add_swap () to update the swap information in the reserved bits in the page's PTE. PRS (Sec. 4.3) works cooperatively with PPM in a pipe-lined way. PRS returns swapped_ht, which is a hash table, to store the reuse patterns for pages. We use separate chaining to resolve hash collision. The shrink_iswap_active_list () function is used to demote pages from the active list to the inactive list. The shrink_iswap_inactive_list () function is used to swap *nr* pages in the inactive list. Both shrink_iswap_active_list () and shrink_iswap_list () are based on the reuse pattern provided by swapped_ht. For the new swap cache mechanism, iSwap uses *frontswap* module [1], which is an interface for storing swap pages in swap devices. iSwap registers two reclaim policies on its back end (Sec. 4.5, policies for pages belonging to LC and BE applications, respectively). After the swap thread migrates pages to the swap cache, iSwap uses reverse mapping similar to try_to_unmap () to get the priority (i.e., $static\_prio \in [100, 139)$) of applications to which the swap-out pages belong, in the function __frontswap_store (). iSwap uses different policies to reclaim pages based on the $static\_prio$ obtained from reverse mapping. In this design, applications with $static\_prio \in [100, 120)$ are LC/high-priority applications. And applications with $static\_prio \in [120, 139)$ are BE/low-priority applications. iSwap uses zswap to manage compressed pages in the main memory and uses the *crypto comp* compress/decompress interface [2] to interact with compressing cache.

6 EVALUATIONS

6.1 Methodology

We evaluate iSwap on a typical cloud server with Intel XEON-6330 2.0GHZ CPUs, 128 GB DDR4 3200MHZ main memory, and an 8TB hard disk. The OS is 64-bit Ubuntu 18.04 with the kernel version 5.10. To better understand how iSwap performs under different memory pressures, we have three memory watermarks (60 GB, 80 GB, and 100 GB) in our experiments. Watermark is the memory threshold that OS starts to swap out inactive pages. Adjusting workload size or memory watermark that leads to different memory pressures has the same effects in the evaluations. Table 1 has the LC and BE applications used for evaluations. We use typical LC cloud services – Redis, MySQL, and Memcached –

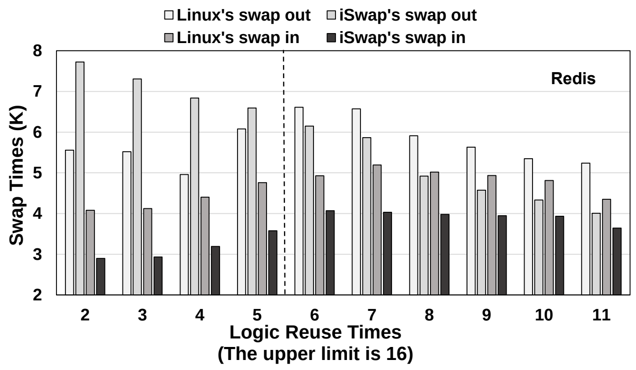


Fig. 7. Swap operations in diverse reuse cases. The x-axis shows the cases of logic reuse time. It is from 2, indicating two reuse times. The y-axis shows the times of swap operations in different cases of reuse time. The larger the value of the x-axis, the higher the reuse rate of the page.

in YCSB benchmark suite [15,16] mainly with workload-A. We run YCSB mainly with heavy-tailed Zipf traffic pattern [16,22] of read and write operations (i.e., 20% of records will be popular while 80% records will be unpopular, which is typical in cloud server data access). In addition, we also test other traffic patterns of YCSB in Sec. 6.4 (e.g., latest, uniform and hotspot). For the QoS, 99th percentile response latency [10,15,29] of LC applications is used as a metric.

Notably, in our evaluations, the memory footprint of LC and BE applications refers to the size of the workloads within these applications. Therefore, at run time, the actual memory footprint of the applications is larger than the memory footprint mentioned in this paper. During our tests, the actual memory footprint exceeds the memory watermark, leading to swap operations. In our experiments, we evaluate iSwap in cases where several applications run together and test iSwap for one specific application that runs in isolation.

Baseline. The swap mechanism in the Linux kernel with version 5.10 is used as the baseline. It uses the widely deployed second chance algorithm discussed in Sec. 2 [4,10,36] and swaps pages directly to the disk.

6.2 Reducing OS swap operations

Page swaps are reduced using iSwap. Compared with the baseline, iSwap reduces the ineffective I/O swaps for Redis, Memcached, and MySQL by 11.4%, 10.3%, and 10.7%, respectively. For swap-in, iSwap reduces 13.6%, 11.2%, and 10.6% for these applications, respectively. These phenomena show that a large number of ineffective swaps are reduced. More details are illustrated in Figure 8. During a specific period in Figure 8, for Redis with a 75 GB memory footprint, 4.53GB of data are moved into main memory by iSwap in total, and 5.68GB of data are swapped out in total. By contrast, without iSwap, 5.24 GB of data are moved into the main memory, and 6.41 GB of data are swapped out. We can see that iSwap swaps fewer data by 13.6% for swap-in and 11.4% for swap-out, as it does not wrongly swap out the to-be-used hot pages.

iSwap can handle diverse memory patterns. We show iSwap's performance for memory pages with different reuse patterns. After running the workload for 30 minutes, according to the value of page reuse pattern learned by PRS, pages are classified into ten categories. As illustrated in Figure 7, when the logic reuse pattern is 6 or more, iSwap reduces the number of pages swapped out by 15.5% and reduces the number of pages swapped in by 19.2%, on average. When the memory system has a high pressure (will run out) and the OS has to swap out some pages with high reuse values, iSwap can swap out fewer than baseline. In the cases where the reuse pattern is below 6, the number of

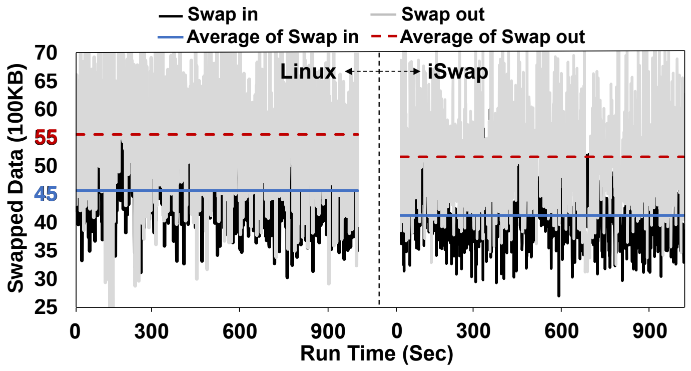


Fig. 8. Linux vs. iSwap. The Application is Redis (75 GB).

