Exploring Synchronous Page Fault Handling

Yin-Chiuan Chen, Chun-Feng Wu, *Member, IEEE*, Yuan-Hao Chang, *Senior Member, IEEE*, and Tei-Wei Kuo, *Fellow, IEEE*

***Abstract*—The advance of nonvolatile memory in storage technology has presented challenges in redefining the ways in handling the main memory and the storage. This work is motivated by the strong demands in effective handling of page faults over ultralow-latency storage devices. In particular, we propose synchronous and asynchronous prefetching strategies to satisfy process executions with different memory demands in supporting of synchronous page fault handling. An adaptive CPU scheduling strategy is also proposed to cope with the needs of processes in maintaining their working sets in the main memory. Six representative benchmarks and applications were evaluated. It was shown that our strategy can effectively save 12.33% of the total execution time and reduce 13.33% of page faults, compared to the conventional demand paging strategy with nearly no sacrificing of process fairness.**

***Index Terms*—Context switch, data prefetching, killer microsecond, page faults, process scheduler, synchronous I/O completion designs, ultralow-latency (ULL) devices.**

I. Introduction

THE ADVANCE of nonvolatile memory (NVM) technology has introduced ultralow-latency (ULL) storage devices and significantly shrink the performance gap between the main memory and the storage [1], [2], [3], [4], [5]. In contrast of traditional storage devices [e.g., conventional solid-state drives (SSDs)], ULL storage devices are one of the promising candidates to be configured to a swap area for extending main memory capacity. It further triggers the reconsideration of asynchronous page faults used in the common practice because the context switching time in handling such page faults is now considered pretty long [6], [7], [8], [9]. In the above studies, IBM and Intel researchers suggested to consider synchronous page fault handling to have CPUs busy waiting for handling processes' page faults. The important observations motivate this work in exploring the essential technical issues in handling synchronous page faults with the supports of ULL storage devices, such as processes' prefetching and CPU scheduling.

Although synchronously handling page fault can get rid of the context switch overhead, systems still need to spend time on waiting for the completion of the page movement between memory and ULL storage devices. To hide the cost of page movement, prefetching multiple pages simultaneously from ULL storage devices to DRAM is one of the effective solutions. Note that the performance costs of data prefetching can be hidden by the high parallelism provided by both ULL storage devices and peripheral buses. MemPod [10], a lightweight hybrid memory management solution, adopts the heuristic majority element algorithm [11] to monitor access hotness for each page and relocates hotter pages to the fast memory (e.g., DRAM). MemPod works well for CPU-intensive applications, but cannot show good performance on memory-intensive applications. The reason is that, MemPod relies on historical information to make relocation (or migration) decision, but hotter pages in memory-intensive applications have a longer reuse distance than that in CPU-intensive applications. In this case, MemPod and least recently used (LRU)-based managements show similar performance improvement when managing memory-intensive applications. In addition to only reply on access hotness, data prefetching solutions relying also on process access behaviors (e.g., access patterns on physical or virtual address space) can provide better performance. Please note that, the baseline setting of all experiments in this work adopts the LRU-based management.

To prefetch pages with considering access patterns on physical address space, Saxena and Swift [12] proposed FlashVM to prefetch data stored in their flash-memory storage device's nearby physical addresses to the DRAM-base main memory. However, Badam and Pai [13] pointed out that the performance of FlashVM could drop seriously while the total size of the flash-memory storage device significantly exceeds the DRAM size. In addition to the physical-address-space prefetching, Wu *et al.* [14] showed the strong spatial locality in the virtual address space of the main memory, and proposed a joint management framework to prefetch sequential pages in the virtual address space (denoted as *virtual prefetching* for the rest of this article). Moreover, with enjoying the high bandwidth and parallelism provided by both DRAM and ULL devices, Wu *et al.* [6] proposed an aggressive virtual-address-space prefetching solution to prefetch more pages with being aware of the device characteristics (e.g., parallelism and response

Manuscript received 24 July 2022; accepted 26 July 2022. Date of current version 24 October 2022. This work was supported in part by the Academia Sinica under Grant AS-IA-111-M01, Grant AS-GCS-110-08, and Grant AS-CDA-107-M05; and in part by the Ministry of Science and Technology under Grant 111-2223-E-001-001 and Grant 111-2221-E-001-013-MY3. This article was presented in the International Conference on 2022 and appears as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding authors: Yuan-Hao Chang; Tei-Wei Kuo.)*

Yin-Chiuan Chen is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan (e-mail: r08944011@csie.ntu.edu.tw).

Chun-Feng Wu is with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan, and also with the Department of Computer Science, Harvard University, Cambridge, MA 02138 USA (e-mail: cfwu417@cs.nycu.edu.tw).

Yuan-Hao Chang is with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (e-mail: johnson@iis.sinica.edu.tw).

Tei-Wei Kuo is with the College of Engineering, City University of Hong Kong, Hong Kong, and also with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan (e-mail: ktw@csie.ntu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2022.3197517

time) by modifying modern huge page management. While the purpose of prefetching is to reduce the possibility of page faults, the real concerns is whether the working set of the next running process is properly maintained in the main memory. Even though the past study [6], [7], [8], [9] proposed to consider synchronous page-fault handling because of the advantages of ULL storage devices over context-switching overheads, the technical challenges in such a successful design is what should be prefetched when a synchronous page fault occurs? What should be done if some of the working set of the next running process is still missing?

This work is motivated by the tremendous improvement of the access latency of NVM-based storage. We explore the demand page designs with synchronous page fault supports. In particular, we shall propose the essential strategies of such designs in: 1) immediate synchronous page prefetching; 2) memory-demand-adaptive CPU scheduling; and 3) asynchronous page prefetching. To improve the efficiency in handling page faults, our immediate page prefetching design smartly selects and prefetches an appropriate set of pages with the considerations of page access behaviors during the occurrence of each page fault. To improve the hit rate of these prefetched pages, processes shall be scheduled with the considerations of their memory demands. Our adaptive CPU scheduling design shall satisfy the memory demands of different processes by adjusting both the scheduling frequency and time slices with the considerations of the fairness of using CPU resources. Specifically, memory-intensive processes may receive a longer time slice to fit (or reconstruct) their large working sets in the memory and nonmemory-intensive processes may be scheduled more frequently to refresh and keep their working sets in the memory. Note that our design will only deal with noninteractive processes, and the priority of interactive processes is, thus, not affected. To further optimize the overall system performance by minimizing the overhead in reconstructing working sets, our asynchronous page prefetching design shall preload working sets belonging to memory-intensive processes in the background while executing nonmemory-intensive processes. The evaluation results show that our strategy can improve the execution time by 12.33% and reduce 13.33% of page faults, compared to the conventional demand paging strategy. We will show that our strategy is fair where all processes have similar time on using CPU resources.

The remainder of this article is organized as follows. Section II elaborates the page fault handler and shows the impact of the working set contention. Section III provides the design concept and implementation of the adaptive synchronous page fault (A-SPF) handler. Section IV evaluates the proposed strategy. Finally, Section V concludes this article.

