How to Steal CPU Idle Time When Synchronous I/O Mode Becomes Promising

Chun-Feng Wu[1], Yuan-Hao Chang[2], Ming-Chang Yang[3], and Tei-Wei Kuo[4,5,6]
[1]Department of Computer Science, National Yang Ming Chiao Tung University
[2]Institue of Information Science, Academia Sinica
[3]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[4]Department of Computer Science and Information Engineering, National Taiwan University
[5]High Performance and Scientific Computing Center, National Taiwan University
[6]Delta Electronics
cfwu417@cs.nycu.edu.tw,johnson@iis.sinica.edu.tw,mcyang@cse.cuhk.edu.hk,ktw@csie.ntu.edu.tw

Abstract

The advent of Ultra-Low-Latency storage devices has narrowed the performance gap between storage and CPU in computing platforms, facilitating synchronous I/O adoption. Yet, this approach introduces substantial busy waiting time and underutilizes computing units. To address this, we propose a light-weighted Idle-Time-Stealing (ITS) design. This involves a self-improving thread conducting prefetching for high-priority processes during synchronous I/O, and an I/O-waiting process continuing subsequent instruction executions when justifiable. Another thread, the self-sacrificing thread, proactively switches low-priority process I/O requests from synchronous to asynchronous mode, prioritizing high-priority executions. Experimental results demonstrate the effectiveness of our ITS design in reducing CPU idle time.

ACM Reference Format:
Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, Tei-Wei Kuo. 2024. How to Steal CPU Idle Time When Synchronous I/O Mode Becomes Promising. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655929

1 Introduction

With the rise of new storage media like flash memory or 3D-XPoint, the performance disparity between storage devices and CPUs on many platforms is rapidly diminishing [17, 19]. Researchers are questioning the continued appropriateness of exclusively utilizing asynchronous I/Os, as the context-switch overheads may no longer outweigh the benefits [4, 15, 18]. Since no productive work takes place during CPU busy waiting in synchronous I/Os, it is essential to reevaluate how we utilize this waiting time to optimize the efficiency of process execution in our systems. These considerations hold particular significance for heavily loaded data-intensive applications, such as graphs, high-performance computing tasks, and Large Language Models [6].

Generally, CPUs synchronously access fast I/O devices (e.g., DRAM) without the intervention of the Operating System (OS) so as to avoid software overhead. On the other hand, CPUs asynchronously access slow I/O devices (e.g., Solid-State-Drives (SSDs)) with more helps from the OS (e.g., context switching, file system, and page cache) for the optimizations depending on different I/O paths. When the CPU accesses the data located in the slow I/O devices, the OS is invoked for processing one of the two I/O paths: process I/O or file I/O[1]. This work focuses on the "process I/O (also called swap I/O)", caused when the CPU loads/stores data (e.g., Heap data) in the swap area. To handle process I/Os, the OS marks the Direct Memory Access (DMA) to move the data to the swap cache in the DRAM, and then performs a context switch to avoid the CPU waiting for the slow I/O.

In recent years, research from Google [1] and industry leaders such as Intel and IBM [15, 20] reveals that the storage response time of certain Ultra-Low-Latency (ULL) I/O devices, like Intel® Optane™ SSDs and Samsung Z-NAND SSDs [9, 21], has reached microseconds-levels (μs), often outpacing the overhead of context switches that can exceed 5-10μs in general-purpose systems [15, 16, 20]. Faced with this shift, asynchronous I/O mode proves ineffective for handling I/O requests with response times faster than a context switch. Instead, Intel Corporation and IBM Corporation [15, 20] advocate synchronous I/O mode, involving CPU busy waiting for swift I/O completion. While this speeds up overall performance, it results in wasted computation resources that cannot contribute to execution progress during CPU busy waiting. This resource inefficiency becomes more pronounced, particularly when dealing with larger I/O sizes like huge page management [7, 13, 18]. The impact is heightened for high-priority processes with longer time slices from the current process scheduler. Hence, optimizing the use of busy waiting time is crucial, especially for high-priority processes.

