FPGA Implementation of Hardware-based Demand Paging on RISC-V Architecture

Wenjing Jin*
Department of Computer Science and Engineering
Seoul National University
Seoul, Korea
wenjing.jin@snu.ac.kr

Jeonghun Gong*
Department of Computer Science and Engineering
Seoul National University
Seoul, Korea
jh.gong@snu.ac.kr

Jae W. Lee
Department of Computer Science and Engineering
Seoul National University
Seoul, Korea
jaewlee@snu.ac.kr

***Abstract*—The Operating System (OS) manages disk I/O operations through a dedicated software stack. Traditionally, the overhead of this software stack was considered negligible for storage systems based on hard disks, due to their inherently long access latencies. However, the advent of low-latency Solid State Drives (SSDs) has significantly increased the impact of OS software overhead on I/O processing efficiency. To address this, hardware-based demand paging has been proposed as a solution to reduce software overhead associated with handling page faults. Though theoretically advantageous, the real-world efficacy of hardware-based demand paging has been mostly untested, with investigations largely confined to simulations. This paper introduces an implementation and evaluation of hardware-based demand paging on an FPGA-based RISC-V system, assessing its viability. Our findings indicate that, relative to traditional OS-based methods, implementing hardware-based demand paging substantially reduces software overhead in actual hardware scenarios, improving the performance of the FIO read random benchmark by up to 85.4%.**

***Index Terms*—Demand paging, Virtual memory, Operating systems, CPU architecture, Hardware extension**

I. Introduction

Demand paging stands as a cornerstone technique in modern computer systems, facilitating data transfer between disk and main memory [8]. Upon a CPU's attempt to access data via memory load/store instructions without available mapping information, an exception is triggered. This exception signals the OS to fetch the requisite data from the disk or the OS's page cache. To optimize CPU resource allocation during such disk operations, the OS typically executes a context switch, thereby enabling other processes to execute.

The recent advent of ultra-low latency storage devices, such as Samsung Z-SSD [9] and Intel Optane SSD [4], has dramatically reduced the performance gap between memory and disk. Disk access times have shrunk from tens of milliseconds to just a few microseconds [6], elevating the OS stack's relative overhead in handling page faults. Consequently, the kernel stack overhead now represents a significant fraction of the total time spent handling page faults. Moreover, the context-switching process can lead to pollution of microarchitectural resources [11], including caches, translation lookaside buffers (TLBs), and branch predictors, thereby impairing the performance of user applications. This dramatic shift underscores the need to reevaluate conventional demand paging mechanisms to address page faults more efficiently in environments characterized by low-latency storage devices [6], [12].

To mitigate software overhead in page fault handling, Lee et al. have proposed a hardware-based demand paging technique [5]. This method circumvents the necessity for OS kernel involvement in page fault resolution by leveraging hardware capabilities directly. Their approach incorporates three hardware extensions and supportive OS mechanisms: firstly, developing an LBA-augmented page table structure alongside necessary modifications to the memory management unit (MMU) for its accommodation; secondly, introducing the storage management unit (SMU) as a novel hardware component to facilitate direct CPU access to NVMe SSDs; and thirdly, the adaptation of the OS kernel to facilitate hardware demand paging through efficient resource allocation and metadata management.

While previous explorations into hardware-based demand paging showed theoretical promise, they largely depended on emulation within x86 architectures and system simulations, not fully addressing its feasibility on actual hardware. This paper elaborates on the deployment of the hardware paging technique within an FPGA-based RISC-V architecture, providing a concrete demonstration of these theoretical concepts in practice. Our comprehensive evaluation confirms the technique's ability to significantly alleviate OS software overhead associated with page fault handling. Remarkably, this leads to an up to 85.4% reduction in average latency for FIO [3] random reads, offering a significant performance

* These authors contributed equally to this work.

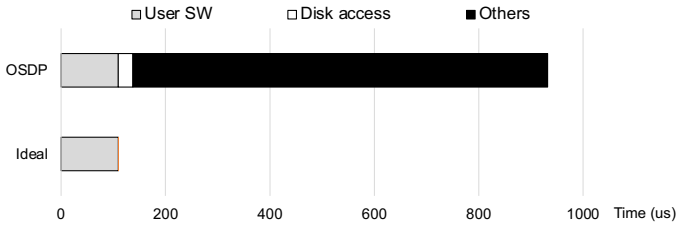

Fig. 1. Latency breakdown of FIO operations on a Xilinx VC707 FPGA with four RISC-V Rocket cores.

enhancement compared to conventional OS-based demand paging approaches.