II. Background, Observation, and Motivation

A. Background: Asynchronous and Synchronous Page Fault Handler

To increase memory capacity with lower costs, virtual memory management [15] is a common solution to extend memory space by using storage devices and the demand paging strategy is used to move the data from storage to memory devices when the data is required. However, it is inefficient to let CPUs directly access the data stored in storage devices because the latency gap between CPUs and storage devices is huge. If the required data is not presented in the memory devices, the CPU will automatically generate an exception, called page fault. The operating system (OS) will then call the demand paging strategy to run an asynchronous page fault handler to manage the exception [16]. The top figure in Fig. 1(a) illustrates the procedure flow of the asynchronous page fault handler. Once raising a page fault, the CPU will switch from the user mode to the kernel mode to run the asynchronous page fault handler (❶). The handler will first verify the legality (or permission) of the virtual address associating to the page fault (❷). If it is legal, the handler will then check that the address points to the file system or the swap area. Note that this work only focus on the case where the address points to the swap area (or memory extension area). After that, OS allocates a page on the memory device and configures the direct memory access (DMA) controller to move the data from the storage device to memory device (❸). Avoid letting CPU wait for the data movement synchronously, the OS runs context switch to switch-in other processes decided by the process scheduler (❹) and the DMA will move the data asynchronously in the background (❺). Once the data is moved to the memory device, the DMA interrupts the CPU to cleanup the page fault and the OS will help to update the page table entry.

Asynchronous page fault handler is widely adopted for several decades, but it becomes inefficient when the latency gap between CPUs and storage devices is greatly shrunk. In recent years, the advance of manufacturing technologies boosts the response time of storage devices from several milliseconds (ms) to several microseconds (μs) [17], [18], [19], where these devices are called ULL devices. For example, the response time of both Samsung Z-NAND SSDs and Intel Optane SSDs is around 3–10 μs [2], [3], [5]. However, as reported by Intel and IBM, the overall context switch time could be longer than 5–10 μs on a general-purpose machine [7], [8], [9], and it could even be around 20 μs on an embedding system [6]. Under this trend, it is inefficient to handle page faults asynchronously by executing a context switch. At this turning point, researchers from Intel and IBM advocate to adopt the synchronous page fault handler [6], [7], [9] so as to let CPUs directly enjoy the fast response time of ULL devices. The bottom figure in Fig. 1(a) illustrates the procedure flow of the synchronous page fault handler. Once a CPU raises a page fault, it will switch to the kernel mode to run the synchronous page fault handler (❶). The step of checking page fault area is same as the asynchronous page fault handler (❷). After that, the CPU will directly move the data by itself without the help from the DMA (❸). Once completing the movement, the mapping between virtual and physical addresses will be established and updated to corresponding page table entries. Finally, the CPU will switch back to the user mode and resume executing the original process.

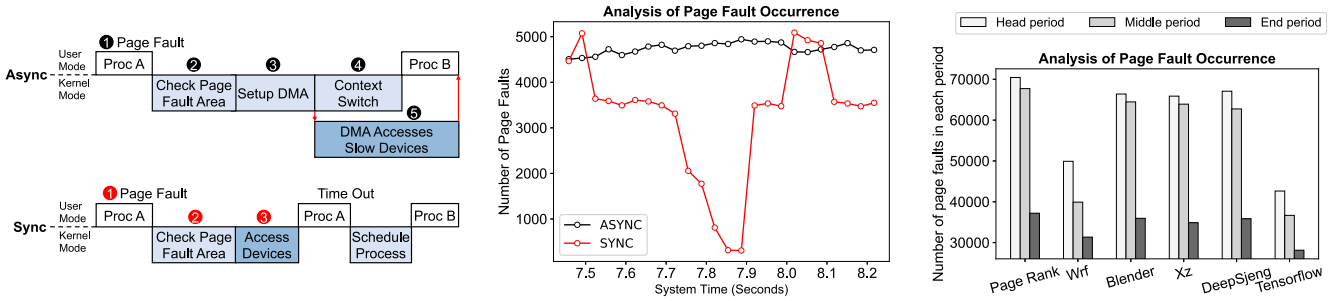

Fig. 1. Mechanism of page fault handling. (a) Asynchronous and synchronous page fault handling. (b) Number of page faults versus time. (c) Page fault distribution among the execution timeslice.

B. Observation: Working Set Contention

Switching the page fault handler from asynchronous to synchronous is a start of renovating the demand paging strategy to manage ULL storage devices. The legacy process scheduler, such as the completely fair scheduler (CFS) used in Linux [20], is designed upon the asynchronous page fault handler and, thus, cannot easily fit with the synchronous page fault handler. There are usually two ways to trigger the process scheduler to select and switch processes, that is the context switching and execution timeout. Systems with the synchronous page fault handler will not call context witching and, thus, processes will only be rescheduled after they use up their allocated time slices, as shown in the bottom figure in Fig. 1(a). Thus, each process may occupy more memory space than before and aggressively kick out working sets belonging to other processes to storage devices, where we call this issue "working set contention." Even worse, the working set contention becomes more serious when systems run memory-extensive processes (e.g., graph processing [21], [22]), which suffer from more page faults. Systems might need to reconstruct working sets by moving pages kicked out by other processes to the memory devices and, thus, suffer from intensive page faults during the period where the contention is serious.

To show the impact of the working set contention, we build a in-house multiprogramming-based simulator to monitor the number of page faults under adopting synchronous and asynchronous page fault handling, respectively. In this experiment, we run six processes with using the CFS to schedule them. Specifically, they are Wrf from SPEC CPU® 2006 [23], Blender, Xz, DeepSjeng from SPEC CPU® 2017 [24], a deep learning framework (Tensorflow [25]), and a graph application (Page Rank on Graphchi [26]). First, according to our results, with using only asynchronous and only synchronous page fault handling, the total execution time is 82.3 and 62.3 s, respectively. That is, using the synchronous page fault handling can save around 24.3% of the total execution time compared with using the asynchronous page fault handling.

In addition to provide the overall execution time, we take a closer look into a certain time range, as shown in Fig. 1(b), where the *y*-axis presents the accumulated number of page faults in each time window (i.e., 33 ms) and the *x*-axis indicates the system time. Using asynchronous page fault handler shows relatively stable results in terms of the number of page faults. The reason is that the most frequently used pages belonging to each processes can be kept in memory devices. On the other hand, if the system adopts the synchronous page fault handler, the number of page faults fluctuates severely over time. For example, we can observe a high occurrence peak of page faults around 7.5 s, but there are extremely few page faults between 7.8 and 7.9 s. The peak is caused when processes are just scheduled to run but most of their working sets are kicked out from memory devices due to the working set contention. Their working sets can be reconstructed after running for a while, and then the number of occurrence page faults will greatly drop.

We also provide some break down results in Fig. 1(c) to show the overhead on reconstructing working sets under using synchronous page fault handler. In this experiment, time slices allocated to each process are divided into three equal time periods, head period, middle period, and end period. The *y*-axis presents the accumulated page faults occured in each period. It is obvious that all six processes suffer from more page faults in the head period, where the process is just scheduled and run. For some memory-intensive processes requiring longer time to reconstruct their working sets still suffer from intensive page faults in the middle period. The occurrence of page faults significantly drops in the end period of all processes because their working sets (or most frequently accessed pages) have been reconstructed in memory devices.

C. Motivation

As suggested by Intel and IBM, switching the page fault handler from asynchronous to synchronous becomes a trend for the demand paging strategy to manage ULL storage devices. With using synchronous page fault handler, processes will not be switched out and will synchronously wait for the completion of handling each page fault. In this case, the system will suffer from the issue of working set contention because each process may occupy more memory space than before and aggressively kick out working sets belonging to other processes to storage devices. This observation motivates us to come up with a new demand paging strategy to deal with the working set contention issue so as to remove the peak of page faults and further reduce the occurrence of page faults. The technical challenge falls on how

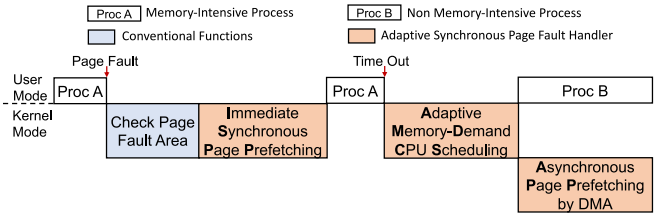

Fig. 2. A-SPF handler.