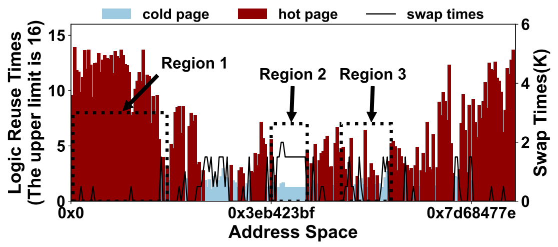


Fig. 9. iSwap's Performance for Redis (35 GB).

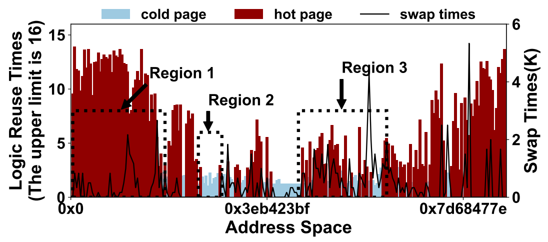
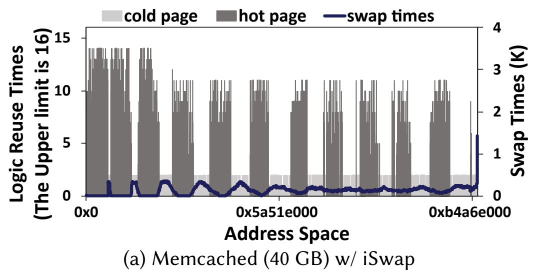


Fig. 10. Linux OS' Swap Mechanism for Redis (35 GB).

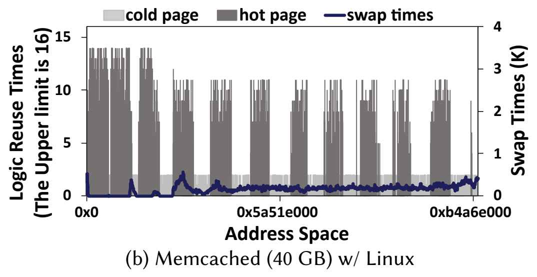
swapped-in pages is also 26.3% lower than the Linux baseline. Notably, Figure 7 shows that in such cases, iSwap swaps more pages out than the baseline by 21.9%, on average. Because iSwap can accurately locate the memory regions with will-not-be-used pages with lower reuse values and swap them out. However, the existing swap mechanism cannot effectively identify the memory regions with lots of cold and hot pages interleaved and thus cannot swap out these cold pages with lower reuse values, leading to a lower number of swapped-out pages.

iSwap conducts effective swaps. We run the cloud workloads for 30 minutes. iSwap works 500 times during this period. We have the average reuse values and the total number of swap operations during the 30 minutes. Figures 9 and 10 show the details for Redis. As illustrated in Figure 9, iSwap can accurately find and swap out the cold pages that will not be reused pretty soon in the applications' address space (e.g., Region 2); and the will-be-used hot pages are kept in main memory (e.g., Region 1). Even for the memory regions in which many cold and hot pages are interleaved (e.g., Region 3), iSwap can work well. By contrast, the baseline mechanism does not work well. In Figure 10, it fails to find the memory pages (regions) that should be swapped (e.g., Region 1 and 2 in Figure 10, hot pages are swapped many times but cold pages are kept), especially in the memory regions many cold and hot pages are interleaved (e.g., Region 3). In addition, the baseline approach completely ignores Region 2 (i.e., the region that mainly contains cold pages and should be swapped out) but swaps to-be-used pages as a result.

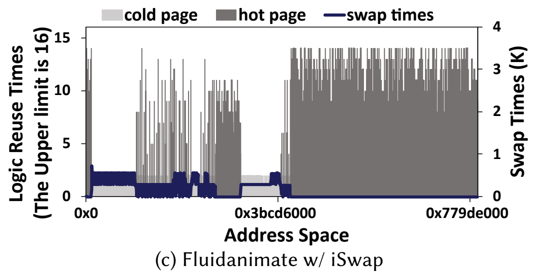
We further evaluate iSwap's performance using Memcached and MySQL with hybrid workloads (i.e., hybrid workloads 2 and 3 in Table 2). Workload 2 has Memcached with a 40 GB memory footprint and BE applications, including Fluidanimate (5 threads, 20 GB) and Streamcluster (10 threads, 10 GB). Workload 3 includes MySQL with a 40GB memory footprint, and BE applications are the same as workload 2 mentioned above. We run these two workloads for 800 seconds with a memory watermark of 60 GB using iSwap and Linux. We record the average reuse values for 800



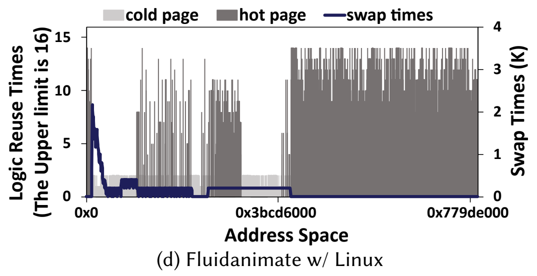
(a) Memcached (40 GB) w/ iSwap



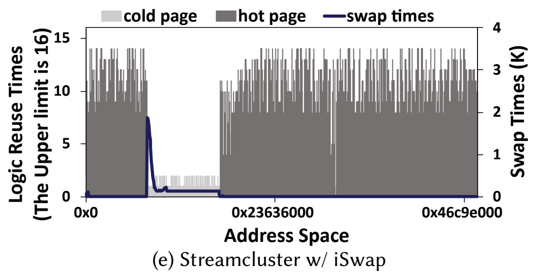
(b) Memcached (40 GB) w/ Linux



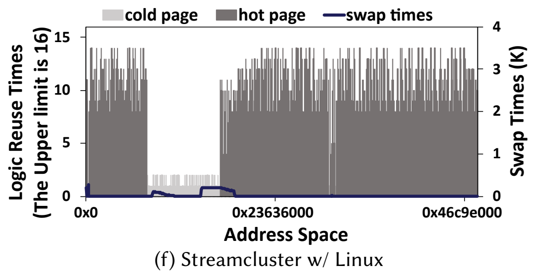
(c) Fluidanimate w/ iSwap



(d) Fluidanimate w/ Linux



(e) Streamcluster w/ iSwap



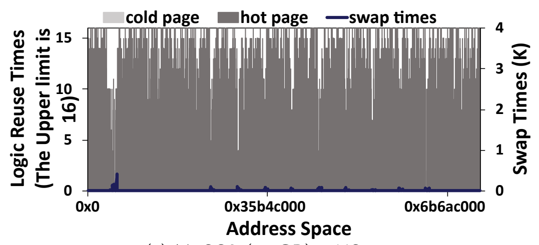
(f) Streamcluster w/ Linux

Fig. 11. Memcached + BE. LC and BE apps' swapping operations. iSwap vs Linux.