II. Background

A. Memory-mapped File I/O

Memory-mapped file I/O facilitates the mapping of disk files into memory, enabling file reading and writing through direct memory access. This approach provides several advantages over conventional file access methods. Typically, file access involves explicit system or library calls (e.g., read or write) to load the file into memory. Memory-mapped files, however, allow the OS to manage file I/O directly between the disk and memory, removing the need for programmers to execute separate read or write operations on the storage device. This feature is particularly useful for handling files larger than the available physical memory, as the OS intelligently loads required file segments into memory and offloads others back to disk, thus easing file management for users.

Supported by demand paging in virtual memory, memory-mapped files utilize page tables to store mapping information for each page, distinguishing between resident (in-memory) and non-resident pages. When accessing a memory-mapped file, the CPU's memory management unit (MMU) queries the page table entries (PTE) for the mapping details of the requested page. If the mapping exists, the CPU accesses the corresponding memory region to fetch the necessary data. Absence of the mapping triggers a page fault, prompting the MMU to signal the OS for page loading. The OS then retrieves the data from either the page cache or through disk I/O, updates the page table accordingly, and a context switch allows the application to continue, seamlessly integrating disk-based files with memory load/store semantics.

B. Page Fault Overhead with Low-latency Storage Device

The rapid improvement in SSD performance has highlighted a critical bottleneck in operating systems: the software stack's time consumption in handling page faults. Conventionally, the impact of OS kernel-based page fault handling, which necessitates context switching and software execution, was mitigated by the relatively long access times of mechanical hard drives. However, with the advent of ultra-low latency storage devices like Samsung Z-SSD [9] and Intel Optane SSD [4], this overhead has become significant, adversely affecting the efficiency of user application processing by polluting microarchitectural resources such as CPU caches and TLBs.

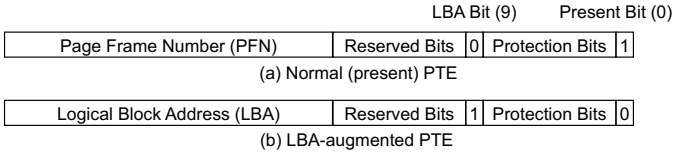

Fig. 2. Adaption of the LBA-augmented PTE within the RISC-V architecture for hardware-based demand paging implementation.

To quantify the impact of the software stack on page fault handling, we conduct an analysis on a system featuring four RISC-V Rocket cores [2], clocked at 100MHz, running Linux. This system was implemented using a Xilinx VC707 FPGA board, and equipped with an Intel Optane SSD. Utilizing the FIO benchmark's *mmap* engine, we measure the page fault latency for 4KB random read. In our experimental setup, we define two scenarios:

- **OSDP (OS-based Demand Paging):** Runs the FIO *mmap* benchmark without modifications to represent the combined latency of user software execution and page fault handling.
- **Ideal:** Enhances the *mmap* call with the *MAP_POPULATE* flag, pre-loading all data into memory to eliminate page faults and I/O, thereby isolating user software execution time.

The difference in latency between the OSDP and ideal scenarios directly measures the overhead introduced by page fault handling. As shown in Figure 1, the OSDP scenario results in an average latency of 921.922us, while the ideal scenario achieves a latency of 109.372us. Subtracting the ideal scenario's latency from OSDP's yields a page fault handling time of 812.55us, with SSD I/O accounting for only 28.16us of this. The substantial remainder, 794.39us (or 97.8% of the page fault handling time), is attributed to the overhead of context switching and software execution. These findings underscore the dominance of software overhead in the total time spent on page fault handling, affirming the critical need for optimization in systems equipped with low-latency storage devices.

III. FPGA Implementation of Hardware-based Demand Paging

A. LBA-augmented Page Table and MMU

In our study, we adapt the page table structure of the RISC-V architecture to incorporate the LBA-augmented page table. This adaptation involves repurposing the 9th bit of the PTE, traditionally unused in the RISC-V specification, to serve as the LBA bit. Figure 2 depicts the modified PTE layout utilized in our implementation. We integrate logic within the MMU of the Rocket core to facilitate command issuance to the storage management unit (SMU) through memory-mapped IO (MMIO), specifically for handling cases where a non-resident page is identified with the LBA bit set to 1. Upon successful page fault resolution by the SMU, the MMU proceeds with the address translation process as usual. In cases where the

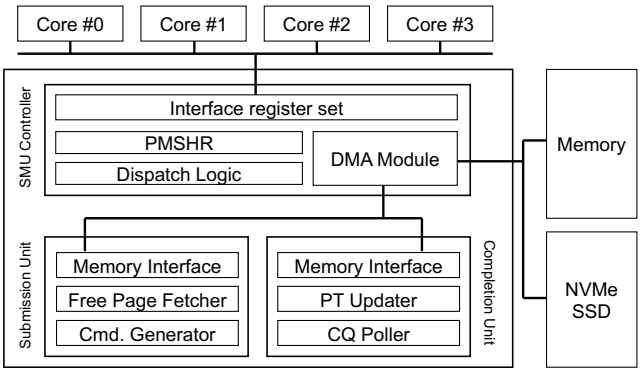

Fig. 3. Overview of the storage management unit (SMU) implemented on the FPGA platform.

