APP: Enabling Soft Real-Time Execution on Densely-Populated Hybrid Memory System

Zheng-Wei Wu*, Yun-Chih Chen*, Yuan-Hao Chang†, Tei-Wei Kuo*‡§
*Department of Computer Science and Information Engineering, National Taiwan University
†Institute of Information Science, Academia Sinica, Taiwan
‡Mohamed bin Zayed University of Artificial Intelligence, Department of Computer Science,
§NTU High Performance and Scientific Computing Center, National Taiwan University
E-mail: {r09922003, f07922039}@ntu.edu.tw, johnson@iis.sinica.edu.tw, ktw@csie.ntu.edu.tw

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247672

***Abstract*—Memory swapping was considered slow and evil, but swapping to Ultra Low-Latency storage like Optane has become a promising solution to save power and cost, helping densely-populated edge server to overcome its DRAM capacity bottleneck. However, the lack of integration between CPU scheduling and memory paging causes soft real-time tasks running on edge servers to miss deadlines under heavy memory multiplexing. We propose APP (Adaptive Page Pinning), lightweight protection of working set memory to ensure meeting soft real-time task deadlines without starving other non-real-time tasks. Experiments show that APP alleviates thrashing in memory-intensive tasks and upholds soft real-time task deadlines.**

I. Introduction

The demand for larger memory capacity has never ceased since the dawn of computing. Virtual memory is a classic operating system technique to expose huge memory capacity to applications than physically available. When memory runs out, the OS swaps less frequently accessed memory pages to slow HDDs, typically resulting in a severe performance drop. A long-standing suggestion is to entirely disable memory swapping and provision sufficient DRAM for peak memory usage. However, the advent of new non-volatile memory (NVM), such as Intel Optane and Ultra-Low Latency SSDs, has led many to reconsider such a suggestion. The new NVMs are orders of magnitude faster than HDD and larger than DRAM, making them a more feasible swap backend than HDD.

Provisioning memory for peak usage can lead to under-utilization in non-peak periods [1]. A more economical solution might be a hybrid memory system composed of a small DRAM and the new low-latency NVM as a swap backend. Many data center operators recently adopted hybrid memory into multi-tenant servers hosting both latency-sensitive services and background memory-hungry jobs [2], [3]. For them, as long as a certain degree of Quality-of-Service (QoS) is guaranteed, cutting hardware and energy cost are preferable to maximum performance. Memory swapping is no longer considered evil but a viable path to higher memory utilization.

In edge computing, edge servers are often DRAM-constrained due to a strict power budget. As edge servers turn increasingly multi-tenant [4], overcoming the DRAM capacity bottleneck becomes an urgent issue. The success of hybrid memory in data centers might lead one to ask: "Can an edge server adopt hybrid memory to extend its logical memory capacity?" To answer this question, we must consider one critical property of many edge applications: the real-time requirement. Failing the real-time requirement (i.e., ensuring a task finishes within a specific deadline) leads to varying severity. For hard real-time systems like autonomous vehicles, a missed deadline can result in fatal accidents. For soft real-time systems like virtual reality and video streaming, a missed deadline is non-fatal but degrades service quality.

While hard real-time applications typically run in strict isolation, soft real-time (abbr. soft-RT) applications often co-run with non-real-time (abbr. non-RT) tasks on a multi-tenant edge server. Non-RT tasks like DNN training and batch data analytics consume significantly more memory than soft-RT tasks, but they also run far less frequently. As the edge server experiences peak memory only occasionally, adopting hybrid memory has the potential to improve DRAM utilization in non-peak periods. However, another issue arises: "Can hybrid memory uphold real-time requirements?"

In this paper, we observe that soft-RT tasks will suffer from severe memory thrashing under a hybrid memory system. The memory thrashing stems from the conflicting assumptions of two independently-designed OS components: real-time scheduling and memory management. Based on this observation, we propose a novel co-design of the two components. We demonstrate that through controlled memory overcommit, hybrid memory can effectively extend an edge server's usable logical memory and still sustain real-time requirements. Our contributions are summarized as follows.

1) We propose Adaptive Page Pinning (APP), lightweight protection of soft-RT tasks co-running with non-RT tasks on hybrid memory. APP identifies the key portion of a soft-RT task's memory pages that must be *pinned* in DRAM and protected from swapping to sustain its real-time requirements. Meanwhile, APP reclaims as much free memory as possible to prevent non-RT tasks from memory starvation.
2) We introduce a lightweight data structure called Pinned Page Pool (PPP) to each soft-RT task to track the key portion of their memory pages. PPP can keep up with soft-RT tasks' changing memory access pattern. Furthermore, PPP can be dynamically resized according to the soft-RT's status retrieved from the real-time scheduler.
3) We introduce the concept of *swap budget* to make real-time scheduling paging aware. We observe that the kernel overhead incurred by memory paging steals the CPU budget of a soft-RT task, causing the scheduler to prematurely preempt the task. Thus, the purpose of the swap budget is to compensate for the lost budget. Experiments show that the swap budget effectively reduces missed deadlines on hybrid memory.
4) Comprehensive evaluations show that APP effectively alleviates system-wide memory thrashing under heavy memory overcommitment. Compared to the baselines, APP reduces soft-RT tasks' deadline misses and increases non-RT tasks' throughput.