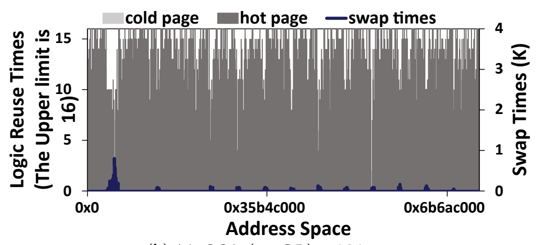
seconds and the total number of swap operations for each application on its address space. Running 800 seconds is sufficient to show iSwap performance. The experimental results are shown in Figure 11 and Figure 12. For both LC and BE applications, iSwap can accurately find and swap out the actual cold pages, whose reuse patterns show that they will not be reused soon in the applications' address space. As illustrated in Figure 11-a,c and e, the swap operations mainly happen in the address regions where cold pages are located. Based on reuse patterns, iSwap can accurately evict cold pages for both LC and BE applications. Moreover, due to the applications' features, iSwap swaps out more pages belonging to BE applications. But these swapped pages are mainly in the cold memory regions (e.g., Fluidanimate in Figure 11 and Figure 12), which has negligible impacts on the overall system performance. By contrast, the baseline approach blindly swaps many hot pages, as discussed before. Details are illustrated in Figure 11-b,d, and f. In general, iSwap outperforms the baseline swap mechanism used in Linux.

6.3 Run-time Details on Swap in vs. Swap out

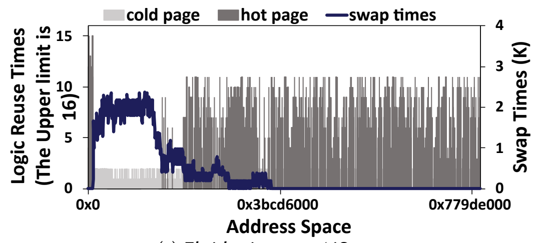
We show how iSwap performs during the run time. We run hybrid workloads 1, 2 and 3 in Table 2. Each workload has an LC application (i.e., Redis, MySQL, or Memcached) with a 40 GB memory footprint and BE applications, i.e., Fluidanimate (5 threads, 20 GB) and Streamcluster (10 threads, 10 GB). we monitor *VmSwap* in *proc* filesystem [7] to obtain the swapped data per second (MB/s) for each application. We show run-time swapping in and out in Figure



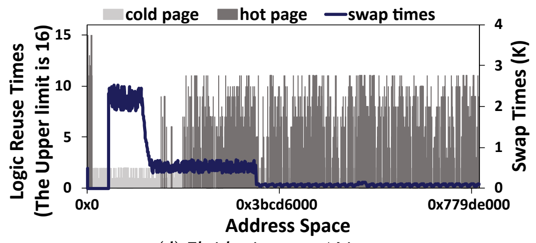
(a) MySQL (40 GB) w/ iSwap



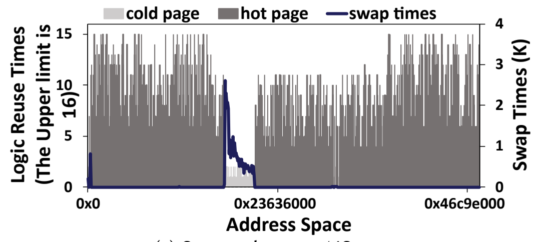
(b) MySQL (40 GB) w/ Linux



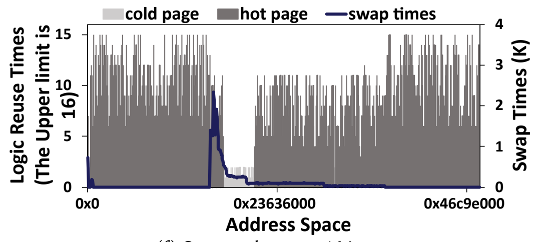
(c) Fluidanimate w/ iSwap



(d) Fluidanimate w/ Linux



(e) Streamcluster w/ iSwap



(f) Streamcluster w/ Linux

Fig. 12. MySQL + BE. LC and BE apps' swapping operations. iSwap vs. Linux.

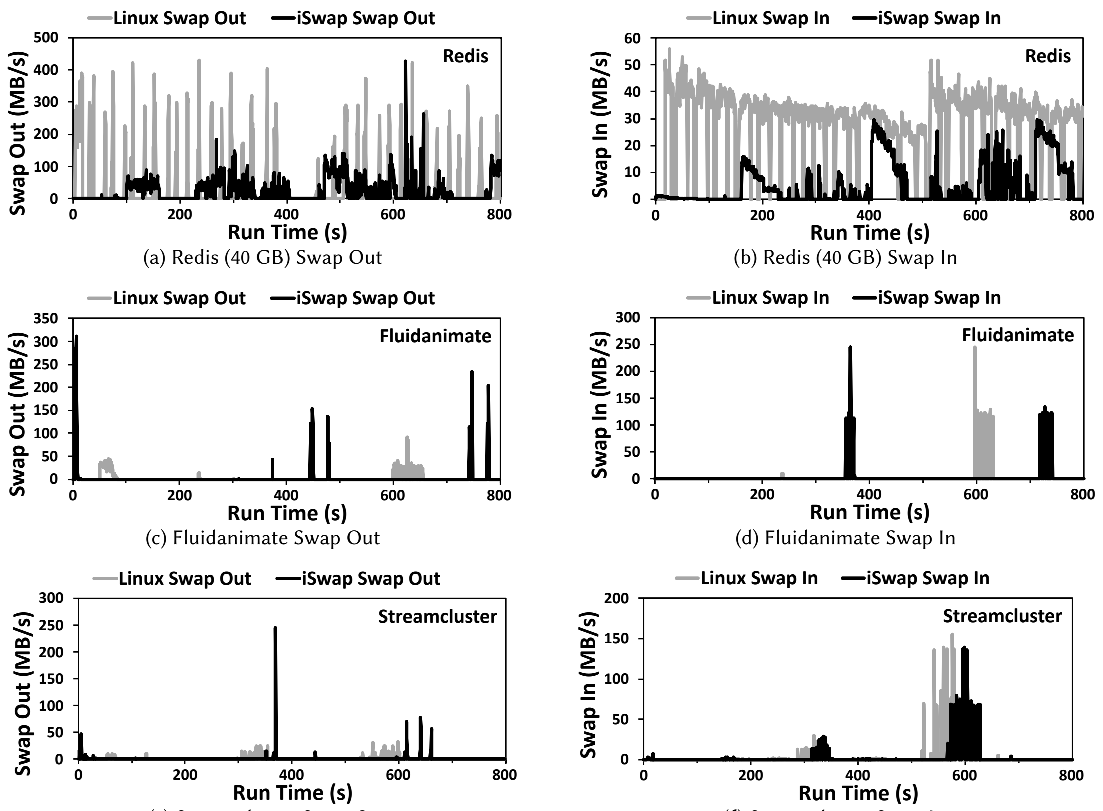
Table 3. Linux: LC and BE apps' proportion of pages swapped out.