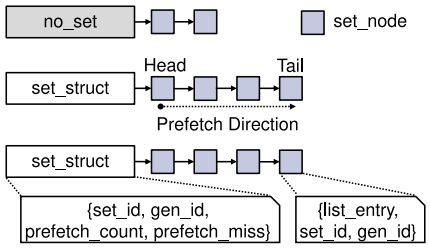

Fig. 3. Fault-aware prefetch set.

to propose a synchronous-page-based demand paging strategy to: 1) identify and smartly prefetch (or preload) pages for processes with different memory demands and 2) adaptively schedule processes to satisfy their memory demands.

III. Adaptive Synchronous Page Fault Handler

A. Overview

In this section, we will present our A-SPF handler to remove the peak of page faults and reduce the occurrence of page faults. As shown in Fig. 2, there are three main designs in our A-SPF handler. We will elaborate the three designs in Section III-B and also introduce on online predictor to classify each process as memory intensive or nonmemory intensive in Section III-C. While handling a page fault, our handler runs the immediate synchronous page prefetching design (introduced in Section III-B1) to carefully decide a few to-be-fetched pages by considering page access behaviors. The page required by the CPU and those to-be-fetched pages together can be moved together with utilizing the high transmission bandwidth provided by the peripheral buses [6]. When a process uses up all its time slices, our handler will run the adaptive memory-demand CPU scheduling design (introduced in Section III-B2) to work with the CPU scheduler to schedule the next process with taking both memory demands and the fairness of CPU occupation into consideration. If the running process is nonmemory intensive, our handler runs the asynchronous page prefetching design (introduced in Section III-B3) to notify the DMA to prefetch the working set belonging to the next-to-be-run memory-intensive process. The rationale behind this design is that nonmemory-intensive processes does not need to handle page faults too often and, thus, idle buses can be used for preloading the memory-intensive process's working set.

B. Strategy for Prefetching and Adaptive CPU Scheduling

1) Immediate Page Prefetching Design: While handling a page fault, our immediate page prefetching design fetches not only the victim page causing the fault but also those pages which caused faults in a close time period around the victim page raised a fault last time. The rationale is that pages causing faults in a close time period may probably be evicted together and then raise page faults together the next time. For example, graph processing and neural network applications usually run multiple iterations to process the same input dataset and, thus, may have similar memory access patterns across different iterations. To keep track of pages causing faults in a close time period, we propose a fault-aware prefetch set (as shown in Fig. 3). Conceptually, our prefetch set maintains multiple *set_struct* where each of them links a series of *set_node* representing pages causing page faults in a short time period. When dealing with the page fault, our prefetcher fetches the pages from the head of the *set_struct* where the *set_id* is same as the page causing the fault. Basically, there are three fields for each *set_node*: *list_entry*, *set_id*, and *gen_id*. A *list_entry* serves as the node of the doubly linked list and a *set_id* represents the set identifier associating with the corresponding page. A *gen_id* stands for the generation id, which will be introduced later. Note that the *set_struct* can be implemented by extending the page structure in the Linux kernel. Here, we will explain the details about the additional information *set_node* based on the implementation of Linux kernel 5.7. In the Linux kernel, each physical page in the system has an associated *struct page* to keep track of the page state and the related information. In our design, we propose to embed the *set_node* structure in every *struct page*. The reason behind this proposal is that we can locate the *set_node* efficiently by simply running pointer calculation when the *struct page* is extracted in the usual routine of the page fault handler. On the other hand, we can get the corresponding *page struct* via *set_node* in the same way.

When handling a page fault, our immediate page prefetching design will run a fault-aware page grouping policy (as shown in Algorithm 1) to judiciously assign a *set_id* to the page and link the *set_node* to the corresponding *set_struct*. There are two inputs in our group policy, that is the *set_node* of previous page fault (*prev*) and the *set_node* of current page fault (*curr*). There is no previous page fault before the first page fault of a process, so we simply set the *set_id* of the *curr set_node* to the *NO_SET* (steps 1 and 3). That is, the page does not belong to any *set_struct* in the fault-aware prefetch set. In addition to dealing with the first page fault, there are four possible cases for any two pages causing consecutive faults. For the case that both the *prev* and the *curr* have their own *set_id*, we will just let two pages stay in its original set (steps 4 and 5). Note that a *set_id* is valid if it is neither NULL nor the *NO_SET*. If both the *prev* and the *curr* do not contain valid *set_id*, we will create and assign a new *set_id* for both pages (steps 6–10). The remaining two cases are very similar, one page has a valid *set_id* but the other page does not. First we check if there still exists any available space from the corresponding *set_struct* (steps 11 and 14). If there is available space (i.e., *not_full*), our policy will group the two pages in to the same set by assigning *set_id* to the one that does not belong to any set (steps 12, 13, 15, and 16). Otherwise, our policy will

Algorithm 1: Fault-Aware Page Grouping Policy

input : The page structure *curr* of current page fault
input : The page structure *prev* of previous page fault

```
1  if prev is NULL then
2      curr.set_id ← NO_SET;
3      return;
4  if valid(prev.set_id) and valid(curr.set_id) then
5      return;
6  if invalid(prev.set_id) and invalid(curr.set_id)
   then
7      new_set_id = create_new_id();
8      prev.set_id ← new_set_id;
9      curr.set_id ← new_set_id;
10     return;
11 if valid(prev.set_id) and not_full(prev.set_id)
   then
12     curr.set_id ← prev.set_id;
13     return;
14 else if valid(curr.set_id) and
   not_full(curr.set_id) then
15     prev.set_id ← curr.set_id;
16     return;
```

assign the *set_id* before running a LRU-based replacement to get available space from that *set_struct*.

After assigning a *set_id* to a *set_node*, the next step of our immediate page prefetching design is to judiciously decide a suitable position of the *set_struct*, that is to insert at the head or the tail. Our design will insert the *set_node* to the head of the corresponding *set_struct* only if the *PG_ACCESSED* bit (or sometimes called *PG_REFERENCED* bit) [27] of the page is set in the page table entry. Otherwise, the page will be inserted to the tail of the fault-aware prefetch set and, thus, has a lower priority to be prefetched. Note that the *PG_ACCESSED* bit is set by the CPU when the corresponding page is accessed. If the *PG_ACCESSED* bit is not set, it represents that the page was prefetched to DRAM by our designs but it was then swapped back to ULL devices without being accessed. In this case, we can infer that the page might not be a frequently accessed page, so it will be placed at the tail.

In addition to deciding the position in the *set_struct*, it is also important to track the effectiveness of the prefetching hit ratio in the same set. Our design maintains three additional fields in every *set_struct*: 1) *prefetch_count*; 2) *prefetch_miss*; and 3) *gen_id*. *Prefetch_count* records the number of pages which are prefetched from the *set_struct* and the *prefetch_miss* records the number of pages that are swapped back to the ULL devices without being accessed after being prefetched. The *prefetch_miss* will be increased when a page is evicted and its *PG_ACCESSED* is not set. The prefetch accuracy can be calculated by using *prefetch_count* and *prefetc_miss*. More pages will be prefetched from the *set_struct* with high prefetch

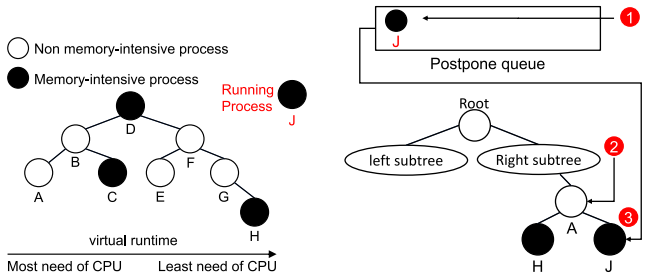

Fig. 4. Adaptive CPU scheduling design. (a) Memory-demand-aware CFS tree. (b) Delay memory-intensive processes.

accuracy, that is *prefetch_count* but low *prefetch_miss*. If the accuracy is too low, our design will mark all *set_nodes* in this *set_struct* as outdated by increasing the *gen_id* maintained in the *set_struct*. The design concept of the *gen_id* is to maintain the freshness (or generation) of *set_nodes*. The *gen_id* of each *set_node* will be set to the *gen_id* of *set_struct* once the *set_node* is updated. Our design will only prefetch the related pages where their *set_node* have the same *gen_id* with the *set_struct*.