The rest of the paper is organized as follows. §II reveals why a naïve adoption of hybrid memory will cause memory thrashing. §III presents how APP mitigates such a hazard. §IV shows APP's effectiveness through experiments. §V compares APP with related works. §VI concludes this work.

II. Background & Motivation

A. Hybrid Memory

Memory capacity remains a bottleneck to vertically scaling a computer due to the limited number of DIMMs in a computer and the slowdown in DRAM's capacity scaling. The memory bottleneck poses a great challenge for edge computing. Due to stringent thermal and area constraints, an edge server cannot afford sufficient DRAM to run memory-intensive applications. A classical solution is memory overcommitment, which assigns to a process more virtual memory than physically available. Memory overcommitment works because a process might allocate a large portion of memory pages from the OS, but only a key portion is frequently accessed. These frequently-accessed pages are known as the process's *working set*. When the

memory runs out, space can be freed up by swapping infrequently-accessed pages to disks. Since the long disk latency can significantly degrade system availability and performance, conventional wisdom often suggests disabling disk swapping. Recently, many have suggested replacing the slow disk with emerging Ultra-Low-Latency (ULL) SSDs, such as Intel Optane SSD and Samsung's Z-NAND, as the swap backend [2], [3], [5], [6], [7]. A hybrid memory system with a small DRAM augmented by a large ULL SSD has the potential to sustain the required performance at a lower cost than a pure-DRAM system. Furthermore, memory-intensive applications can be transparently offloaded from a cloud environment to an edge server, adhering to the edge's DRAM constraint without code modification.

B. Linux's Deadline Scheduler

Linux's Deadline Scheduler, `SCHED_DEADLINE`, implements the *Earliest Deadline First* and *Constant Bandwidth Server* principles in real-time theory. `SCHED_DEADLINE` guarantees a soft-RT task, S, a CPU time budget of T_s that it can spend before a relative deadline D_s. S runs periodically at the interval of I_s, and the deadline is typically the same as I_s. To prevent starving other processes, `SCHED_DEADLINE` preempts S if it runs more than T_s. S will have to wait until its next period before it can run again.

In a typically multi-tenant edge server, soft-RT applications (e.g., video streaming) and non-real-time applications (e.g., background data analytics) are often co-located to increase resource utilization. `SCHED_DEADLINE` can effectively guarantee the soft-RT applications' QoS when they co-run with other throughput-oriented processes governed by the default general-purpose scheduler. Compared to Linux's other fixed-priority scheduling algorithms (e.g., `SCHED_RR`), `SCHED_DEADLINE` provides better timing guarantees. While full-fledged real-time OSes (e.g., the `PREEMPT_RT` Linux) are for hard real-time applications, `SCHED_DEADLINE` focuses on soft-RT applications.

C. Hybrid-memory-based Edge Server

Past real-time research focuses primarily on CPU schedulability. Due to the long disk latency, disk swapping is usually dismissed. To handle peak memory usage, DRAM is usually over-provisioned. Though over-provisioning brings predictability, much DRAM is left idle, wasting energy in non-peak periods.

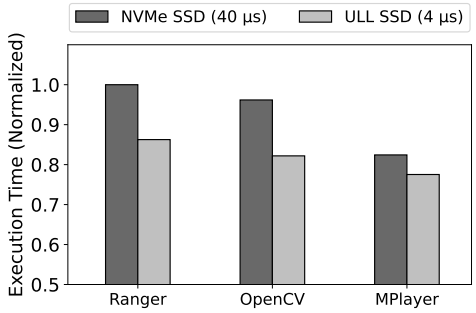
Hybrid memory is a more cost-effective solution that has been adopted in both data centers [2], [3] and mobile devices [5], [6], [7]. However, prior works have yet to study the implication of adopting hybrid memory into an edge server that hosts both real-time and non-real-time applications. A straightforward adoption might look as follows. We supply the soft-RT process with sufficient DRAM capacity to meet its timing constraints. When a batch non-RT process is invoked, we preserve the soft-RT process's key portion of pages in DRAM but swap out its cold pages so that the batch process can allocate memory and make forward progress.

Multiplexing the DRAM capacity in peak memory usage allows us to provision less DRAM; thus, in non-peak periods, less memory is left idle. However, one issue arises: how to preserve the memory pages for a soft-RT process to satisfy its timing objective? As we will show, this turns out to be a subtle issue. A soft-RT task can suffer from severe memory thrashing when it co-runs with a memory-intensive task.