Linux	LC proportion	BE proportion
Hybrid workload 1 (Redis + BE)	67.1%	32.9%
Hybrid workload 2 (MySQL + BE)	54.3%	45.7%
Hybrid workload 3 (Memcached + BE)	64.1%	35.9%

Table 4. iSwap: LC and BE apps' proportion of pages swapped out.

iSwap	LC proportion	BE proportion
Hybrid workload 1 (Redis + BE)	42.8%	57.2%
Hybrid workload 2 (MySQL + BE)	32.9%	67.1%
Hybrid workload 3 (Memcached + BE)	36.6%	63.4%

13, 14, 15 for workload 1, 2, and 3 respectively. Each figure compares the swap in/out for iSwap and the baseline. For instance, as illustrated in (Figure 13-a/b) for LC application Redis, iSwap performs better than baseline. It reduces the ineffective swap operations, and thus the amount of data swapped out/in is lower. During the 800 seconds, the data swapped out are reduced by 30.7%, and the data swapped in are reduced by 82.1%, on average. As the ineffective swap-out operations are reduced, the swap-in operations for moving those to-be-used/hot pages into the main memory again are also significantly reduced. The same phenomena can also be observed for the BE applications in Figure 13. In Figure 14, we also find that the data swapped out are reduced by 15.3%, and the data swapped in are reduced by



(a) Redis (40 GB) Swap Out (b) Redis (40 GB) Swap In

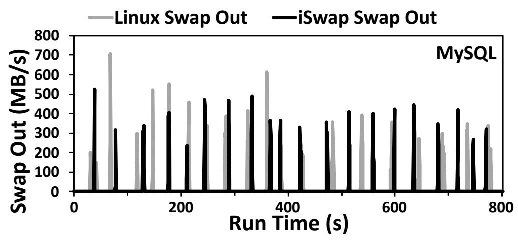
(c) Fluidanimate Swap Out (d) Fluidanimate Swap In

(e) Streamcluster Swap Out (f) Streamcluster Swap In

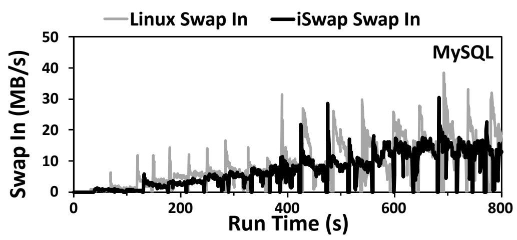
Fig. 13. Redis + BE. LC and BE apps' swapping at run time. iSwap vs Linux.

21.0% for LC application MySQL. In Figure 15, for Memcached, iSwap also reduces data swapped out by 73.3% and data swapped in by 58.8%.

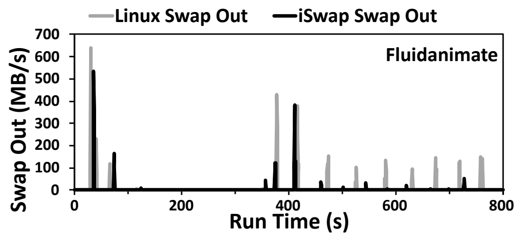
To further show how iSwap performs, we record the number of pages belonging to LC applications (*lc_num*) and the number of pages belonging to BE applications (*be_num*) that are actually swapped out to the hard disk for iSwap and baseline approach, respectively. For iSwap, the swapped pages include BE applications' pages and LC applications' pages that are decompressed and evicted from compressing cache (Sec. 4.5). Then, we calculate *LC proportion* $lc\_num/(lc\_num + be\_num)$, and *BE proportion* is calculated as $be\_num/(lc\_num + be\_num)$. The two metrics reflect iSwap's effectiveness in keeping LC applications' pages in memory and evicting pages belonging to BE applications. Table 3 and 4 have the results. For LC applications, using Linux's original swap approach, the LC proportion for the three workloads is 67.1%, 54.3%, and 64.1%, respectively, indicating that the swapped pages belonging to LC applications accounted for more than half of the total number of swapped pages. This is because Linux's swap mechanism does not distinguish pages for applications and conducts swap operations blindly. As LC applications have a larger memory footprint, more pages belonging to LC applications are swapped as a result. By contrast, using iSwap, the LC proportion during swapping is reduced by 24.3%, 21.4%, and 27.5% for these workloads, respectively. This is because iSwap conducts intelligent swaps, which avoid ineffective swapping pages belonging to higher-priority LC applications and evicting more pages from BE applications. Therefore, we can see BE proportion is increased in Table 4.