2) Adaptive CPU Scheduling Design: The goal of our adaptive CPU scheduling design is to remove page-fault peaks, which are caused by frequently moving the evicted working sets belonging to memory-intensive processes from ULL storage devices to the DRAM-based main memory. The key concept of our design is to avoid scheduling two memory-intensive processes continuously by prioritizing and running nonmemory-intensive processes before running a memory-intensive process. We first enable the conventional CPU process scheduler to be aware of the memory-demand of each process (i.e., memory intensive or not). Then, our design lets nonmemory-intensive processes be scheduled to run more frequent so as to refresh and keep their working sets in the memory. On the other hand, our designs allocates longer time slice to memory-intensive processes so as to minimize the frequency of reconstructing their working sets. Processes can be classified as *memory-intensive* or *nonmemory-intensive* at offline by profiling their overall occurrence of page faults. Note that we will introduce an online design to periodically decide whether a running process is memory-intensive or not in Section III-C.

Our adaptive CPU scheduling design is extended from the CFS, which is the default process scheduler in the current Linux kernel. The CFS decides the next to be executed process by searching for the one with the least virtual runtime. The virtual runtime of each process is updated based on the priority and the time spent on using CPU resources. For example, a high-priority process will get a small virtual runtime and will get an even smaller virtual runtime if it occupies CPU resources for a relatively short time. With considering both the virtual runtime and memory demands, we propose a memory-demand-aware CFS tree [as shown in Fig. 4(a)], where nodes with black or white color represent a memory-intensive or nonmemory-intensive process, respectively. Processes with smaller virtual runtime will be

placed closer to the left-hand side. After the current running process J uses up all its available time slices, the system runs our design to schedule the next running process by picking the leftmost process in the tree, and that is process A. After that, our design needs to place the previous running process J back to the tree according to its virtual runtime. If the system follows the design of the original CFS, the process J will very likely be placed in the rightmost of the tree, and it will make two memory-intensive processes (i.e., H and J) to stay in nearby. In this case, when process J is scheduled the next time, most pages related to its working set might be evicted and, thus, it shall spend more time on the working set reconstruction, which causes immersive number of page faults.

To deal with this issue, we propose to schedule each nonmemory-intensive process more frequent and then allocate more time slices to memory-intensive processes. Under our design, both nonmemory-intensive and memory-intensive processes will eventually have similar time on using CPU resources. Note that while running nonmemory-intensive processes, we will run our asynchronous page prefetching design to help preload the working set for the next running memory-intensive processes in the background (more details can be found in Section III-B3). Fig. 4(b) shows the details of our design. After the memory-intensive process J uses up all its time slices, our design temporarily places the memory-intensive process J in the postpone queue when the rightmost process in the tree is memory-intensive (❶). The memory-intensive process J will stay in the postpone queue until the rightmost process in the tree becomes nonmemory intensive. For example, after the nonmemory-intensive process A uses up all its time slices, it will be placed to the rightmost position in the tree (❷), and then process J can be moved from the postpone queue to the rightmost position in the tree (❸).

Algorithm 2 is presented to show the details of the adaptive CPU scheduling design. The main goal of this algorithm is to add a memory-intensive process from the postpone queue to the memory-demand-aware CFS tree and to decide a suitable virtual runtime for the memory-intensive process. The algorithm is triggered every time when a nonmemory intensive process p uses up its available time slices and is going to be put back to the memory-demand-aware CFS tree. There are three inputs, that is, the memory-demand-aware CFS tree t, the postpone queue q, and the nonmemory-intensive process p. The algorithm first checks the to-be-inserted position of the nonmemory intensive process p. If p is not inserted to the rightmost position, we cannot pop any process from the postpone queue (steps 1–3). On the other hand, we pop an memory-intensive process *first* from the postpone queue (steps 8 and 9), if p has the largest virtual runtime than all the others in the tree t and, thus, is going to be inserted to the rightmost position. Then, our algorithm adjusts the virtual runtime of the process *first* (step 11) by overwriting it with the p's virtual runtime plus one, so as to put p in front of *first*. Also we need to record how much is the process *first* delayed (i.e., the *delay_time*) in the aspect of the virtual runtime (step 12).

In order to compensate the runtime for the memory-intensive process which is postponed, our algorithm runs (1) to decide how much of virtual runtime can the process really receive. Our algorithm will adjust virtual runtime when a memory-intensive process *mp* is on the leftmost position of the memory-demand-aware CFS tree and it is going to be scheduled in. After shrinking the virtual runtime of a process, it will receive longer time slices. Note that each process's virtual runtime also represents each process's overall CPU resource used time. CPU process scheduler (e.g., CFS) tends to level every process's virtual runtime, so as to let every process's has similar CPU resource used time. Thanks to this leveling design, even if memory-intensive processes use up more CPU resources in a short time, the overall CPU resource use time allocated to each process will be similar eventually

$$mp.vruntime = mp.vruntime - mp.\text{delay\_time}. \quad (1)$$

Note that for serving processes requiring a low response time, we suggest the users to set those processes as high-priority processes by adjusting the nice value in the Linux system. By applying our designs, high-priority memory-intensive processes will be scheduled more frequent to meet the requirement of low-response time and will also receive a longer time slice to run as much as possible on the reconstructed working set. However, we do not suggest to run too many of processes with the characteristics of both low-response time and memory intensive in the same system, because they might aggressively occupy most of CPU and memory resources. In our future works, we plan to explore the system impacts and some management strategies when there are several low-response-time and memory-intensive processes running in the systems.

3) Asynchronous Page Prefetching Design: Our A-SPF handler also comprises an asynchronous page prefetching

Algorithm 2: Pop Policy of the Postpone Queue

input : Memory-demand-aware CFS tree t
input : Postpone queue q
input : Non memory-intensive process p that will be put back to the tree t

```
// p is not inserted to the rightmost;
1  if p.vruntime < t.max_vruntime then
2      tree_insert(t, p);
3      return;

// Check if the postpone queue is empty;
4  if q.empty() then
5      tree_insert(t, p);
6      return;
7  else
8      first ← q.front();
9      q.pop();
10     old_vruntime ← first.vruntime;
11     first.vruntime ← p.vruntime + 1;
12     first.delay_time ← first.vruntime − old_vruntime;
13     tree_insert(t, p);
14     tree_insert(t, first);
15     return;
```

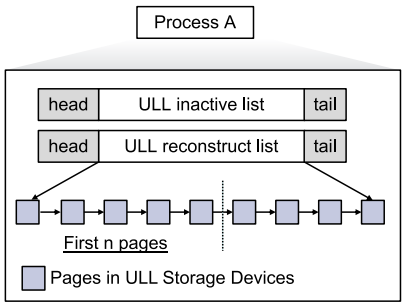

Fig. 5. Management of the ULL pages (per process).

design to minimize the intensive page faults caused by reconstructing the working set of a memory-intensive processes, which usually have a larger working set than nonmemory-intensive processes. Specifically, our design will identify pages related to the working set of memory-intensive processes by checking the page table and notify the DMA to preload the working set during running nonmemory-intensive processes. The rationale behind this design is that nonmemory-intensive processes does not need to handle page faults too often and, thus, idle buses can be used for preloading the working set belonging to the next memory-intensive process.