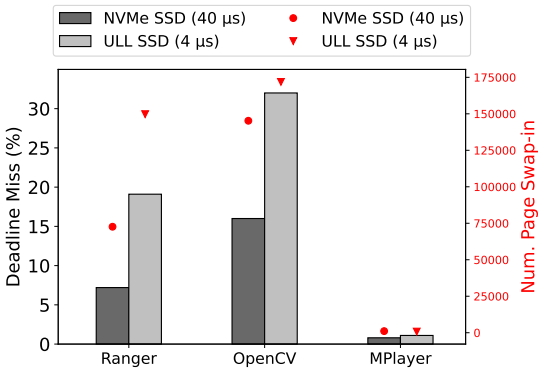
D. Challenge: Soft Real-Time Task Suffers from Memory Thrashing

We conduct experiments to investigate a hybrid memory system's behavior under peak memory usage. In each experiment, we co-run a soft-RT task (one of *Ranger*, *OpenCV*, and *MPlayer*) with the non-RT, memory-intensive task (i.e., GraphChi) in a virtual machine emulating two kinds of swap backend, NVMe SSD and ULL SSD, whose emulated latency is 40 μs and 4 μs, respectively. Table I lists the details of the investigated workloads (detailed experiment settings are in §IV-A).

Fig. 1 compares the two swap backends' behavior in three combinations of co-executed tasks. The X-axis lists the name of the soft-RT tasks. Fig. 1(a) shows the perspective of the memory-intensive task. Using ULL SSD as the swap backend improves its execution time by 5% – 14% over NVMe SSD because ULL SSD shortens page



(a) Memory-Intensive Task's Execution Time



(b) Soft-RT Task's Deadline Miss Rate

Fig. 1. Comparison of Using Different Storage Media as Swap Backend

swapping's I/O delay and helps memory-intensive tasks allocate the memory it needs more quickly. Fig. 1(b) shows the perspective of the soft-RT task. Surprisingly, using ULL SSD as the swap backend causes as much as 16% more missed deadlines. This is because the shortened I/O delay also causes more soft-RT task's pages to be evicted when the memory-intensive task is running while the soft-RT task is in the sleep state waiting for its next period. Thus, the OS considers the soft-RT task's pages idle and selects them as page replacement victims. Consequently, the soft-RT task has to swap in more pages in its next period and spend more of its CPU time budget T_s in the swap-in overhead. The soft-RT task might not finish all its work before the scheduler detects T_s has been depleted and preempts the task. A missed deadline thus occurs.

E. Motivation

The memory thrashing described in §II-D stems from Linux's lack of proper integration between real-time scheduling and memory paging policy. The real-time scheduler, `SCHED_DEADLINE`, ignores the case of heavy memory multiplexing in its design. On the other hand, the memory paging policy selects victim pages without considering the page owner's real-time semantics. While real-time scheduling and memory paging are traditionally designed for separate use cases, edge server, requiring real-time guarantee yet austere in DRAM, brings up a new use case that calls for an integration of the two independently-designed components.

Motivated by this need, we aim to design a new scheme on top of a hybrid memory system that upholds real-time semantics under memory scarcity. The new scheme should satisfy the following goals. First, it should protect a soft-RT task's memory pages from swapping, but not too much so that non-RT task can acquire sufficient memory to make forward progress. Second, the protection should have light overhead to avoid consuming too much of real-time tasks's CPU budget. Third, the real-time scheduler should compensate for the lost CPU budget due to paging overhead.

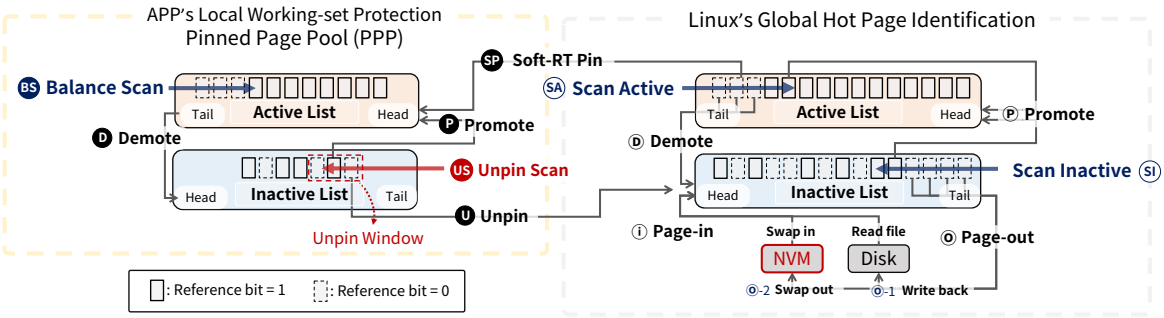


Fig. 2. Adaptive Page Pinning

III. Adaptive Page Pinning

We present Adaptive Page Pinning (APP), lightweight protection of soft-RT tasks co-running with non-RT tasks on hybrid memory. APP aims to capture and protect a soft-RT's working set memory. Instead of isolating all of the soft-RT task's pages from swapping, APP dynamically determines what pages and how many pages to *pin* in DRAM to satisfy its real-time requirement. In contrast to the full isolation approach, APP aims to leave as many free pages as possible to other non-RT tasks to improve their throughput.