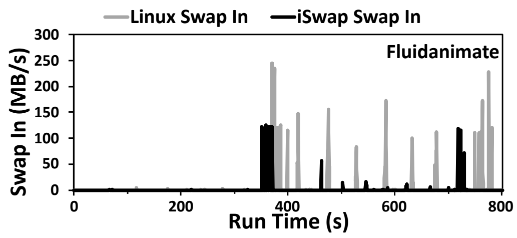
(a) MySQL (40 GB) Swap Out



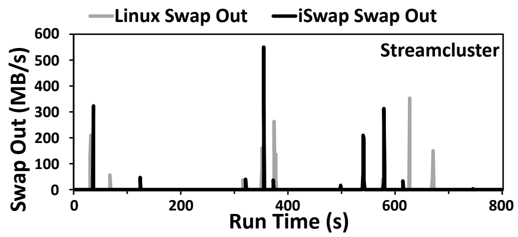
(b) MySQL (40 GB) Swap In



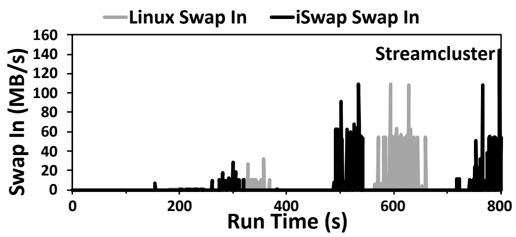
(c) Fluidanimate Swap Out



(d) Fluidanimate Swap In



(e) Streamcluster Swap Out



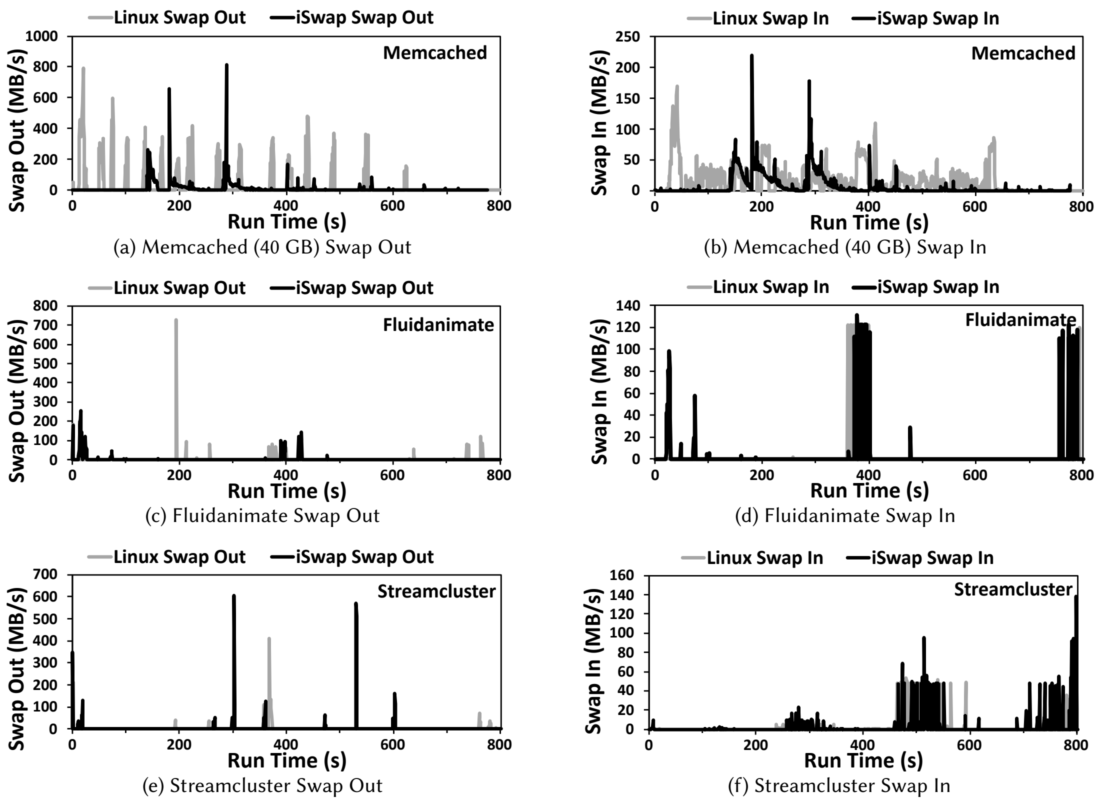
(f) Streamcluster Swap In

Fig. 14. MySQL + BE. LC and BE apps' swapping at run time. iSwap vs Linux.

Memory compression. In the above experiments, for Redis in Figure 13-a, we find that iSwap starts to swap at time point 79 and the Linux swap mechanism starts to work from time point 6. This is because iSwap compresses the pages belonging to LC applications in its compressing cache at the beginning. When the compressing cache is full, iSwap begins to decompress the oldest pages in compressing cache and evicts them, as discussed in Sec. 4.5. The same phenomena also happen for MySQL and Memcached in our experiments. For MySQL in Figure 14-a, iSwap starts to swap at time point 56, and Linux starts to swap at time point 45. In Figure 15-a, iSwap starts swapping at time point 147, while Linux swaps at time point 14. During the 800 seconds in Figure 13, we also record how much data is compressed. For the LC application Redis in workload 1, 25.8 GB of data is compressed, and 16.6 GB of data is decompressed. This indicates that the compressing cache in iSwap avoids I/Os for 9.2 GB of data. For MySQL in workload 2, iSwap compresses 14.9 GB of data and decompresses 5.7 GB of data. In workload 3, 20.5 GB of data is compressed, and 13.2 GB is decompressed.

6.4 Overall QoS Benefits for Cloud Applications

To show the overall performance benefits brought by iSwap, we evaluate the QoS (the 99th percentile response latency) for co-located LC applications. Figure 16 shows the LC applications' QoS when using iSwap and baseline. In general, we observe that iSwap can bring lower response latency for LC services, especially when memory pressure is high. For Redis in Figure 16-a, at the beginning, since the memory pressure is low, the swap operations rarely happen, and



(a) Memcached (40 GB) Swap Out
(b) Memcached (40 GB) Swap In
(c) Fluidanimate Swap Out
(d) Fluidanimate Swap In
(e) Streamcluster Swap Out
(f) Streamcluster Swap In

Fig. 15. Memcached + BE. LC and BE apps' swapping at run time. iSwap vs Linux.

thus iSwap and the baseline perform similarly. As the system continues running, when the memory pressure increases, iSwap outperforms the baseline. This is because when the swap mechanism starts to work, and as discussed before, iSwap swaps out more cold pages and keeps pages belonging to LC applications in memory, avoiding the involvement of low-speed I/O operations. As a result, Redis has a lower response latency. iSwap reduces the response latency of LC application Redis by 73.6% on average, significantly improving the QoS for LC/high-priority application in cloud environments. We find iSwap also reduces the response latency for other LC applications in Figure 16. For MySQL in Figure 16-b, though the latency trend is different than Redis, iSwap reduces the response latency by 37.7%. And in Figure 16-c, iSwap reduces the response latency by 83.4% for the LC application Memcached. Moreover, in Figure 16, the response latency curve for the iSwap curve fluctuates less than the baseline, indicating a more stable QoS.