Before diving deep into the detail of our asynchronous page prefetching design, we will first introduce the memory management of the hybrid DRAM-ULL system in modern Linux systems, where ULL storage devices are configured as the swap area. It is expensive to keep track of every memory accesses and use an LRU policy to monitor the page access behaviors. Linux systems use an approximate way to keep track of the page access [16] by using two lists, that is, active and inactive lists. Linux systems will run a kernel thread to periodically scan referenced bits set in the page table,[1] and move all pages with their bits set between active and inactive list.

However, Linux mixes all pages belonging to different processes in both active and inactive lists. Besides, Linux pays more attention on tracking pages inside DRAM and, thus, it is hard to identify the access behaviors (e.g., access frequency) for those pages inside ULL storage devices (or the swap area). In contrast to Linux's tracking mechanism, we propose a process-centric tracking mechanism to keep track of each process's pages in the ULL storage devices. As shown in Fig. 5, we maintain two ULL page lists in the system and named ULL inactive list and ULL reconstruct list for each process, respectively. The ULL inactive list collect the pages which are only in ULL storage devices and are probably not accessed in the future. On the contrary, the ULL reconstruct list holds the pages which are in ULL storage devices and are accessed more recently. Under this design, pages in the ULL reconstruct list might have higher probability to be deemed as part of the working set.

When the process requests a page in a ULL storage device but there is no enough DRAM space, our design will run page swapping and tracking, where details are described in Algorithm 3. Our design first swaps out the page from DRAM to ULL storage devices so as to make available space on DRAM (step 1). To decide a suitable list (i.e., ULL inactive or ULL reconstruct) for maintaining the swapped page, we should find out the process owning this page and the page table entry (*pte*) assoicated with this page (steps 2 and 3). According to the *pte*, we can retrieve the access information about this page by examine the *PG_ACCESSED* bit. If this bit is unset, it means this page is not accessed for a relative long time. Hence, this page will be inserted to the ULL inactive list (steps 4 and 5). On the contrary, if the *PG_ACCESSED* bit is set, we categorize this page as one of the candidates in the working set. In this case, this page will be inserted to the head of the ULL reconstruct list (step 7) and, thus, it might be possible be brought back to the DRAM while running the asynchronous page prefetching design.

Algorithm 3: Management of ULL Pages

input : The page *page* to be evicted from DRAM to NVM
output: Page *page* is tracked by the proper list

```
1 swap_to_nvm(page) // Migrate from DRAM to NVM;
2 p ← page.owner // Find the owner of the page;
3 pte ← PTE(page) // Get the page table entry;
4 if PG_ACCESSED(pte) is 0 then
5     p.insert_head_inactive(page);
6 else
7     p.insert_head_reconstruct(page);
```

When running a nonmemory-intensive process, our asynchronous page prefetching design will fetch the first *n*th pages from the head of the ULL reconstruct list associated with the next memory-intensive process by using DMA. The next question is how many pages should be moved during the asynchronous page prefetching design, that is how to set "*n*." It is common that processes might change their access behaviors over time. For instance, the process may intensively access its working set in some time, but may change its working set in other time. To cope with this problem, our design dynamically decides a suitable to-be-prefetched amount *n*. During running each memory-intensive process, our design uses a variable called *prev_fault* to record the accumulated number of page faults. Note that *prev_fault* will be reset each time when the process is scheduled to be run. The variable *prev_fault* will be updated every time handling a page fault. It is reasonable to infer that the working set of the process is destroyed severely if *prev_fault* is high. In this case, if the process is memory intensive, then our design tends to reconstruct more pages.

C. Online Memory-Demand Predictor

Every process will have several execution phases and also their access behaviors vary from time to time. Instead of

[1]To access each virtual address, CPU will lookup the corresponding page table entry to get its mapped physical address, and will automatically set up the reference bit maintained in that page table entry.

Algorithm 4: Online Memory-Demand Predictor

input : The *period* of the time window
input : The *queue* to store the timestamp record

```
// In the page fault handler;
1 do_page_fault (error_code, address):
    // Get the timestamp of page fault;
2   timeval = gettimeofday(timeval);
    // Push into the timestamp queue;
3   queue.push(timeval);

4 get_avg_fault_count ():
5   while (queue.front() < queue.now() - period)
6     queue.pop();
7   return queue.size() / period;
```

statically assigning the label of memory intensive or not to each process, we propose an online memory-demand predictor to find out processes that are relatively memory-intensive compared with others by monitoring the occurrence history of page faults. We observe that when a memory-intensive process is executing, the system will suffer from lots of page faults. Inspired by this observation, we can dynamically judge the process status by monitoring the accumulated page faults in each fixed period time window. As shown in Algorithm 4, we retrieve the page fault timestamp in the very beginning of the page fault handler (step 2), and then store the time information in a FIFO queue (step 3). A timer is set for each process and it will interrupt the system when the time meets the period of the time window. Our predictor then calculates the average page faults within the time window (or the period) (step 7).

We will maintain a global average page fault by periodically averaging all processes' average page faults. If a process suffers from average page faults higher than the global average page fault, our predictor will classify the process as a memory-intensive process. Practically, we reserve a variable called ts_fault within every process structure which counts the number of page faults in the current time slice. When the process is context switched by the other process, we will calculate the average page faults in the previous time slice.

IV. Evaluation

A. Experimental Setup

In this section, we will evaluate the proposed A-SPF handler in both performance and fairness. Five methods are involved in this evaluation. The first one is the *baseline* which adopts the synchronous page fault handling as suggested by the IBM and Intel [7], [9]. Rather than adopting the asynchronous page fault handling, the *baseline* is more efficient because the response time of the ULL storage devices is faster than performing context switch. When the page fault occurs, the *baseline* will fetch the fault page from the ULL storage devices to the main memory (DRAM) without triggering the context switch. The second one is the *virtual prefetch* (*Virt_prefetch*) [14] which adopts the synchronous page fault handling as *baseline*. Besides, when page fault occurs, the *virtual prefetch* also prefetches several following virtual address pages based on the observation of strong spatial locality in virtual memory address space. The third one is the *aggressive virtual-address-space prefetching* (*Agg_virtual*) [6] which prefetch more pages with being aware of the device characteristics (e.g., parallelism and response time) by modifying modern huge page management. The other two methods are *A-SPF* and *A-SPF** which stand for the A-SPF handler without and with online memory-demand predictor, respectively. For all comparison methods, we use 4KB page size and the LRU-based DRAM page replacement policy.

TABLE I
Configuration of Traces

$Benchmarks$	$Mix3$	$Mix4$	$Mix5$	$Mix6$
$PageRank$	V	V	V	V
$RandomWalk$	V	V	V	V
$Blender$	V	V	V	V
Xz		V	V	V
Wrf			V	V
$Tensorflow$				V

Before conducting the experiments, we collect the memory traces by using the valgrind framework [28] which is a famous binary instrumentation tool. In our trace-driven simulator, the memory traces are given as input which will first go through the 16-way set associative 8-MB CPU cache. Filtered by the CPU cache, the remaining memory requests will then access the DRAM pages and may fetch the page from the ULL storage devices if a page fault occurs. The DRAM capacity is set to fit the working set size of the process. Note that the definition of the working set is that the size of the DRAM can absorb more than 99% of the CPU accesses. The latency of DRAM and ULL storage devices are configured by 50 ns [29] and 3 μs [5], respectively. Note that modern ULL devices, such as Intel Optane DC DIMM, usually manage write operations in an asynchronous way. Practically, devices usually have a write buffer [30] so as to avoid write operations influencing the device response time. To simplify the discussions, we assume the write response time is same as the read response time (i.e., 3 μs).