Distinct from past works, we proposed a light-weighted Idle-Time-Stealing (ITS) design to utilize the otherwise-wasted I/O busy time. We favor the executions of high-priority processes (called self-improving processes for the rest of this paper) by executing their I/O requests in synchronous mode to shorten their execution time. We propose to use a kernel thread (called a self-improving thread) for each self-improving process to initiate needed memory prefetching actions by utilizing the synchronous I/O waiting time. Suppose there is any time left during the synchronous I/O waiting time. In that case, the self-improving thread might even pre-execute subsequent instructions following the I/O request under our proposed pre-execute policy to proactively shorten the process execution time, where the pre-execute policy must justify the trade-off in pre-execution. We propose to execute the I/O requests of low-priority processes (called self-sacrificing processes) in the asynchronous mode under our proposed priority-aware I/O mode selection policy so as to give way to high-priority process executions. Even though low-priority processes seem to be disfavored in executions over CPU and I/O devices, they might receive

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655929

[1]Each file I/O is triggered when the CPU runs read/write system calls, and it involves filesystem and page cache managements.

much more concentrated attention when they can execute (after high-priority processes rapidly finish their executions) and might even have an earlier finish time in many cases.

The evaluation results show that our ITS design could effectively reduce the CPU idle time by around 61%∼66% and 17%∼43%, than those with the asynchronous I/O mode and the synchronous I/O mode, respectively. Meanwhile, the average finish time of high-priority processes can be reduced by 65%∼75% and 11%∼33% than those with the asynchronous I/O mode and the synchronous I/O mode, respectively.

2 Background, Observation, and Objective

2.1 Background

2.1.1 Context Switch. To provide the illusion of large memory space at a lower price, virtual memory management enables extending memory space with storage devices. Due to the huge latency gap between memory and storage devices, the CPU will stall significantly while accessing storage devices. Long CPU stalling leads to low CPU utilization and further degrades system performance. To hide long CPU stalling, the OS triggers a context switch [14] to yield the idle CPU to other processes and let the I/O request be served asynchronously without blocking CPU resources. The design objective of the context switch is to back up process contexts before switching the process, which is waiting for the I/O completion and then restoring the other ready-to-run process. Specifically, process contexts[2] stored in the CPU registers will be copied to the main memory, and the process contexts related to the switched-in process will be moved from the main memory to the CPU registers. Due to the movements of process context between main memory and CPU registers, the time spent on a conducting context switch is around several microseconds, e.g., longer than 5–10 μs on a general-purpose machine as reported by several researchers, such as from Intel and IBM [15, 16, 18, 20]. Frequently performing context switching may cause frequent CPU cache misses and Translation Look-aside Buffer (TLB) shootdown.

2.1.2 Killer Microsecond. The context switching overhead becomes the performance bottleneck when the CPU accesses the ULL devices [9], whose access latency is around a few microseconds. Even worse, this microsecond-level CPU stalling cannot be effectively hidden by using CPU optimizations. Specifically, researchers from Google [1] point out that simultaneous multi-threading (e.g., Intel Hyper-Threading) and other techniques for nanosecond time scales (e.g., superscalar out-of-order execution and branch prediction) do not scale well to deal with microsecond requests. The reason is that designers lack enough hardware-managed thread contexts or instruction-level parallelism to hide the microsecond-level CPU stalling. Due to the difficulty in tackling the microsecond-level CPU stalling, it is also called the "killer microsecond" [1, 15]. To deal with the killer microsecond, the most recent solution is to wait directly for the fast access latency provided by the ULL devices without performing the context switch. *As a result, instead of adopting the asynchronous I/O mode, Intel and IBM advocate to adopt the synchronous I/O mode by forcing the CPU busy waiting during accessing ULL devices [15, 20].*

2.2 Observation and Objective

Although adopting the synchronous I/O mode to wait for the I/O completion of the ULL devices can eliminate the performance bottleneck caused by conducting a context switch, the time CPU spent on busy waiting (i.e., CPU idle time) cannot be utilized to proceed with process progress. We conduct an experiment to show the CPU idle time while running multiple processes. We select five representative processes from several popular benchmark suites, that is, Wrf, Blender, page rank, random walk algorithm, and also the single shortest path algorithm. More details for the datasets and simulation configuration can be found in Section 4.1. According to our evaluation results, more than 22% of CPU idle time is spent waiting for the completion of the synchronous I/O, where all results are normalized to the time spent on running 2 processes. Please note that the CPU idle time is the aggregated time of the CPU busy waiting for the response of memory and storage devices during the cache misses and page faults, respectively. Moreover, the long idle time issue becomes more serious when more processes are run simultaneously. The reason is that all processes share and contend the memory resources; thus, this phenomenon will frequently cause a page fault which makes the CPU become idle. *With this observation in mind, this work aims to propose a new I/O design to steal and utilize the otherwise-wasted CPU idle time so as to further improve the system performance.*