We further evaluate iSwap using more applications from Table 2 with diverse system configurations (OS memory watermark with 80 GB and 100 GB). Each workload has an LC application (i.e., Redis, MySQL, and Memcached) with a 60 GB memory footprint. Table 2 shows that Workloads 1-3 have two different BE applications, Fluidanimate (5 threads, 20 GB) and Streamcluster (10 threads, 10 GB). For workloads 4-6, we add another two BE applications, i.e., Canneal (5 threads, 20 GB) and Freqmine (5 threads, 5 GB). We run these workloads for 800 seconds and record the average 99th response latency. We summarize the results in Figure 17. Generally, iSwap still works well. In the cases where the watermark is 80 GB, for workloads 1-3, iSwap reduces response latency by 69.2%, 51.1%, and 91.3% for Redis, MySQL, and Memcached on average, respectively. For workload 4-6, as these workloads have a larger memory footprint,

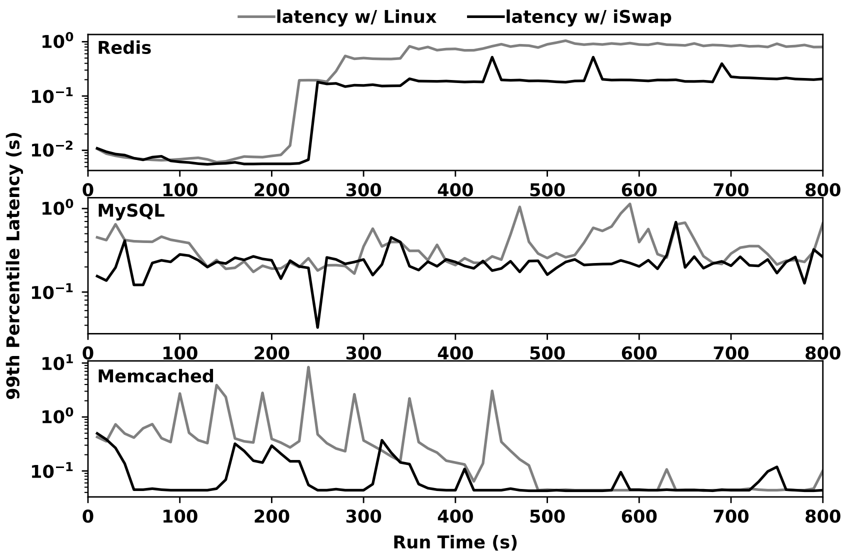


Fig. 16. 99th percentile response latency of LC applications. iSwap vs Linux.

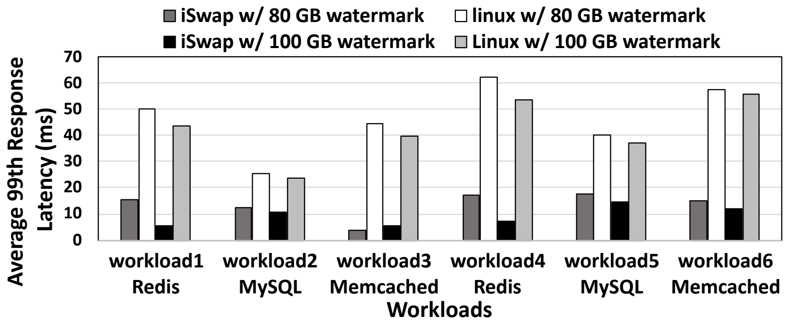


Fig. 17. Overall Performance for Cloud Applications.

response latency increase for all workloads. But iSwap still reduces the response latency of LC applications Redis, MySQL, and Memcached by 72.5%, 55.4%, and 73.8%, respectively. For the cases where the watermark is 100 GB, we observe the iSwap performs better. In workload 1-3, response latency is reduced by 87.7%, 53.8%, and 85.6% for Redis, MySQL, and Memcached, respectively. As for workload 4-6, iSwap reduces response latency by 86.2%, 60.9%, and 78.7%, respectively. We also run Redis, MySQL, and Memcached with latest, uniform, and hotspot traffic pattern respectively. We observe similar experimental results in the experiment as mentioned above. Therefore, we conclude that iSwap works well in diverse cases in cloud environments and performs better in cases where the memory footprint is large.

Moreover, iSwap brings around 1.8% overheads to the OS kernel. The overheads originate from OS kernel operations. For instance, we observe from vmstat that iSwap brings additional 5.3 seconds for running kernel code when Redis with workload-A (35 GB) runs 298.5 seconds. This extra overhead is mainly caused by PPM and PRS scanning pages. However, the overhead is not significant, as modern servers can have more cores for OS threads, and the performance improvements brought by iSwap overweight its kernel overheads.

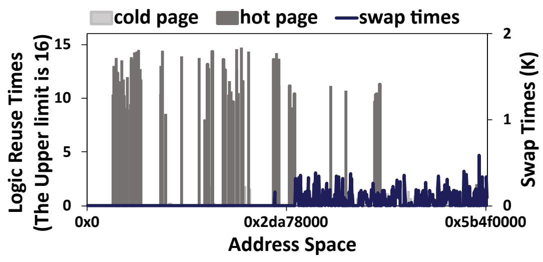


Fig. 18. iSwap's performance for Redis on ARM (60 GB).

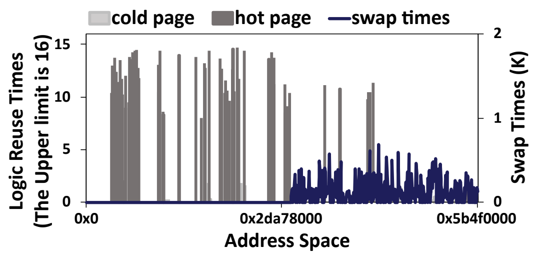


Fig. 19. Linux' paging operations for Redis on ARM (60 GB).