In this experiment, we use CFS as our process scheduler. Six traces collected from popular benchmarks are available for the scheduler to choose from. Specifically, the traces are Wrf from SPEC CPU® 2006 [23], Blender, Xz from SPEC CPU® 2017 [24], a deep learning framework (Tensorflow [25]), and two graph applications (PR and Random Walk algorithm on Graphchi [26]). Among the six processes, Page Rank and Random Walk are memory-intensive. To further evaluate the four methods, we select and mix the six traces to generate the multiprogramming workloads (i.e., Mix3 to Mix6), as shown in Table I. The CFS is modified from Linux Kernel 5.7 and with *sysctl_sched_min_granularity* and *sysctl_sched_latency* configured to 0.75 and 6 ms, respectively. The entire experiment is running on the machine equipped with Intel Core i7-7800X CPU and Linux Kernel 4.4.0. Every process is set with the default nice value. The overhead of performing context switch in the simulator is set to 5 μs [6], [8].

B. Experimental Results

1) Evaluation of Overall Performance: Our proposed A-SPF handler adjusts the CFS to create opportunities for reconstructing the working set. Also during the process execution,

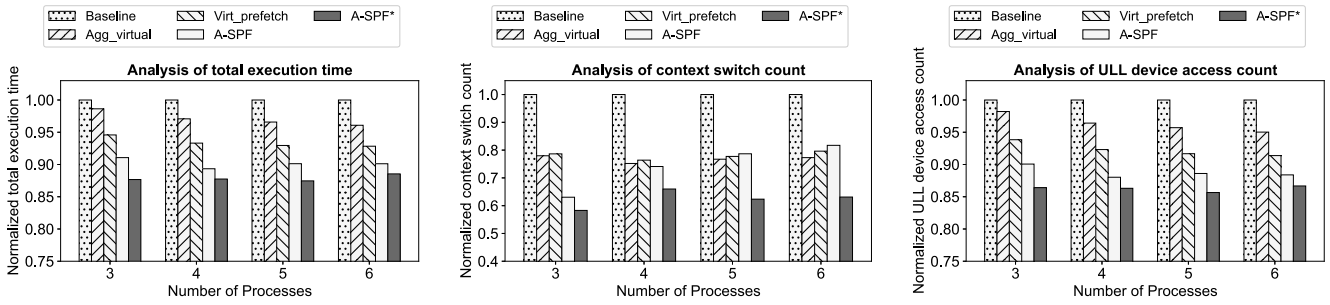

Fig. 6. Evaluation of overall performance. (a) Total execution time. (b) Context switch count. (c) ULL storage device access count.

we will collect the related pages so as to further reduce the future page faults. With the proposed design, the system can execute the processes more efficiently. As a result, we can boost the progress and reduce the process execution time. Fig. 6(a) shows the results of the total execution time, where the *y*-axis is the total execution time normalized to the *baseline* approach. For the three processes workload (*Mix*3), *A-SPF**, *A-SPF*, *Virt_prefetch*, and *Agg_virtual* can improve the total execution time by 12.33%, 8.93%, 5.42%, and 1.36% compared with the *baseline*. With consideration of asynchronous page prefetching design and the adaptive CPU scheduling design, the system can switch between the processes more smoothly. For the six processes workload (*Mix*6), *A-SPF**, *A-SPF*, *Virt_prefetch*, and *Agg_virtual* can improve the total execution time by 11.45%, 9.88%, 7.16%, and 3.92% compared with the *baseline*, respectively. The results clearly show that *A-SPF** outperforms among five methods for these workloads. With online memory-demand predictor, *A-SPF** can identify the memory-intensive processes accurately and, thus, adaptively adjust the scheduler to asynchronously reconstruct working sets. As a result, we can find that *A-SPF** is more powerful than *A-SPF* in total execution time. Note that *Agg_virtual* does not perform well as expected. This is because some processes such as Page Rank may not have strong spatial locality in wide range of the virtual-address space. As a result, some prefetching operations seem to be ineffective and even waste the memory space.

Fig. 6(b) provides the details about number of context switch for each approach, where the *y*-axis is the context switch count normalized to the *baseline* approach. For the three processes workload (*Mix*3), *A-SPF**, *A-SPF*, *Virt_prefetch*, and *Agg_virtual* can reduce the total number of context switches by 41.70%, 36.95%, 21.35%, and 22.03% compared with the *baseline*, respectively. We can conclude this result from two aspects. More effective approaches can help the system finish the entire execution in less time. As a result, the number of total context switch is decreased. The other reason is that the proposed *A-SPF** approach tends to give a longer CPU time to the memory-intensive processes when they are once executed and, thus, also decrease the number of context switches.

In the ULL storage-based memory extension system, we want every memory request can be served by the DRAM without any ULL access. However, due to the limited memory resources, it is impossible to keep everything in DRAM. As a result, it is critical to reduce the number of ULL access count so as to improve the system performance. *A-SPF**, *A-SPF*, *Virt_prefetch*, and *Agg_virtual* all take this into account and adopt different strategies. Fig. 6(c) provides the details about number of ULL storage device access account for each approach, where the *y*-axis is the access count normalized to the *baseline* approach. For the six processes workload (*Mix*6), *A-SPF**, *A-SPF*, *Virt_prefetch*, and *Agg_virtual* can reduce the total number of ULL accesses by 13.33%, 11.61%, 8.62%, and 4.98% compared with the *baseline*, respectively. With the asynchronous page prefetching design, *A-SPF** can prevent the future page faults in advance. Also, when the access behavior is weak in virtual memory locality, the immediate page prefetching in *A-SPF** performs better than the *Virt_prefetch* by collecting the related page set.

In the fault-aware prefetch set design, we collect the related page set together so as to utilize the relationship information to further alleviate future page faults. In this experiment, we also tag the pages that are prefetched by the fault-aware prefetch design and analyze the efficacy of the page grouping policy. Once a page with a prefetch tag is swapped back to the ULL device, we will examine the PG_ACCESSED bit. If it has not been accessed since it was prefetched, we would increment the fail counter of the algorithm. Otherwise, we will count it as successful set prefetch operation. The results show that 54.54% prefetch operations of the page grouping policy are effective under the scenario of running six traces (i.e., *Mix*6).

Note that we configure DRAM capacity to hold the minimum working set size among all processes. Specifically, DRAM is set to 200 MB to fit the working set of the Tensorflow process. As a result, we can know that even under the resource-constrained condition, the proposed design can effectively improve the overall system performance. With the help of the Online Memory-Demand Predictor, the system can identify the processes with stronger memory-demand and then dynamically adjust the system resources.

2) Evaluation of Performance on Different ULL Response Time: Here, we will explore the impact of different ULL storage device response time. Six processes workload (*Mix*6) is used. We evaluate the four approaches with different ULL response time range from 3 μs to 7 μs. The result is shown in Fig. 7, where the *y*-axis is the total execution time normalized to the *baseline* approach and *x*-axis is the ULL device response

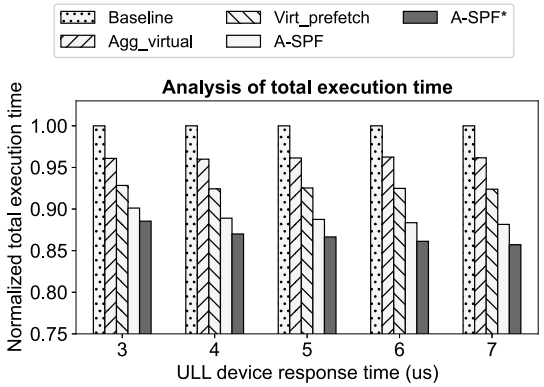

Fig. 7. Evaluation of performance on different ULL response times (us).