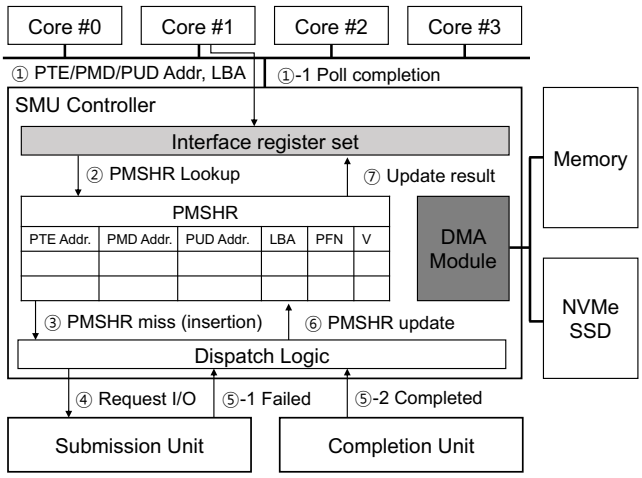

Fig. 4. Structure of the SMU controller.

SMU cannot successfully manage the page fault, the MMU is programmed to revert to standard protocol by triggering an exception. This allows the operating system's page fault handler to take over and resolve the fault.

B. Storage Management Unit (SMU)

The SMU is developed and integrated into our FPGA-based system to facilitate hardware-based demand paging. As depicted in Figure 3, the architecture of the modified SMU tailored for our FPGA platform comprises several key components:

- **SMU controller:** Facilitates communication with both the OS and the MMU, managing the logistics of page fault occurrences, including the handling of duplicate page faults. This coordination ensures a streamlined demand paging process.
- **Submission unit:** Responsible for generating NVMe commands [7] tailored to specific storage operations, this unit plays a crucial role in initiating disk I/O operations by dispatching these commands to the NVMe SSD.
- **Completion unit:** Tracks the completion of NVMe I/O operations, ensuring that the page table is accurately updated upon the successful execution of storage commands.

SMU Controller. The SMU controller orchestrates communication between the OS, MMU, and main memory, efficiently handling cases of duplicate requests to mitigate unnecessary disk access. Communication with the OS and MMU occurs through MMIO, where interface registers within the SMU controller are accessible to the CPU for read or write operations via the memory interface. To operationalize the SMU, the OS must allocate memory and NVMe resources, notifying the SMU by updating these interface registers. Based on the provided data, the SMU then undertakes page allocation and disk access tasks. Page fault handling requests are initiated by the MMU writing to the interface registers.

To support memory interactions essential for the SMU's operation, the SMU controller is equipped with a direct memory access (DMA) module. This module autonomously conducts requested memory operations for the submission and completion units, delivering the outcomes directly to them. Additionally, the SMU controller contains page miss status holding registers (PMSHR), specialized registers tracking ongoing page fault processing activities. Functioning similarly to the miss status holding registers (MSHR), the PMSHR filter duplicate requests, leveraging a content addressable memory (CAM) structure to rapidly identify if there are any redundant page fault requests based on PTE addresses.

Figure 4 illustrates the command execution workflow managed by the SMU controller, detailing each step in the process. Initially, ① the SMU receives from the MMU all the necessary information for conducting I/O operations and updating the page table, including the hardware thread ID (HART ID), LBA, and the addresses of the PTE, PMD, and PUD. Upon receiving a page fault handling request, ② the SMU first consults the PMSHR to check if the same page fault is already being processed. If an ongoing process for the requested PTE address is found, the SMU waits until this processing completes before updating the corresponding hardware register of the MMU that requested the page fault, indicating the fault's resolution. If there is no current processing for the requested PTE address, ③ the controller inserts information about the page fault into the PMSHR under the entry corresponding to the HART ID. ④ It then instructs the submission unit to address the page fault by providing the LBA and PMSHR entry ID. Once the submission/completion unit completes the processing ⑤, the SMU controller ⑥ removes the respective entry from the PMSHR and ⑦ updates the hardware register corresponding to the requesting MMU with the page fault handling result, effectively communicating the resolution of the page fault.