3 Idle-Time-Stealing (ITS) Design

3.1 Design Overview & Concept

In Figure 1, the interaction among our Idle-Time-Stealing (ITS) design, OS modules (e.g., page fault handler), and hardware components is depicted. The Memory Management Unit (MMU) plays a key role in translating virtual memory addresses to corresponding physical addresses and checking page status through page table entry lookups. When a user process lacks permission to access a page or the required page is absent from memory, the MMU triggers an exception known as a page fault (❶). Subsequently, the CPU transitions from user mode to kernel mode to execute the page fault handler (❷). The page fault handler examines page table information, such as page table entries, linked to the fault address and performs fault area checking to determine whether it is a minor or major fault[3] (❸). In the case of a major page fault, the handler invokes the page swapping function to instruct the DMA controller to transfer the required page's physical address from ULL devices to DRAM (❹). It is noteworthy that this work concentrates solely on addressing major page faults due to their more substantial impact on execution time.

Our ITS design (colored in Figure 1) initiates new kernel threads (called ITS threads) to run their corresponding kernel functions by utilizing the otherwise-wasted CPU resources. To avoid reinventing wheels, we reuse some functions provided by the Linux kernel to realize our ITS design. In Section 3.2, our priority-aware thread selection policy first checks the priority information of the user process by querying the OS scheduler. Then, it initiates the self-sacrificing kernel thread if the user process is a low-priority process. Otherwise, the policy initiates the self-improving kernel thread (❺). The design concept of the self-sacrificing kernel thread in Section 3.3 is to execute any I/O request issued by a low-priority process in the asynchronous mode so as to give way to high-priority process executions.

On the other hand, during the synchronous I/O waiting time of a high-priority process, its self-improving kernel thread (introduced in Section 3.4) will run the page-prefetch policy in Section 3.4.1 to initiate memory prefetching actions over DMA by considering access locality explored from the page table so as to shorten its future execution time. Suppose there is any time left during the synchronous I/O waiting time. In that case, the self-improving kernel thread might even pre-execute subsequent instructions following the I/O

[2]Process contexts are the process states stored in the CPU registers including stack pointer, instruction pointer, kernel stack, page table directory, program counter, and so on.

[3]Major page faults involve data movement between memory and storage devices, while minor page faults only require metadata adjustments.

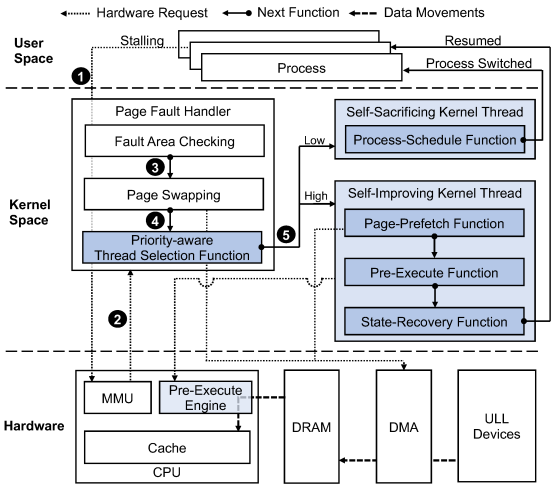


Figure 1. System Architecture of the ITS Design.

request under our pre-execute policy proposed in Section 3.4.2 as much as possible. The real effects of the pre-execute policy are to populate the cache so that high-priority processes have better chances to finish earlier. For ensuing computation correctness, the pre-execute policy should not execute instructions related to the result of the executing I/O instruction and could write to any variable. In Section 3.4.3, a state-recovery policy is presented to restore the CPU state of a high-priority process to its original state, which might be updated by the execution of its self-improving kernel thread.

3.2 Priority-aware Thread Selection Policy

The priority-aware thread selection policy selects to execute the self-sacrificing kernel thread to yield the computation resources occupied by a low-priority process or to run the self-improving kernel thread to boost the progress of a high-priority process. Noting that, our policy does not change the priority of each process and the process-execution orders maintained by the process scheduler. In this work, we compare the priority value of the current running process against the next-to-be-run process to decide whether the current running process is a low-priority or a high-priority process. For example, the current running process will be identified as a low-priority process if its priority value is lower than the next-to-be-run process and vice versa.

Our ITS design activates exclusively during CPU idle time, running for a maximum of several microseconds to avoid impeding process progress post-I/O completion. With a focus on maximizing underutilized CPU resources, a primary objective is to minimize the overhead in transitioning from the page fault handler to our ITS designs. Consequently, all proposed software designs in this work operate within the OS (i.e., in kernel space). This choice is made because switching to kernel-level designs takes only hundreds of nanoseconds, whereas transitioning to user-level designs demands several microseconds.