We structure our introduction to APP as follows. We first briefly introduce Linux's existing memory paging policy (§III-A). Then, we present APP's lightweight working set protection called PPP (§III-B), how to size PPP appropriately, and how to adapt PPP to the evolving working set (§III-D). Finally, we show how to incorporate paging overhead into the scheduler.

A. Preliminary: How Linux Identifies Hot/Cold Pages

A memory paging policy aims to evict infrequently-accessed (i.e., cold) pages when memory runs out while ensuring frequently-accessed (i.e., hot) pages remain in DRAM. To distinguish hot pages from cold pages, Linux kernel maintains a pair of lists called "Active List" and "Inactive List," as shown on the right side of Fig. 2. Newly-allocated pages or pages fetched from disks are pushed to the head of "Inactive List" because their access frequency is yet to be determined (Fig. 2-ⓘ). When the system is under memory pressure, Linux triggers the Page Reclamation (PR) routine to reclaim cold pages and provide free pages for other processes. PR scans from the tail of the Inactive List (Fig. 2-(SI)) to reclaim two types of relatively cold pages: file-backed and anonymous pages. A clean file-backed page can be directly released; a dirty one must be written to disk (Fig. 2-Ⓞ-1). On the other hand, an anonymous page is swapped out to a low-latency NVM (Fig. 2-Ⓞ-2). When the number of reclaimed pages lags behind the system's memory demand, PR further scans from Active List's tail (Fig. 2-(SA)), demoting cold pages to Inactive List for further reclamation (Fig. 2-Ⓓ), and promoting hot pages to Active List's head (Fig. 2-Ⓟ).

Linux determines page migration between Active/Inactive List through the *Second Chance* heuristics, which relies on a per-page *reference bit* that is set when the page is accessed and unset when the page is scanned by PR. A page in Inactive List is promoted to Active List only when it is accessed twice (Fig. 2-Ⓟ). The first access sets the reference bit without promotion. The second access enacts the promotion only if it finds the reference bit set. If the second access occurs too late, PR might have unset the reference bit, preventing the promotion. Similarly, a page in Active List is demoted to Inactive List (Fig. 2-Ⓓ) if it remains idle when a PR is initiated.

B. Pinned Page Pool (PPP)

The root cause of the memory thrashing observed in §II-D is that the soft-RT task's locally hot pages might look globally cold. Thus, an isolated access frequency accounting is needed. Based on this observation, for each soft-RT task, we track its local access frequency and preserve its working set memory with a data structure called Pinned Page Pool (PPP), which also consists of an "Active List" and an "Inactive List".

When PR is demoting pages from Active List (Fig. 2-(SA)), we intercept pages belonging to any soft-RT task and insert them into the Active List of the corresponding soft-RT's PPP, preventing their demotion (Fig. 2-(SP)). Note that we choose not to insert them into PPP's Inactive List because Linux's Second Chance heuristics already certifies them as locally hot, compared to the soft-RT task's other pages in the global Inactive List. This strategy can keep pages that are both locally and globally *cold* from entering PPP, effectively reducing PPP's overhead.

C. How many pages to pin?

Pinning too many pages might leave too little free memory for non-RT tasks to function properly. Thus, a bounded capacity (the maximum number of pages PPP can pin) is required. The capacity should come close to the soft-RT task's instantaneous working set size. Fig. 3 shows the relationship between PPP's capacity and the deadline miss rate of a soft-RT task (i.e., OpenCV) when it co-runs with a non-RT task (i.e., Graphchi). The deadline miss rate converges to zero as the capacity expands up to 7000, which roughly corresponds to the soft-RT's working set size. In practice, the working set might rapidly shrink or expand. Hence, a static capacity might be inaccurate. Meanwhile, dynamically obtaining the capacity through periodic profiling is too heavyweight. To accurately adjust PPP's capacity in low overhead, we use an iterative approach, which utilize readily-available information from the real-time scheduler.

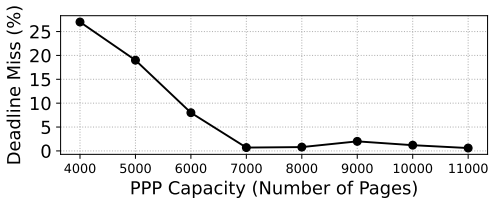


Fig. 3. Deadline miss converges to zero as PPP accommodates more pages

When the user registers a new soft-RT task, S, we initialize its PPP's capacity to an initial value (7000 is used in this work). We run Alg. 1 to adjust the capacity when S starts a new period. S's PPP capacity will be updated based on the information collected from S's previous period. In addition, we retrieve three input variables of Alg. 1 from `SCHED_DEADLINE`: T_s (S's CPU budget, which is the maximum CPU time it can spend in one period), Q (S's actual running time in its previous period), and M (whether or not S meets the deadline in its previous period).

If S missed the deadline and some of the pages S accessed need to be swapped in ($N_{swap} > 0$), then we increase the capacity by N_{swap} because the old PPP has not fully captured S's working set. On the other hand, if S met the deadline, we stop expanding the capacity. If S's running time is less than its CPU budget to a certain degree ($Q < T_s * \alpha$ where α is set to 0.9 in this work), we shrink the capacity by a small factor β. By expanding quickly but shrinking slowly, we aim to capture the current working set quickly, yet also carefully converge to the optimal capacity.