Submission Unit. The submission unit is responsible for allocating memory space for NVMe I/O operations, generating NVMe commands, and managing disk I/O requests to the NVMe SSD. This unit comprises a memory interface, a free page fetcher, and a command generator. The memory interface crucially directs all memory operations to the SMU

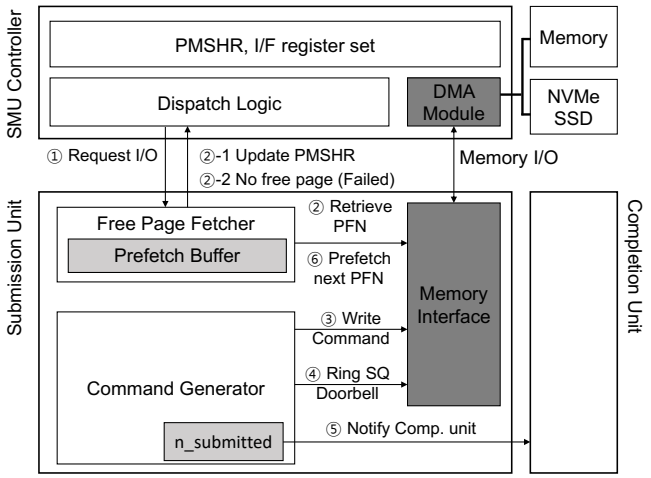

Fig. 5. Structure of the SMU submission unit.

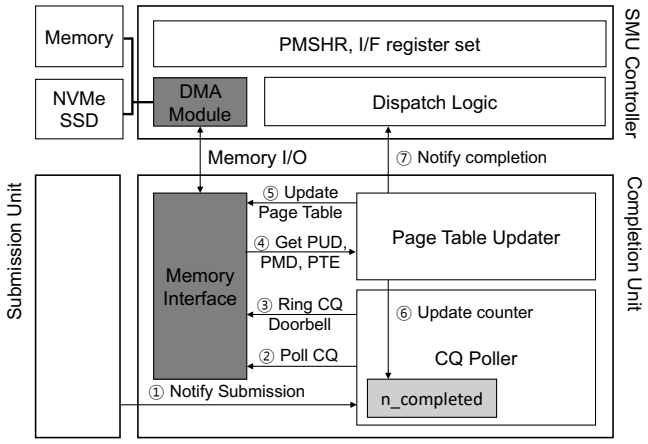

Fig. 6. Structure of the SMU completion unit.

controller's DMA module and ensuring the results are conveyed to the internal components. The free page fetcher, an essential component of the submission unit, retrieves free pages collected by OS for I/O use. To facilitate this, the OS creates a circular queue, termed the free page pool, populating it with the page frame number (PFN) of free pages. This setup communicates through interface registers that capture the free page pool's base address, head, and tail values. Upon establishing the queue, the OS records its base address in the SMU's hardware register. Free pages are added to the queue periodically by the OS in the background, which updates the tail register accordingly. As the SMU utilizes pages from the queue, it updates the head register, which signals to the OS that those pages have been used.

Figure 5 illustrates the execution flow of the submission unit. ① On receiving a new page fault handling request, ② the free page fetcher first inspects the head/tail registers of the free page pool to check free page availability for allocation. Upon a successful allocation, ②-1 the fetcher updates the PMSHR's PFN field to facilitate the page table update. In situations where page allocation is not feasible, ②-2 it informs the SMU controller about the failure in handling the page fault, handing over the resolution process to the OS's page fault handler. Following successful page allocation, the command generator, leveraging the LBA information and the allocated page as provided by the SMU controller, formulates an NVMe command. This command, distinguished by a PMSHR entry number in the cmdid field of the NVMe submission command, ③ is then relayed to NVMe submission queue (SQ) in the main memory via the SMU controller's DMA module, and ④ activates the NVMe SSD by writing to its SQ doorbell register. Concurrently, ⑤ an update is made to the *n_submitted* register connected to the completion unit, signaling the issuance of a new command.

After processing a command, the submission unit features a prefetch function to quickly handle the next command by pre-fetching free memory pages. This action is triggered after notifying the completion unit of an NVMe command delivery and is executed by the free page fetcher. If it can allocate a memory page after checking the free page queue's registers, the page is assigned and stored in the free page fetcher's prefetch register. This prefetched page enables the rapid creation of an NVMe command for the next operation without accessing memory, thereby accelerating the SMU's actions. Once page fetching is concluded, the submission unit pauses, awaiting further instructions from the SMU controller.