The time advantage arises from two factors [2]: CPU mode switching costs and context movement overhead. Firstly, when executing the page fault handler or kernel-level designs, the CPU remains in kernel mode, necessitating a mode switch to user mode for user-level designs. Secondly, in Linux's architecture, kernel functions (e.g., the page fault handler and kernel-level designs) share similar process contexts, leading to minimal overhead in moving required contexts from DRAM to the CPU cache. Conversely, switching to other user-level designs mandates moving entire process contexts since each user process possesses its own context, encompassing the user stack and addressing space.

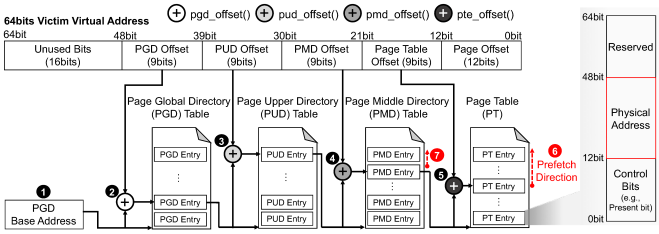


Figure 2. Design Concept of Page-Prefetch Policy.

3.3 Self-Sacrificing Kernel Thread

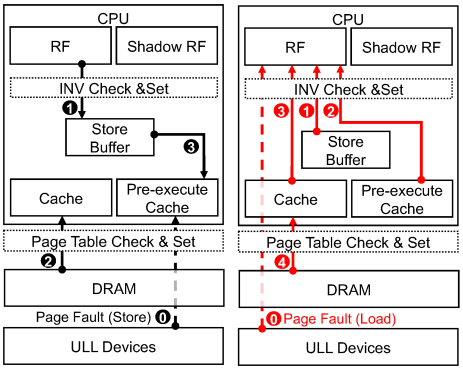
When a low-priority process is waiting for the I/O completion, the self-sacrificing kernel thread forces this process to give up its CPU resources even when it still has sufficient time slices. Specifically, all high-priority processes only yield their CPU resources (i.e., execute context switch) when their allocated time slices are exhausted. Still, all low-priority processes are forced to switch their CPU resources to other processes once they are waiting for I/O completion. Although conducting process switching consumes more time than busy waiting for the I/O completion, this earlier resource release can benefit the high-priority processes' progress in several aspects. First, switching low-priority processes earlier can probably prevent the low-priority process from evicting the data pages belonging to the high-priority process, especially when the low-priority process is a data-intensive application that leads to frequent page replacements. Besides, CPU resources can be allocated to high-priority processes more frequently so as to finish their progress earlier. Noting that, low-priority processes seem to sacrifice their progress to favor the execution of high-priority processes. Still, their finish time will not be increased because low-priority processes can receive more dedicated resources after the completion of high-priority processes. The experiment results evaluating the process finish time of low-priority processes are shown in Section 4.2.2.

3.4 Self-Improving Kernel Thread

High-priority processes retain resources until their allocated time slice ends, unlike low-priority ones. Self-improving kernel threads execute both page-prefetch and pre-execute policies to optimize hardware resources for these high-priority processes.

3.4.1 Page-Prefetch Policy.

The goal of the page-prefetch policy is to conceal the time taken to transfer data pages between storage and memory by preemptively moving a group of anticipated pages, expected to be accessed soon. Leveraging the substantial parallelism offered by SSDs, this policy initiates a prefetcher to forecast the likely accessed pages. It then dispatches the physical addresses of these pages to the DMA for relocation. Employing DMA for this task bypasses utilizing CPU resources, allowing the CPU to focus on other optimization tasks like the pre-execute policy mentioned in Section 3.4.2. Our virtual-address-based data prefetcher walks the page table to simultaneously translate the virtual address to physical address and to find multiple candidate to-be-prefetched pages, which are located right after the victim page on the virtual addressing space. The victim page is the page that causes the page fault. After finding those pages, our policy looks up the physical



(a) Store Instructions. **(b)** Load Instructions.

Figure 3. Design Concept of Fault-aware Pre-execute Policy

address for each candidate page and sends these physical addresses to the DMA for data moving.