Algorithm 1: Adjustment of PPP's capacity Size

Input: C, previous capacity
Input: N_{swap}, number of swapped-in pages in previous period
Input: T_s, soft-RT task's CPU budget
Input: Q, previous period's running time
Input: M, whether deadline is missed in previous period
Parameter: α, minimum running time factor
Parameter: β, shrink factor
Output: C', new capacity

if *M is true* (deadline is missed) **and** $N_{swap} > 0$ **then**
  $C' \leftarrow C + N_{swap}$;
else if *M is false* (deadline is met) **and** $Q < T_s * \alpha$ **then**
  $C' \leftarrow C * \beta$;
else
  $C' \leftarrow C$;
return C';

D. Adopt to shifting working set

A soft-RT's working set might change. Pinned pages might turn cold and must be unpinned and evicted from PPP to free up memory. Thus, when we detect PPP's capacity has been reached in Fig. 2-SP, we run Alg. 2 to unpin pages that might have fallen out of the working set.

In Alg. 2, we first ensure the estimated working set is not oversized by conducting a balance scan (Fig. 2-BS), which demotes pages from the tail of PPP's Active List (Fig. 2-D) until Active List is shrunk below a certain ratio γ of Inactive List. γ should be well above 1 to prevent thrashing the working set; we set $\gamma = 4$ in this work. Unlike Linux's demotion (Fig. 2-Ⓓ), the balance scan demotes pages and unsets their reference bits without checking the reference bit to speed up the unpinning later.

Next, we scan a small window of pages from the Inactive List's tail (Fig. 2-US) and attempt to unpin potentially cold pages. Like Linux's Second Chance heuristics, a scanned page is promoted or unpinned based on its reference bit. A hot page will not be unpinned because it will likely have a set reference bit within the time interval after its demotion. If a page is selected as victim, it is moved to the tail of Linux's global Inactive List. The unpinned page remains in DRAM, but it is no longer protected from a swap-out. If no page is eligible for unpinning during the unpin scan, we directly unpin the next page following the window, and migrate it to the global Active List's tail. If we accidentally migrate a hot page, it will re-enter PPP in the next PR scan.

The unpin window size is kept small to minimize the overhead of an unpin scan, but not so small that no page can be unpinned. Thus, we initialize the scaling factor ϵ to 2^{-10}, double it if no pages are eligible for unpinning, and halve it otherwise.

E. Paging-aware Deadline Scheduler

Recall the counter-intuitive observation from §II-D that using a faster storage device as the swap backend will aggravate missed deadlines because the increased swap-in overhead consumes too much of a soft-RT task's CPU budget, causing premature preemption. When a soft-RT task, S, accesses a swapped page, the CPU triggers a page fault and transitions the control to the OS. The OS performs the following steps on behalf of S. 1) Check if the desired page is already in the *swap cache* 2) Allocate free space for the desired page. 3) Submit the read request to the block layer for I/O scheduling. 4) Put S to sleep. 5) Context switch. `SCHED_DEADLINE` stops deducting a task's CPU budget until the task has transitioned to the sleep state. So, S needs to pay the combined kernel overhead from step 1 to step 3. According to our profiling, the kernel overhead is usually under 10 microseconds. However, step 2 can take hundreds of microseconds

Algorithm 2: Window-Based Unpinning

Parameter: γ, ratio of Active List & Inactive List size
Parameter: ϵ, unpin window scale factor

// Fig. 2-BS: balance scan
while len($L^{PPP}_{inactive}$) == 0 **or** len(L^{PPP}_{active})/len($L^{PPP}_{inactive}$) > γ **do**
  $P \leftarrow$ pop_tail(L^{PPP}_{active});
  reference_bit(P) $\leftarrow 0$;
  push_head($L^{PPP}_{inactive}$, P);

// Fig. 2-US: unpin scan
found $\leftarrow$ False;
unpin_window_size $\leftarrow$ len($L^{PPP}_{inactive}$) $* \epsilon$;
for $i \leftarrow 1$ **to** unpin_window_size **do**
  $P \leftarrow$ pop_tail($L^{PPP}_{inactive}$);
  if reference_bit(P) == *0* **then**
    push_head($L^{Global}_{inactive}$, P);
    found $\leftarrow$ True;
  else
    push_head(L^{PPP}_{active}, P);
if found == False **then**
  $P \leftarrow$ pop_tail($L^{PPP}_{inactive}$);
  push_head(L^{Global}_{active}, P);
  $\epsilon \leftarrow min(1, \epsilon * 2)$;
else
  $\epsilon \leftarrow \epsilon * 0.5$;

when the system is in memory scarcity because memory allocation will first require running the time-consuming Page Reclamation (PR) routine introduced in §III-A. The kernel overhead also depends on the memory access pattern, which influences the swap cache's hit rate.