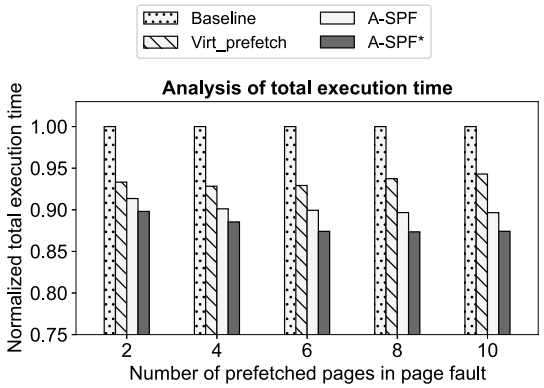

Fig. 8. Evaluation of performance on different numbers of prefetched pages.

time in μs. For the ULL response time ranging from 7 μs to 3 μs, *A-SPF** improves the total execution time 14.29%, 13.88%, 13.35%, 12.99%, 11.45% compared with *baseline* approach, respectively. We can see that as the ULL device is faster, the improvement will become lesser. We can observe the same tendency for the *Virt_prefetch* with 7.61%, 7.52%, 7.47%, 7.57%, 7.16% improvement over *baseline* approach for 7 μs to 3 μs ULL response time. The reason of this tendency is that the time saved by page prefetching decreases gradually when the access latency gap between memory and storage devices keeps closing.

3) Evaluation of Performance on Different Number of Prefetched Pages: Data prefetching can reduce the number of page faults, but how can we determine the number of prefetched page in the page fault handler. We conduct the experiments for different number of prefetched page. The result is shown in Fig. 8, where the *y*-axis is the total execution time normalized to the *baseline* approach and *x*-axis is the number of prefetched pages in the page fault handler. When the prefetched page is increased from 2 to 10, the *Virt_prefetch* improves the total execution time 6.68%, 7.16%, 7.08%, 6.26%, 5.70% compared with *baseline* approach, respectively. It is interesting that the *Virt_prefetch* does not gain the performance improvement when the prefetched count is increased from 4 to 10. The main reason is when we prefetch lots of pages, same DRAM pages may be replaced and, thus, cancel out the benefits of the prefetching. On the other hand, the *A-SPF** has 10.19%, 11.45%, 12.59%, 12.65%, 12.58% improvement for 2 to 10 prefetched page. The number of prefetched page in *A-SPF** depends on the accuracy of the fault-aware prefetch set and the previous history. So it will not prefetch as many pages as *Virt_prefech* and, thus, can still have competitive performance improvement. Generally, the number of prefetched pages is highly proportional to the energy consumption, where moving each page consumes energy. Thus, we also measure the energy consumption and show that the *A-SPF** consume lower energy than *Virt_prefech*. Specifically, based on our measurement, *A-SPF** consumes 47.6% less power than the *Virt_prefetch* for the default configuration.

4) Evaluation of Fairness: The proposed A-SPF handler adjusts the CFS so as to create opportunities for reconstructing the working set. However, the adjustment of the scheduler in the adaptive CPU scheduling design may affect the fairness [31] between processes, CPU resources may not be evenly distributed anymore. Considering this issue, we try to even the distribution of CPU time among all the processes (described in the end of Section III-B2) by compensating the memory-intensive processes. In this section, we will analyze the distribution of CPU resources. Two approaches will be evaluated together: 1) *baseline* and 2) *A-SPF**. Note that without any modification to the CFS, the *baseline* can achieve the best fairness. We evaluate the fairness by running with two memory-intensive (PR stands for Page Rank and RW stands for Random Walk) and one nonmemory intensive (Wrf) traces in the system and observe the distribution of CPU time. The asymmetric number between memory-intensive and nonmemory intensive traces can enlarge the unfairness between each process. We can see the evaluation results on CPU time distribution in Fig. 9, where Fig. 9(a) and (b) shows the CPU time distribution over 5% and 10% execution time, respectively. The *x*-axis is three processes and the *y*-axis indicates the percentage of CPU time occupied by the corresponding process. For 5% execution time [Fig. 9(a)], PR, RW, and Wrf will gain 33.45%, 33.13%, and 33.42% CPU time in *baseline* approach, which is very close to 33.33%. With using *A-SPF**, the ratio of CPU time consumed by three traces (i.e., PR, RW, and Wrf) is 33.40%, 32.06%, and 34.54%, respectively. It is obvious that the ratio of CPU time consumed by the three traces is close to each others, which means the CPU resource is fairly distributed to each trace. Besides, the CPU resource distribution under running *A-SPF** is very close to that of running the *baseline* (or the CFS), which is designed to fairly distribute CPU resources. As the system executes for longer time, the *A-SPF** can distribute the CPU resources more evenly among all processes. As shown in Fig. 9(b) where systems run for around 10% of execution time, with using *A-SPF**, the ratio of CPU time consumed by three traces is 32.73%, 33.39%, and 33.88%, respectively. The CPU resource distribution among three traces becomes closer to 33.33%. That is, running the system with *A-SPF** for 10% of execution time shows a better fairness result than that of running only for 5% of execution time.

C. Analysis of Management Overhead

1) Analysis of Computation Overhead: The proposed A-SPF handler comprises three main designs: 1) the immediate

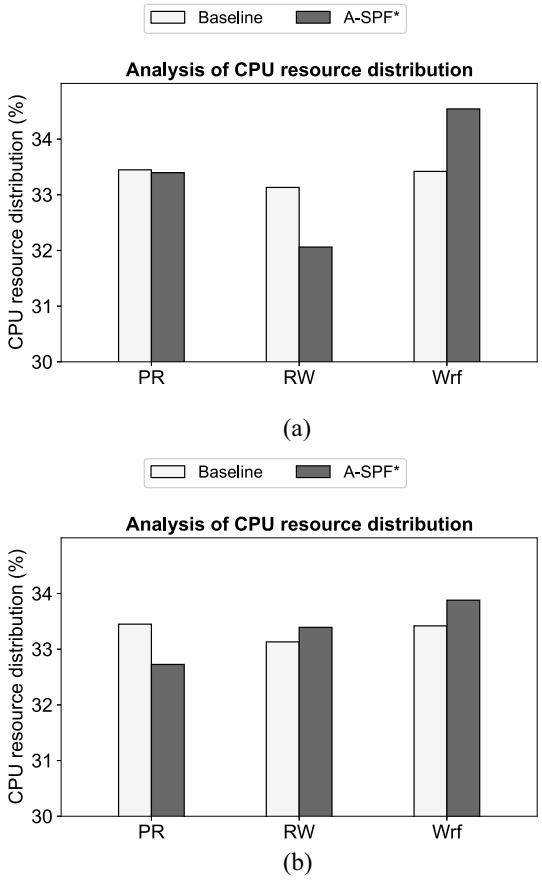

Fig. 9. Evaluation on CPU time distribution. (a) CPU time distribution of 5% execution time. (b) CPU time distribution of 10% execution time.

page prefetching design; 2) the adaptive CPU scheduling design; and 3) the asynchronous page prefetching design. Among these designs, we use the linked list implementation to flexibly store the required information. For the immediate page prefetching design, the OS will find the corresponding *struct page* and locate the embedded *set_node* in the *struct page* which can all be finished in constant time. After that, we will prefetch from the head of the linked list which also incurs extra constant time overhead to find the *set_node* and the *struct page*. In the fault-aware page grouping policy, the most complicated logic is to assign the correct *set_id* which can be implemented within a constant time overhead by only a few branch instructions. In the adaptive CPU scheduling design, we maintain the postpone queue to delay the scheduling of the memory-intensive process, the virtual runtime calculation can also be done within constant time. The most time consuming part is the red–black tree insertion which will require logarithmic time complexity. In the asynchronous page prefetching design, we will reconstruct the page from the head of the reconstruct list which is also a constant time operation. The dominated operation is to identify whether the next process is memory intensive. Thanks to the delicate design of the red–black tree which caches the left most node of the tree. As a result, we can identify the property of the next process in constant time too.