Figure 2 illustrates the functionality of a virtual-address-based page prefetcher operating on a 64-bit virtual address to find n candidate pages. In a 64-bit x86_64 Linux system using 4-level page tables, the prefetcher initially accesses the base address of the Page Global Directory (PGD) table via the pgd base pointer within the memory descriptor structure (*mm_struct* from the Linux kernel) (❶). Using the *pgd_offset()* function, a synthetic address is generated by adding PGD offsets to the base PGD table address, facilitating traversal to access the corresponding PGD entry (❷). This entry yields the base address of the Page Upper Directory (PUD) table. Subsequently, the prefetcher navigates the PUD, Page Middle Directory (PMD), and Page Table utilizing functions like *pud_offset()*, *pmd_offset()*, and *pte_offset()* (❸❹❺).

While traversing the page table, our policy iteratively increments the page table offset and utilizes the *pte_offset()* function to retrieve the candidate page following the victim page in the virtual addressing space (❻). To prevent prefetching pages already present in DRAM, the policy checks the present bit stored in the PT entry. If the candidate page remains in storage, the policy retrieves its physical address located between bit positions 12 and 48 in the PT entry. In cases where an insufficient number of candidate pages is gathered after walking through the entire page table, the policy reverts to traversing the next PMD entry in the PMD table to access an alternative page table (❼).

3.4.2 Fault-aware Pre-execute Policy. The goal of fault-aware pre-execution is to diminish CPU cache misses by pre-executing upcoming instructions and caching data in the CPU cache during I/O-related delays due to page fault handling. This strategy significantly curtails cache misses as its effectiveness scales with its execution duration, typically within several microseconds, aligning with the time required to manage page faults.

A key challenge in pre-execution is handling instructions following a page fault-causing instruction. Invalid data in these subsequent instructions can stall pre-execution. Invalid or bogus data refers to data that hasn't been recently accessed or is unknown. For instance, the initial invalid data is what triggers the page fault. Instead of halting pre-execution, our policy flags all instructions interacting with invalid data as invalid. It skips executing these invalid instructions and proceeds to execute the next valid one. Additionally, to prevent execution on bogus data, our policy cascades the invalid mark to all registers associated with an instruction if any of its source registers are marked as invalid.

To implement this design, we expand the Register File (RF) by adding additional "INV" bits for each register [5, 11]. Within each CPU, we introduce a pre-execute cache, associating an "INV" bit with each byte [11]. This cache stores both data values and their associated INV statuses linked to retired store instructions from the store buffer. Consequently, pre-execute load instructions dependent on these retired store instructions can be verified by checking the pre-execute cache. Notably, the pre-execute cache access is solely permissible during pre-execution for load and store operations. Additionally, we propose integrating an INV bit in each page table entry. In Linux, several spare bits in the control-bit area of each page table entry can be repurposed for the INV bit. This design allows judgment of the pre-executed instruction's data status if it resides in the main memory or cache but not in the pre-execute cache. This judgment occurs post-TLB access or CPU traversal through the page table. Figure 3 illustrates the operation flow when the pre-execute execution runs store and load operations.

The pre-execute store operation (refer to Figure 3a) becomes invalid if its associated data resides in the storage device. In such cases, our policy allocates a cache line in the pre-execute cache and sets the INV bit for the written bytes (❽). However, if the data is in the DRAM or CPU cache, the pre-execute store operation remains valid and writes the result into the store buffer. Simultaneously, our policy manages the associated INV bit, setting or resetting it to an invalid or valid state based on the data's status (❶). When the corresponding data is in memory but not in the cache, a data fetch query is sent to move it from memory (or DRAM in this context) to the cache (❷). Additionally, if the pre-execute store operation is invalid, the INV bit in the page table entry corresponding to the data is set. Upon writing the result of a pre-execute store operation to the store buffer and needing to transfer it to the cache line in the pre-execute cache, the INV bit for all written bytes in the pre-execute cache is updated based on the operation's status (❸). Importantly, pre-execute store operations do not write or modify any data in the CPU cache or memory.

On the other hand, the pre-execute load operation (refer to Figure 3b) is invalid under several conditions. Firstly, when the required data resides in the storage device (❽) or depends on the result from an invalid pre-execute store operation in the store buffer or pre-execute cache (❶❷). Moreover, if the data isn't in the store buffer or pre-execute cache but exists in the CPU's main cache, the CPU then checks the INV bit in the corresponding page table entry to determine the operation's status (❸). Ultimately, if the data is only available in memory, the pre-execute load operation becomes valid, and the data is moved to the CPU cache (❹).

3.4.3 State-Recovery Policy. Utilizing CPU resources with our ITS design modifies the architectural register file state, potentially affecting computation accuracy upon the CPU's return to regular processing. To ensure correctness, we adopt a state-recovery policy. This policy checkpoints the register file state, including the program counter and stack pointer, to a shadow register file (Figure 3) upon ITS activation, restoring this information before ITS termination. To sustain post-ITS process performance, critical registers such as the branch history register and return address stack are checkpointed to avoid reconstruction after ITS completion. The state-recovery policy is triggered by either polling, where a timer periodically checks I/O completion, or interruption, initiated by DMA upon I/O completion in the self-improving kernel thread.