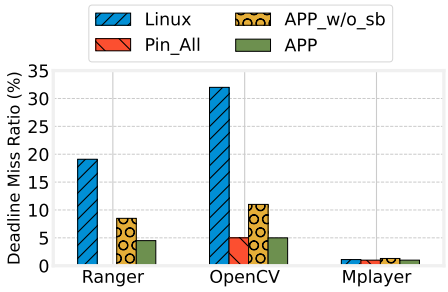
Algorithm 3: Swap-Budget Adjustment

Input: t, the previous swap budget
Input: K_{swap}, profiled swap-in overhead in previous period
Input: N_{swap}, number of swap-in events in previous period
Input: M, whether deadline is missed in previous period
Input: $\hat{T}$, previous period's remaining CPU budget
Parameter: λ, swap budget deduction factor
Output: t', the adjusted swap budget

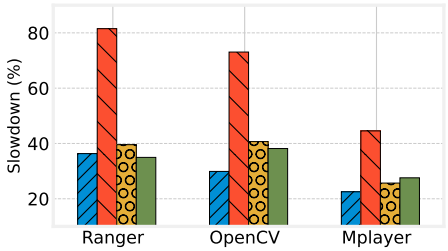
if *M is true* (deadline is missed) **and** $N_{swap} > 0$ **and** $\hat{T} == 0$ **then**
  $t' \leftarrow t + K_{swap}$;
else if *M is false* (deadline is met) **and** $t * \lambda < \hat{T}$ **then**
  $t' \leftarrow t * \lambda$;
return t';

To prevent premature preemption, we compensate S with a *swap budget*, t, and adjust t with Alg. 3 at the beginning of a new period to ensure it reflects the actual kernel overhead. We add a running counter for each soft-RT task to record the time spent in the swap-in handler (the counter incurs negligible overhead). The counter is provided as one of the input variables of Alg. 3, K_{swap}, and reset after the swap budget adjustment.

Suppose S missed the deadline in the previous period. Further-



(a) Deadline miss ratio



(b) Execution time

Fig. 4. Soft-RT task's deadline miss versus non-RT task's slowdown

more, S experienced swap-in events in the previous period, and its CPU budget $\hat{T}$ ran out. In that case, we attribute the missed deadline to the kernel overhead and compensate S with an additional swap budget of K_{swap}. The assumption is that the kernel overhead in the present and the previous period are likely correlated. When the assumption fails, t might be oversized, and S might be given too much CPU budget. This is undesirable as S might over-run and starve other tasks. Thus, when we detect that S has too much remaining budget from the previous period, we deduct t with a factor λ (λ is set to 0.8 in this work).

IV. Evaluation

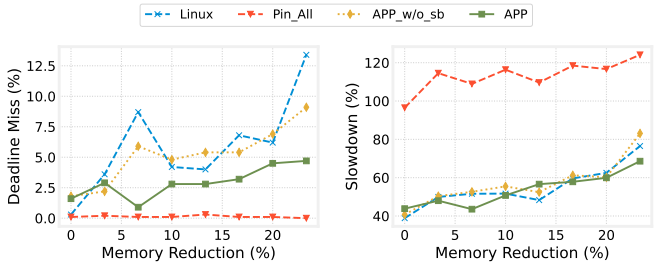
A. Experiment Setup

• **Swap backend emulation:** We evaluate APP with FEMU, a high-fidelity full-system emulator based on the QEMU hypervisor. FEMU exposes a block device to the guest OS with configurable I/O microsecond-level latency. It does so by emulating the block device with a RAM disk, intercepting I/Os, and delaying the intercepted I/O with a specific duration. To emulate an ultra-low latency storage backend, we modified FEMU to support a read latency down to 4 microseconds by removing FEMU's virtualization-specific overhead, such as VM-Exit.

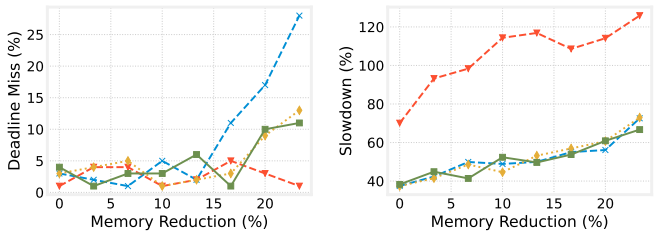
• **Workloads:** We select one representative soft-RT workload from three areas: machine learning, image classification, and video streaming, listed as follows. (1) Ranger, a real-time bank market prediction framework based on random forest. (2) OpenCV, an image processing framework to conduct image classification. (3) MPlayer, a video streaming player. We co-run these soft-RT workloads with the non-RT, memory-intensive workload, GraphChi, a graph processing engine running graph random walk. Table I provides the selected workloads' detailed properties.