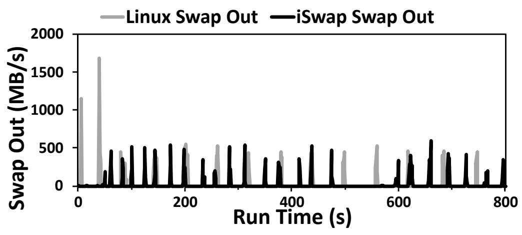


Fig. 20. Swap out operations on ARM. iSwap vs Linux.

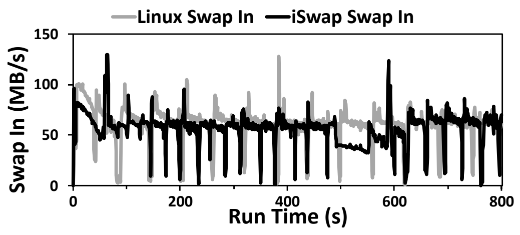


Fig. 21. Swap in operations on ARM. iSwap vs Linux.

6.5 iSwap on ARM Platform

Besides the x86 platform, we also evaluate iSwap on the ARM platform. We use a typical ARM cloud server with Phytium S2500 2.1GHZ CPUs, 128 GB DDR4 3200MHZ main memory, and a 1TB hard disk. The OS is 64-bit CentOS 8.5 with the kernel version 5.10, which uses a 3-stage page table, and the page grain is 64KB. By contrast, the system used in our experiments above is x86 with a 5-stage page table and 4KB page. On the x86 platform, iSwap records swapped pages using reserved bits 52-55 in its PTE. However, the PTE format on the ARM platform is different. iSwap uses reserved bits 12-15 in PTE to record swapped pages. We evaluate iSwap using workload 1 in Table 2. The memory watermark in OS is 80 GB. The LC application is Redis with a 60GB memory footprint. The BE applications are Fluidanimate (5 threads, 20 GB) and Streamcluster (10 threads, 10 GB). We use the same metrics as before.

We run the workloads for 800 seconds using iSwap and Linux baseline, respectively. The experimental results for the distributions of hot/cold pages and the swap operations are illustrated in Figure 18 and 19. On the ARM platform, as shown in Figure 18, iSwap can accurately find and swap out the cold pages that will not be reused soon in the Redis' address space. In Figure 19, since Linux uses the same hot/cold page classification algorithm as on the x86 platform, Linux fails to find the memory pages that should be swapped but swapped to-be-used pages. So, we can see that iSwap has fewer swap operations. We further show how iSwap performs during the 800-second run time. We capture the swapped data per second (MB/s) for the overall system. We show run-time swapping in and out in Figure 20 and 21. Figure 20 compares the swap in, and Figure 21 compares the swap out for iSwap and the baseline. Figure 20 and 21 illustrate that iSwap performs better than the baseline. iSwap can reduce ineffective swap operations. Therefore, the amount of data swapped out/in is lower. During the 800 seconds, the data swapped in are reduced by 8.0%, and data swapped out are reduced by 18.2%, on average, respectively.

We further evaluate QoS (the 99th percentile response latency) for Redis. Figure 22 shows the experimental results. We observe that iSwap can bring lower response latency for LC services on the ARM platform, especially when memory pressure is high. iSwap swaps out more cold pages and keeps pages belonging to LC applications in memory, avoiding

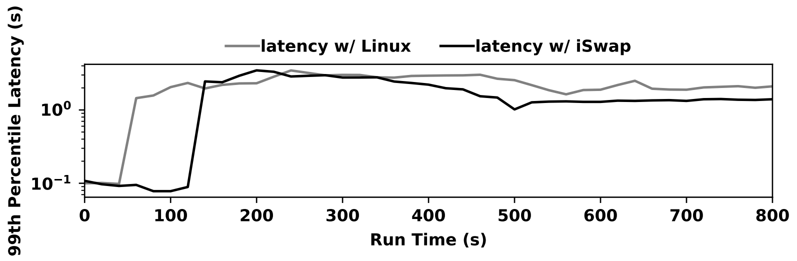


Fig. 22. 99th percentile response latency of Redis on a ARM platform. iSwap vs Linux.

the low-speed I/O operations. As a result, Redis has a lower response latency. iSwap reduces the 99th response latency for Redis by 36.8%, on average. Moreover, we notice that compared with the benefits brought by iSwap on the x86 platform, the benefits on the ARM platform are not that significant. The underlying reason is the OS on the ARM platform used in our experiments has a coarser page granularity (i.e., 64KB). Using a coarser page granularity decreases the number of allocated memory pages, and a specific page can cover more memory patterns.

6.6 iSwap on NVMe SSD

In addition to the above experiments on HDD, we conduct experiments using a 1.5TB NVMe SSD equipped on the previously used Intel platform in sections 6.1 - 6.4. Compared with HDD, NVMe SSD has a faster access speed. We run Workload 1 in Table 2. The workload has Redis with a 40GB memory footprint and BE applications - Fluidanimate (5 threads, 20GB) and Streamcluster (10 threads, 10 GB). The memory watermark in the OS is 60 GB. We run the workload for 800 seconds. In general, iSwap reduces data swapped in by 12.1% and data swapped out by 13.3%, exhibiting a similar trend as on HDD. For the QoS improvement of LC application, iSwap reduces the 99th percentile latency by 47.8%, on average. The QoS improvement on SSD is less significant than the 69.2% improvement on HDD. The reason is that the NVMe SSD has a faster access speed than the HDD, so the overheads brought by ineffective swap operations are mitigated. However, from another angle, the efforts and money paid on improving the storage performance are wasted by these ineffective swap operations, which iSwap can tackle.

7 RELATED WORK

Page classification. Many existing studies conduct memory page classification. The work in [47] monitors the access bit to track the page miss ratio curve. The study in [43] uses a region-based sampling to trace cold pages and an adaptive region adjustment to limit the overhead. G-swap [44] additionally tracks pages' age histograms and sets the cold age threshold using an ML approach. Then, it reclaims pages according to the threshold. The work in [46] introduces an additional page flag to track cold pages. The study in [50] uses a user-space process that reads a bitmap stored in sysfs to track cold pages. The studies in [25,26,51] also monitor the access bit to have the temperature of memory pages. TMO [45] uses a new metric to measure the memory pressure and determines whether to swap out anonymous pages or file pages. In this work, iSwap monitors the access bit and learns the page-level logic reuse patterns.