4 Performance Evaluation

4.1 Performance Metrics and Evaluation Setup

We evaluate the effectiveness of the proposed ITS design in reducing the CPU idle time and the process finish time. In particular,

we evaluate both metrics by comparing the proposed idle-time-stealing design (denoted as "ITS") with four baseline approaches, i.e., the traditional asynchronous I/O management method (denoted as "Async"), the synchronous I/O method advocated by Intel and IBM (denoted as "Sync") [15] the synchronous I/O method with only running the CPU runahead execution[4] [5, 10, 11] (denoted as "Sync_Runahead"), which populates upper-level (e.g., L1 and L2) caches by doing pre-execution during handling the last level cache misses, and the synchronous I/O method with the page-based prefetching policy[5] [17] (denoted as "Sync_Prefetch").

To evaluate the effectiveness of our design, we implemented an in-house trace-based simulator, including a simulated CPU design and a mini Linux-based kernel. The front end of our trace-based simulator adopts the dynamic binary instruction tools, Valgrind [12], to capture the accessed virtual addresses generated by each benchmark or application. Our simulated CPU simulates a 16-way set associative 8MB Last-Level Cache (LLC), where a half size of the LLC will be configured as the pre-execute cache for both Sync_Runahead and ITS. We also integrate the pre-execute engine with the simulated CPU. Our mini Linux-based kernel is crafted based on the Linux kernel version 4.4. We integrate the page table management with our virtual-address-based page prefetcher. To support multi-programming, our mini kernel includes the Linux real-time round-robin process scheduler (i.e., SCHED_RR). The priority of each process is assigned randomly, and the allocation of time slices follows Linux NICE mechanism. That is, the time slice allocated to the highest and lowest priority processes is set to 800 ms and 5 ms, respectively. According to our measurements, the time elapsed for performing a context switch is 7 μs on the machine with Intel Core i7-7800X CPU running Linux kernel version 4.4.

The DRAM size is tailored to match the working set, while the ULL storage device size accommodates the memory footprint of general-purpose processes. The working set is defined as the minimum memory size capable of capturing over 99% of accesses resulting from CPU cache misses. The memory footprint refers to the total size of memory pages accessed by a process. The access latency for DRAM and the ULL storage device, exemplified by Samsung Z-NAND SSD with latencies of 50ns [3] and 3us [9], respectively. To model the data bus, we simulate a 4-lane PCIe 5.x host interface between the DRAM and ULL devices, providing approximately 3.983 GB/s bandwidth per lane.

Nine representative traces covering six general-purpose processes and three data-intensive processes are evaluated in our evaluations. Specifically, six general-purpose processes include one deep learning application (i.e., using Caffenet on inferencing 160 images), one from SPEC CPU® 2006 (i.e., Wrf), three from SPEC CPU® 2017 (i.e., Blender, Xz, and DeepSjeng), and one from graph application (i.e., running community detection algorithm on the graph framework, GraphChi [8].) Three data-intensive processes include one from Graph500 benchmark (i.e., single shortest path) and two graph applications (i.e., running random walk and page rank on GraphChi). We build four synthesis process batches by selecting six processes among the nine traces, including different numbers of data-intensive processes. All four process batches comprise Wrf, Blender, and community detection. Besides, No_Data_Intensive comprises Caffe, DeepSjeng, and Xz. 1_Data_Intensive comprises Caffe, DeepSjeng, and random walk. 2_Data_Intensive comprises DeepSjeng, random walk, and Graph500. 3_Data_Intensive comprises random walk, Graph500, and page rank.

[4]Traditional runahead execution runs the pre-execution during handling cache misses, but ours does the pre-execution during handling page faults.

[5]It groups a static number of pages with continuous page id into a page-on-page unit and fetches an entire unit during handling a page fault.