TABLE I
Workload Characteristics

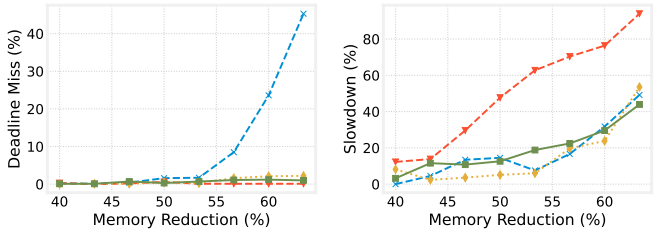
Workload	Ranger	OpenCV	MPlayer	GraphChi
Soft Real-time	Yes	Yes	Yes	No
Resident Set Size (RSS)	68%	54%	26%	136%
Running Time (isolated)	4.5	84	30	29.7
T_s (budget) (ms)	5	120	15	N/A
I_s (period) (ms)	30	300	30	N/A



(a) Ranger (soft-RT) + GraphChi (non-RT)



(b) OpenCV (soft-RT) + GraphChi (non-RT)



(c) MPlayer (soft-RT) + GraphChi (non-RT)

Fig. 5. Sensitivity Analysis on Memory Overcommitment

The "Resident Set Size (RSS)" row represents the amount of physical memory a workload ever accesses relative to the total usable memory size. We use Linux's CGroup to precisely control the usable memory size. Note that we supply the soft-RT tasks with sufficient memory (i.e., RSS < 1) but configure GraphChi to overcommit the memory (i.e., RSS = 136%). The "Running Time (isolated)" row represents the CPU time of running a soft-RT workload in isolation. We must configure the budget T_s to be larger than the running time in isolation.

• **Baseline Setup:** We compare APP with three baseline approaches. (1) Vanilla implementation based on Linux 5.11.10 (denoted as "Linux"). (2) Same as the vanilla implementation, but all pages belonging to a soft-RT process are isolated from page reclamation by making the `mlock_all` system call at the soft-RT process's initial stage (denoted as "Pin_All"). (3) APP without swap-budget (denoted as "APP_w/o_sb").

B. Results

1) Deadline Miss Ratio versus Performance: Fig. 4(a) shows the deadline miss ratio of the three evaluated soft-RT workloads when they co-run with the non-RT task, GraphiChi. Fig. 4(b) shows GraphChi's slowdown (relative to running in isolation) when it co-runs with the soft-RT tasks on the X axis. We compare APP to the three baselines.

• **APP versus "Pin_All":** "Pin_All" has an excellent deadline miss ratio of 0%, 5%, and 1%. This is expected as "Pin_All" prevents all of the soft-RT task's pages from memory reclamation and enjoys less swap-in overhead. However, this comes at the price of degrading GraphChi's performance. Unable to allocate sufficient memory, GraphChi suffers from memory thrashing, resulting in 82%, 73%, and 44% slowdown. In contrast, APP does not slow down GraphChi too much more than "Linux," which mixes soft-RT's working set from non-RT tasks' memory pages. In other words, APP can pin an

adequate amount of the SRT's key memory pages with Alg. 1, and releases sufficient free memory for non-RT tasks.

• **APP versus "Linux":** MPlayer has close to zero deadline miss in all configurations because it accesses memory sequentially and rarely triggers swap-in. In contrast, OpenCV and Ranger are more vulnerable to memory thrashing under "Linux" because they tend to reuse more past memory pages that might have been swapped out. Compared to "Linux", APP reduces OpenCV's and Ranger's deadline misses by 15% and 27%, respectively, demonstrating its effective working set protection.

• **APP versus "APP_w/o_sb":** While PPP can effectively protect a soft-RT task's working set from swapping out, some less frequently-accessed pages of the soft-RT task might still be swapped out. When re-accessed, their swap-in kernel overhead might still cause premature preemption and deadline misses. Compared to "APP_w/o_sb", APP reduces OpenCV's and Ranger's deadline misses by 4% and 6%, respectively. This indicates that the paging-aware scheduler design can alleviate the deadline misses caused by premature preemption.

2) Sensitivity Analysis of Memory Overcommitment: When less DRAM is put into a hybrid memory system, the system cost is reduced, and less energy is consumed in non-peak periods. However, the increased degree of memory overcommitment might worsen memory thrashing in peak periods. To evaluate APP's sensitivity to varying degrees of memory overcommitment, we co-run the same set of workloads but reduce their usable memory size relative to a base size of 300MiB.

Fig. 5 show the soft-RT tasks' deadline miss on the left and GraphChi's slowdown on the right at varying degree of memory reduction. As system memory decreases under "Pin_All", GraphChi suffers from worsened memory thrashing. Most memory is dedicated to soft-RT tasks, leaving less usable DRAM capacity for non-RT tasks. Meanwhile, under "Linux", it is the soft-RT tasks that suffer from memory thrashing. When memory is heavily overcommitted, Linux invokes PR more frequently, clearing more memory pages' reference bits and making them more likely to be swapped out by the subsequent PR scan.