2) Analysis of Storage Overhead: We have introduced several data structures to maintain the related information. In the immediate page prefetching, each *set_struct* consists of a *set_id* (uint16_t), a *gen_id* (uint16_t), a *prefetch_count* (uint32_t), a *prefetch_miss* (uint32_t), and a pointer of the doubly linked list (8 bytes). In short, each *set_struct* requires 20-bytes storage overhead. We also use *set_node* to record the page information and it consists of a *set_id* (uint16_t), a *gen_id* (uint16_t), and a *list_entry* (8 bytes). Thus, each *set_node* is 12 bytes. In order to enable the adaptive CPU scheduling design, every process will need extra information including a *delay_time* (uint64_t), an *is_intense* (1 bit), a *prev_timeslice* (uint64_t), and a *prev_fault* (uint64_t). Among these fields, *prev_timeslice* is the previous timeslice period of the process, and *prev_fault* represents the number of page faults occurred during previous process timeslice period. In summary, there will be 25-bytes storage overhead for every process. Apart from that, there is also a postpone queue which is a dynamic array. Every element in the queue is just a pointer (uint32_t). For the asynchronous page prefetching design, it relies on a doubly linked list to track the information. Every node in the reconstruct and inactive list represents the corresponding page and contains extra fields such as a *list_entry* (8 bytes).

V. Conclusion

The advance of NVM technology has introduced ULL storage devices and the response time of ULL storage devices is now faster than conducting the context switch. Researchers from Intel and IBM suggest to redefine the ways in handling the main memory and storage by switching the page fault handler from asynchronous to synchronous. However, by synchronously handling page faults, the process may suffer from the working set contention issue. That is, processes might occupy more memory space than before and aggressively kick out other processes' working sets. To deal with the working set contention issue, this work proposes an A-SPF handler. Results showed that our strategy can save 12.33% of the total execution time and reduce 13.33% of page faults, compared to the conventional demand paging strategy with nearly no sacrificing of process fairness.

References

[1] H.-Y. Cheng *et al.*, "Future computing platform design: A cross-layer design approach," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2021, pp. 312–317.

[2] "Produciton brief: Intel optane SSD DC P4800X/P4801X series." Intel.com. 2018. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-dc-p4800x-p4801x-brief.html

[3] W. Cheong *et al.*, "A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D nand flash with 3μs read time," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 338–340.

[4] C.-F. Wu, M.-C. Yang, Y.-H. Chang, and T.-W. Kuo, "Hot-spot suppression for resource-constrained image recognition devices with nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2567–2577, Nov. 2018.

[5] "Technology brief: Ultra-low latency with samsung Z-NAND SSD." Samsung.com. 2017. [Online]. Available: https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf

[6] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "When storage response time catches up with overall context switch overhead, what is next?" *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4266–4277, Nov. 2020.

[7] D. Waddington and J. Harris, "Software challenges for the changing storage landscape," *Commun. ACM*, vol. 61, no. 11, pp. 136–145, 2018.

[8] V. M. Weaver, "Linux perf_event features and overhead," in *Proc. 2nd Int. Workshop Perform. Anal. Workload Optimized Syst.*, vol. 13, 2013, p. 5.

[9] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proc. FAST*, vol. 12, 2012, p. 3.

[10] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "MemPod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 433–444.

[11] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, 2003.

[12] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2010, pp. 1–14.

[13] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *Proc. 8th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2011, pp. 1–14.

[14] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "Joint management of CPU and NVDIMM for breaking down the great memory wall," *IEEE Trans. Comput.*, vol. 69, no. 5, pp. 722–733, May 2020.

[15] P. J. Denning, "Virtual memory," *ACM Comput. Surveys*, vol. 2, no. 3, pp. 153–189, 1970.

[16] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. Beijing, China: O'Reilly Media, 2005.

[17] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of {ultra-low} latency solid state drives," in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2018.

[18] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 603–616.

[19] J. Zhang *et al.*, "FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 477–492.

[20] C. S. Pabla, "Completely fair scheduler," *Linux J.*, no. 184, p. 4, 2009.

[21] T.-S. Lo, C.-F. Wu, Y.-H. Chang, T.-W. Kuo, and W.-C. Wang, "Space-efficient graph data placement to save energy of ReRAM crossbar," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, 2021, pp. 1–6.

[22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, *Introducing the Graph 500*, Cray Users Group (CUG), vol. 19, 2010, pp. 45–74.

[23] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[24] "SPEC CPU 2017." 2018. [Online]. Available: https://www.spec.org/cpu2017/

[25] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement.*, 2016, pp. 265–283.

[26] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Oper. Syst. Design Implement.*, 2012, pp. 31–46.

[27] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall, 2004.

[28] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[29] R. Chen, Z. Shao, and T. Li, "Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2016, pp. 1–12.

[30] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: Os design for heterogeneous memory management in datacenter," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 521–534.

[31] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun, "Fairness and interactive performance of O (1) and CFS Linux kernel schedulers," in *Proc. Int. Symp. Inf. Technol.*, vol. 4, 2008, pp. 1–8.

Yin-Chiuan Chen received the B.S. degree from the Department of Electronics and Electrical Engineering, National Yang Ming Chiao Tung University, Taipei, Taiwan, in 2019, and the M.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, in 2021.

He is currently a Software Engineer with Google Taiwan, Taipei. His work focuses on the display driver in the mobile systems. His primary research interests include memory/storage systems and operating systems.

Chun-Feng Wu (Member, IEEE) received the M.S. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2016, and the Ph.D. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2021.

He is currently an Assistant Professor with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu. Previously, he was a Postdoctoral Scholar with the Department of Computer Science, Harvard University, Cambridge, MA, USA, from 2021 to 2022. He served for the Research and Development Alternative Service with the Institute of Information Science, Academia Sinica, Taipei, from 2017 to 2021. His primary research interests include memory/storage systems, embedded systems, operating systems, and next-generation memory/storage architecture designs.

Yuan-Hao Chang (Senior Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2009.

He is currently a Research Fellow with the Institute of Information Science, Academia Sinica, Taipei, where he served as an Assistant Research Fellow from August 2011 to March 2015 and an Associate Research Fellow from March 2015 to June 2018. His research interests include memory/storage systems, operating systems, embedded systems, and real-time systems.

Dr. Chang is a Senior Member of ACM.

Tei-Wei Kuo (Fellow, IEEE) received the B.S.E. degree in computer science from the National Taiwan University, Taipei, Taiwan, in 1986, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 1994.

He is the Dean of the College of Engineering, City University of Hong Kong, Hong Kong, and a Distinguished Professor with the Department of Computer Science and Information Engineering, National Taiwan University, where he served as the Department Chairman from August 2005 to July 2008. He served as a Distinguished Research Fellow and the Director of the Research Center for Information Technology Innovation, Academia Sinica, Taipei, from 20 January 2015 to 31 July 2016. His expertise is embedded systems, nonvolatile memory system and software, and real-time systems.

Dr. Kuo is a Fellow of ACM.