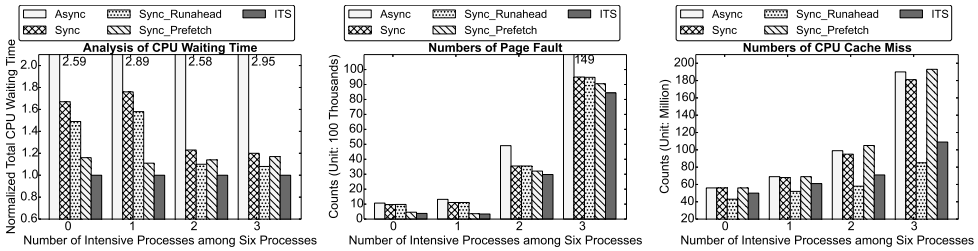
4.2 Evaluation Results

4.2.1 Evaluation on the Total CPU Idle Time. The proposed idle-time-stealing design steals the idle CPU resource for pre-caching data to memory and CPU cache to reduce future page faults and the total CPU idle time. The definition of CPU idle time is the time that the CPU's progress cannot proceed because it is waiting for the completion of memory or storage requests. Figure 4 shows the results of the normalized total CPU idle time and two supportive metrics: the number of page faults and CPU cache misses. Figure 4a provides the results of the normalized total CPU idle time under applying our ITS design against the four baseline approaches, where the x-axis indicates four different process batches and the y-axis shows the total CPU idle time normalized to the ITS design. Comparing the ITS design with the four baseline approaches (including Async, Sync, Sync_Runahead, and Sync_Prefetch), the total CPU idle time can be saved at most 66%, 43%, 37%, and 15%, and saved at least 61%, 17%, 7%, and 10% respectively.

Two main reasons explain the significant reduction of the total CPU idle time. First, for tackling the process batches with fewer data-intensive processes (i.e., no_Data_Intensive and 1_Data_Intensive), our self-improving thread can effectively reduce the numbers of page faults as shown in Figure 4b, where the unit of the y-axis is a hundred thousand counts. With running no_Data_Intensive and 1_Data_Intensive, ITS can save at least 65% and 61% of the numbers of page faults compared with the Async, and Sync approaches. The reason is that the access behaviors of non-data-intensive processes are easily predicted by our page-prefetch policy. Thus it can accurately prefetch those pages to memory. Second, for the process batches with more data-intensive processes (i.e., 2_Data_Intensive and 3_Data_Intensive), our self-improving thread can also effectively reduce the numbers of CPU cache misses as shown in Figure 4c, where the unit of the y-axis is a million counts. The reduction of the CPU cache misses is contributed by the fault-aware pre-execute policy. Based on our design, the fault-aware pre-execute policy will only be triggered during the OS handling a page fault, and thus the fault-aware pre-execute policy can save more CPU cache misses if more page faults are triggered.

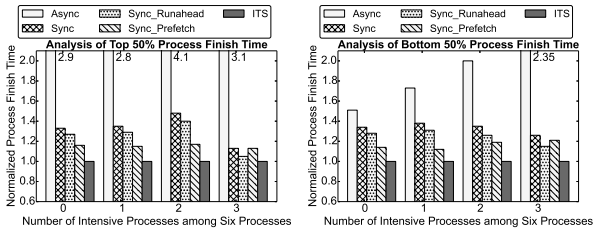
Our ITS design can still outperform the Sync_Runahead approach in all cases, even though the Sync_Runahead approach is more effective in reducing CPU cache misses, as shown in Figure 4c. The reason is that our page-prefetch policy will reduce the number of page faults, and thus our ITS design runs the pre-execute execution infrequently than the Sync_Runahead approach. But, handling page faults is more time-consuming than handling CPU cache misses. On the other hand, our ITS design can also outperform the Sync_Prefetch approach, which executes the virtual-address-based page-prefetch policy to minimize the number of page faults. The reason is that not only running both pre-execution and page prefetching, our ITS design will also schedule the self-sacrificing thread to further save the number of page faults. The rationale is that the self-sacrificing thread avoids pages belonging to low-priority processes to kick out high-priority process's pages so as to avoid both high-priority and low-priority processes contending the memory resources.

4.2.2 Evaluation on the Average Process Finish time. To demonstrate the efficacy of the self-sacrificing thread, Figure 5 presents an analysis of average process finish times. We highlight the average finish times of the top 50% and bottom 50% priority processes in each batch, in Figure 5a and Figure 5b, respectively. The x-axis represents different process batches, and the y-axis shows average process finish times normalized to our ITS design. Figure 5a illustrates the effectiveness of our ITS design in reducing average process finish times for the top-priority processes, with savings ranging from 75% to 14% when compared to the four baseline



(a) Normalized CPU Idle Time. **(b)** Numbers of Page Faults. **(c)** Numbers of CPU Cache Misses.

Figure 4. Evaluation on CPU Idle Time.



(a) Top 50% Processes. **(b)** Bottom 50% Processes.