Page compression. The page compression mechanisms in the Linux kernel include zswap [9], zram [48] and zcache [49]. zswap and zram compress pages in the main memory. Zswap can evict pages out to the storage when compressing cache is full while zram compresses all the pages in the main memory. zcache only compresses file pages in the main memory. The work in [44] swaps cold pages into a compressed in-memory pool and uses machine learning to enforce a stable page swapping rate. The main difference between our work and [9,44,48,49] is that iSwap compresses

LC/high-priority applications' pages in the main memory and swaps out BE applications' pages to hard disk. Therefore, LC/high-priority applications can avoid inefficient IO swaps, benefiting their QoS. iSwap is more applicable to the cases where LC and BE applications are co-located on a specific server.

8 CONCLUSION

Memory swap mechanism in OSes fundamentally plays an essential role in overall system performance and QoS. The swap mechanism directly affects the throughput and QoS on cloud servers, edge devices, and mobiles. How to design a new swap mechanism in modern OSes is still a hot topic. This paper proposes iSwap, a new memory page swap mechanism that learns the page-level reuse features to reduce ineffective I/O swap operations. Moreover, iSwap conducts swap operations according to the applications' priority, compressing high-priority applications' memory pages and keeping them in the swap cache, evicting pages belonging to low-priority applications. We show that iSwap performs well in cloud environments. It improves the overall system performance and QoS for LC services, especially for the cloud platform running applications with large memory footprints and low latency requirements. Compared with one-size-fits-all approaches, iSwap has a regression-based learning ability and intelligently enables a suitable memory swap policy. Regarding future OS designs, in a world where hardware and applications become increasingly complicated, making OSes intelligent can be a promising way to handle the complexity and diversity.

9 ACKNOWLEDGMENTS

We thank the reviewers, AE and EIC for their invaluable comments. We also thank the previous student members who paid attention to this project. This work is supported by the Key-Area R&D Program of Guangdong (grant no. 2021B0101310002), NSFC (grant no. 62072432). Z. Wang is a student member in Sys-Inventor Lab led by L. Liu.

REFERENCES

[1] "Cleancache and frontswap," https://lwn.net/Articles/386090/.
[2] "The crypto compression api," https://docs.kernel.org/crypto.
[3] "The FreeBSD project," https://www.freebsd.org.
[4] "The linux kernel archives," https://www.kernel.org/.
[5] "Page table management," https://www.kernel.org/doc/gorman/html/understand/understand006.html.
[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008.
[7] "The /proc filesystem," https://docs.kernel.org/filesystem/proc.html.
[8] "Tunable watermark," https://lwn.net/Articles/422291/.
[9] "The zswap compressed swap cache," https://lwn.net/Articles/537422/.
[10] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *NSDI*, 2018.
[11] S. Bai, H. Wan, Y. Huang, X. Sun, F. Wu, C. Xie, H.-C. Hsieh, T.-W. Kuo, and C. J. Xue, "Pipette: Efficient fine-grained reads for SSDs," in *DAC*, 2022.
[12] P. Banerjee, *Parallel algorithms for VLSI computer-aided design.* Prentice-Hall, Inc., 1994.
[13] S. Bergman, N. Cassel, M. Bjorling, and M. Silberstein, "ZNSwap: un-Block your swap," in *USENIX ATC*, 2022.
[14] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *ASPLOS*, 2019.
[15] B. Cooper, "YCSB: Yahoo! cloud serving benchmark."
[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[17] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, 2004.
[18] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets." in *FIMI*, 2003.
[19] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *PERCOM*, 2011.
[20] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *USENIX ATC*, 2005.

[21] S. Kim and J.-S. Yang, "Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system," in *DAC*, 2018.

[22] C. Kurumada, S. C. Meylan, and M. C. Frank, "Zipfian frequency distributions facilitate word segmentation in context," *Cognition*, 2013.

[23] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *USENIX ATC*, 2020.

[24] L. Liu, et al, "Intelligent resource scheduling for co-located latency-critical services: A multi-model collaborative learning approach," in *USENIX FAST*, 2023.

[25] L. Liu, et al, "Rethinking memory management in modern operating system: Horizontal, vertical or random?" in *IEEE TC*, 2016.

[26] L. Liu, et al, "Hierarchical hybrid memory management in OS for tiered memory systems," in *IEEE TPDS*, 2019.

[27] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami, "Multi-clock: Dynamic tiering for hybrid memory systems," in *HPCA*, 2022.

[28] M. Müller, D. Charypar, and M. H. Gross, "Particle-based fluid simulation for interactive applications." in *SCA*, 2003.

[29] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads," in *NSDI*, 2019.

[30] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani, "High-performance clustering of streams and large data sets," in *ICDE*, 2002.

[31] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: making page walks single-access cache hits," in *ASPLOS*, 2022.

[32] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu, "Reducing solid-state drive read latency by optimizing read-retry," in *ASPLOS*, 2021.

[33] B. K. Tanaka, "Monitoring virtual memory with vmstat," in *Linux Journal*, 2005.

[34] A. S. Tenenbaum, *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[35] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," in *ASPLOS*, 2013.

[36] J. Yang, Y. Wang, and Z. Wang, "Efficient modeling of random sampling-based LRU," in *ICPP*, 2021.

[37] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *EuroSys*, 2009.

[38] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *HPCA*, 2020.

[39] "MySQL Database," https://www.mysql.com.

[40] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2011.

[41] T. Anderson and M. Dahlin, *Operating Systems: Principles and Practice.* Recursive books, 2014.

[42] J. H. Saltzer and M. F. Kaashoek, *Principles of computer system design: an introduction.* Morgan Kaufmann, 2009.

[43] Park, SeongJae, Yunjae Lee, and Heon Y. Yeom. "Profiling dynamic data access patterns with controlled overhead and quality," in Proceedings of the 20th International Middleware Conference Industrial Track, 2019.

[44] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *ASPLOS*, 2019.

[45] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang *et al.*, "TMO: transparent memory offloading in datacenters," in *ASPLOS*, 2022.

[46] "Idle page tracking/working set estimation.," https://lwn.net/Articles/460762/.

[47] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGPLAN Notices*, 2004.

[48] "zram: Compressed RAM based block devices.," https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[49] "zcache: a compressed file page cache.," https://lwn.net/Articles/562254/.

[50] "Idle and stale page tracking.," https://lwn.net/Articles/461461/.

[51] L. Liu, C. Wu, and X. Feng. "Memory resource optimization method and apparatus," US Patent No. 9,857,980, 2018.