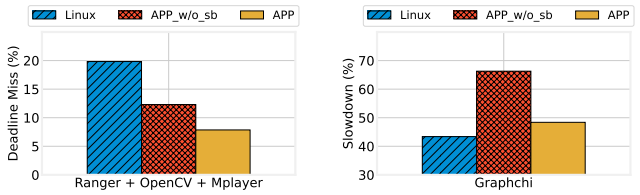
Contrary to "Linux", APP isolates a soft-RT's working set in PPP, keeping them safe from the increased PR scans. As soon as the soft-RT's working set stabilizes, APP rarely needs to run Alg. 2 to exchange memory pages with the global Inactive List. As a result, APP can bound the deadline miss rates of Ranger, OpenCV and MPlayer below 5%, 11% and 1%, respectively, when the memory is reduced by 25% or more.

3) Multiple Co-running Soft-RT Tasks: Recall that each soft-RT task has a dedicated PPP independently adjusted to capture the soft-RT task's working set. To evaluate the resource competition among multiple soft-RT tasks, we co-run Ranger, OpenCV MPlayer simultaneously with GraphChi.

Fig. 6(a) shows the soft-RT tasks' combined average deadline miss rates, and Fig. 6(b) shows GraphChi's slowdown. Note that "Pin_All" is omitted because the combined RSS of all tasks significantly exceeds the amount of physical memory, and they will fail in "Pin_All". By independently managing each soft-RT task's working set, APP reduces the overall deadline misses by 12% over "Linux". Even with multiple PPPs, APP still manages to incur no more than a 5% slowdown to GraphChi compared to "Linux".

V. Related Works

Real-time scheduling and memory management are traditionally designed in a separate context. Most real-time scheduling designs assume ample memory and focus on schedulability. Meanwhile, most hybrid-memory management designs are optimized for large-scale data center workloads [1], [2]. The emergence of edge computing has spurred recent works to co-design real-time scheduling and memory management. Aswathy et al. [8] observe that soft real-time tasks can miss deadlines when they access a slow memory media (i.e., PCM). Their solution is a memory scheduler aware of the latency discrepancy. Unlike their work, APP focuses on ULL-based storage-level paging instead of memory scheduling. Lee et al. [9] identify frequently-accessed pages of a soft-RT task with a profile-guided approach and determine the optimal number of pages to pin to meet the deadline. APP chooses not to introduce a separate profiling module for better compatibility and lower overhead. By integrating with



(a) Multiple soft-RT tasks (b) non-RT task

Fig. 6. Co-running three soft-RT tasks with a non-RT task

two of Linux's battle-tested components (i.e., `SCHED_DEADLINE` and Active/Inactive List), APP aims to be a lightweight plugin adoptable by existing systems. Liu et al. [10] propose FastResponse to meet latency-sensitive services' QoS requirements on a ULL-based hybrid memory sytem. FastResponse optimizes for file-system, CPU scheduler, and I/O scheduler. On the other hand, APP is a co-design between memory management and real-time scheduler. The two works are both integrated with Linux and thus can be combined for further system enhancement.

VI. Conclusion

We propose APP, a novel integration of real-time scheduling and memory management that enables soft real-time tasks to run safely on densely-populated systems using Ultra Low-Latency storage as swap backends. Through lightweight isolation of soft-RT task's working set memory, APP reduces deadline miss rates by as much as 27% under severe memory scarcity while reclaiming as much free memory as possible for other processes. We aim as future work to eliminate the kernel overhead on the memory paging's critical path for faster page exchange with ULL storages.

VII. Acknowledgement

This work was supported in part by National Science and Technology Council under grant nos. 108-2221-E-002-062-MY3, 111-2223-E-001-001, 111-2923-E-002-014-MY3, 111-2221-E-001-013-MY3, and 112-2927-I-001-508 and Academia Sinica under grant nos. AS-IA-111-M01 and AS-GCS-110-08.

References

[1] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *ASPLOS*, 2019, p. 317–330.

[2] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *ASPLOS*, 2022, p. 609–621.

[3] Y. Zhang, "Workload consolidation in alibaba clusters: The good, the bad, and the ugly," in *SoCC*, 2022.

[4] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana, "Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds," in *USENIX ATC 20*, 2020, pp. 927–942.

[5] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue, "Acclaim: Adaptive memory reclaim to improve user experience in android systems," in *USENIX ATC*, 2020, pp. 897–910.

[6] G. F. Oliveira, S. Ghose, J. Gómez-Luna, A. Boroumand, A. Savery, S. Rao, S. Qazi, G. Grignou, R. Thakur, E. Shiu, and O. Mutlu, "Extending memory capacity in consumer devices with emerging non-volatile memory: An experimental study," 2021.

[7] D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, "Non-volatile memory based page swapping for building high-performance mobile devices," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1918–1931, 2017.

[8] N. Aswathy, H. K. Kapoor, and A. Sarkar, "A soft real-time memory request scheduler for phase change memory systems," in *RTCSA*, 2021, pp. 109–118.

[9] D. Lee, J.-C. Kim, C.-G. Lee, and K. Kim, "mrt-plru: A general framework for real-time multitask executions on nand flash memory," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 758–771, 2013.

[10] M. Liu, H. Liu, C. Ye, X. Liao, H. Jin, Y. Zhang, R. Zheng, and L. Hu, "Towards low-latency I/O services for mixed workloads using ultra-low latency ssds," in *ICS*, 2022.