Figure 5. Evaluation on the Average Process Finish time

approaches. In Figure 5b, comparing our ITS design with baseline approaches (Async, Sync, Sync_Runahead, and Sync_Prefetch), the finish time of the three lower-priority processes can be saved up to 58%, 27%, 24%, and 17%, and at least 34%, 21%, 13%, and 11%, respectively. These findings affirm that the self-sacrificing thread in our ITS design enhances the progress of both high- and low-priority processes. That is, self-sacrificing threads compel low-priority processes to yield to high-priority ones before completing their allocated time slice. This enables low-priority processes to access additional resources, including memory and CPU, without interference from high-priority processes that finish earlier.

5 Conclusion

Synchronous I/O mode becomes promising when the storage response time catches up with the context switch overhead. However, synchronously waiting for the I/O completion incurs a considerable amount of CPU idle time. An Idle-Time-Stealing (ITS) design is proposed to better utilize otherwise-wasted I/O busy time. Practically, we propose to have a self-improving kernel thread to do page-prefetching and fault-aware pre-execute execution for high-priority processes during some synchronous I/O time. We also propose a self-sacrificing kernel thread to proactively switch out the low-priority process so as to favor high-priority process executions. The evaluation results show that our ITS design could effectively reduce the CPU idle time by around 61%∼66% and 17%∼43%, than those with totally the asynchronous I/O mode and the synchronous I/O mode, respectively.

6 Acknowledgement

This work was supported in part by National Science and Technology Council under grant nos. 112-2223-E-001-001, 111-2221-E-001-013-MY3, 111-2923-E-002-014-MY3, 113-2927-I-001-502, and 112-2222-E-A49-002-MY2; Academia Sinica under grant no. AS-IA-111-M01; the Research Grants Council of Hong Kong SAR (Project No. CUHK14208521) and Ministry of Education under Yushan Young Fellow Program.

References

[1] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Communications of the ACM 60*, 4 (2017), 48–54.

[2] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[3] Chen, R., Shao, Z., and Li, T. Bridging the i/o performance gap for big data workloads: A new nvdimm-based approach. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), IEEE, pp. 1–12.

[4] Chen, Y.-C., Wu, C.-F., Chang, Y.-H., and Kuo, T.-W. Exploring synchronous page fault handling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41*, 11 (2022), 3791–3802.

[5] Dundas, J., and Mudge, T. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing* (1997), pp. 68–75.

[6] Jin, Y., Wu, C.-F., Brooks, D., and Wei, G.-Y. S^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems 36* (2024).

[7] Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., and Witchel, E. Coordinated and efficient huge page management with ingens. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 705–721.

[8] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)* (2012), pp. 31–46.

[9] Lab, S. M. S. Technology brief: Ultra-low latency with samsung z-nand ssd.

[10] Mutlu, O., Kim, H., and Patt, Y. N. Techniques for efficient processing in runahead execution engines. In *32nd International Symposium on Computer Architecture (ISCA'05)* (2005), IEEE, pp. 370–381.

[11] Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* (2003), IEEE, pp. 129–140.

[12] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices* (2007), vol. 42, ACM, pp. 89–100.

[13] Panwar, A., Prasad, A., and Gopinath, K. Making huge pages actually useful. In *ACM SIGPLAN Notices* (2018), vol. 53, ACM, pp. 679–692.

[14] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating system concepts.* John Wiley & Sons, 2006.

[15] Waddington, D., and Harris, J. Software challenges for the changing storage landscape. *Communications of the ACM 61*, 11 (2018), 136–145.

[16] Weaver, V. M. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath* (2013), vol. 13.

[17] Wu, C.-F., Chang, Y.-H., Yang, M.-C., and Kuo, T.-W. Joint management of cpu and nvdimm for breaking down the great memory wall. *IEEE Transactions on Computers* (2020).

[18] Wu, C.-F., Chang, Y.-H., Yang, M.-C., and Kuo, T.-W. When storage response time catches up with overall context switch overhead, what is next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 11 (2020), 4266–4277.

[19] Wu, C.-F., Wu, C.-J., Wei, G.-Y., and Brooks, D. A joint management middleware to improve training performance of deep recommendation systems with ssds. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (2022), pp. 157–162.

[20] Yang, J., Minturn, D. B., and Hady, F. When poll is better than interrupt. In *FAST* (2012), vol. 12, pp. 3–3.

[21] Zhang, J., Kwon, M., Gouk, D., Koh, S., Lee, C., Alian, M., Chun, M., Kandemir, M. T., Kim, N. S., Kim, J., et al. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 477–492.