Completion Unit. The completion unit ensures that commands issued by the submission unit are completed and communicates the results back to the SMU controller. It consists of a memory interface, a CQ (completion queue) poller, and a page table updater. The CQ poller monitors the NVMe completion queue (CQ) for the completion of commands, signaling to the NVMe that a command has been acknowledged. The page table updater then updates the page table according to the completed information and notifies the SMU controller of the successful completion of NVMe I/O operations.

Figure 6 illustrates the process within the completion unit. The workflow begins ① when the CQ poller observes a discrepancy between the number of commands submitted (as recorded in the *n_submitted* register) and the number of commands completed (as recorded in the *n_completed* register). It then ② polls the NVMe CQ to check the outcome of NVMe commands. Upon verifying a command's completion, ③ the memory interface updates the NVMe SSD's CQ doorbell register, indicating receipt of the completion information. Subsequently, ④ the page table updater reviews the cmdid field within the CQ entry to determine which PMSHR entry matches the executed command, retrieving the PMSHR value to identify the PUD, PMD, PTE, and the PFN that holds the data retrieved from the disk. The page table updater then ⑤ reads the values corresponding to the PUD and PMD addresses, amends them to incorporate the LBA bit, and utilizes the assigned PFN to generate a new PTE value, including the protection bit, LBA bit, and present bit. Following this, ⑥ the value in the *n_completed* register is

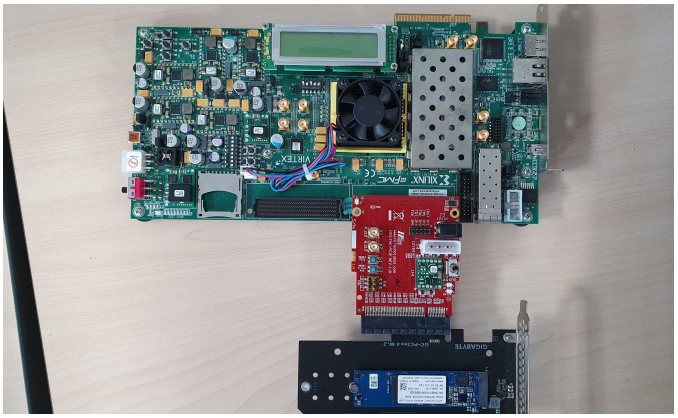

Fig. 7. The Xilinx VC707 FPGA evaluation board setup.

TABLE I
SPECIFICATIONS OF THE FPGA PLATFORM

Hardware Platform	Xilinx VC707 evaluation board
CPU	4-core RISC-V Rocket @ 100MHz
Kernel Version	Linux 4.15.0
Memory	1GB LPDDR3
NVMe SSD	32GB Intel Optane SSD

incremented, marking the completion of a command. Finally, ⑦ the completion unit informs the SMU controller about which PMSHR entry has been processed and prepares to monitor the completion of the next NVMe command.

IV. EVALUATION

A. Experimental Environment

For our study, we chose the Freedom U500 platform [10], generously provided as open-source by SiFive. This enabled us to program a CPU with hardware demand paging capabilities onto a Xilinx VC707 FPGA board. Figure 7 showcases our FPGA setup. This platform allows us to leverage a robust environment designed for FPGA programming, equipped with four RISC-V Rocket cores [2], a memory controller, a PCIe Root Complex for NVMe SSD attachment, an SD card reader for Linux booting, and a UART interface for user interaction. The RISC-V Rocket, a 5-stage in-order CPU, is implemented in Chisel3 [1]. We adapt the Rocket core's MMU to support hardware demand paging, integrating the SMU as an MMIO device, facilitating direct communication between the CPU and NVMe SSD without requiring OS intervention. Table I encapsulates the specifications of our system realized on the FPGA platform, detailing the hardware platform, CPU configuration, kernel version, memory, and NVMe SSD specifics.

To evaluate the performance of the hardware-based demand paging system implemented on an FPGA, we utilized the FIO [3] benchmark version 3.19-27, specifically configured to perform 4KB random reads using the *mmap* engine. Table II illustrates the details of evaluation configuration. This setup allowed us to conduct a comprehensive analysis of the impact of hardware-based demand paging on I/O performance metrics across these varied scenarios.

TABLE II
EVALUATION CONFIGURATION

	No free page refill	**Small**	**Large**
File size	320MB	4GB	16GB
Access size per thread	64MB	64MB	2GB
Free page pool size	320MB	4MB	4MB
Number of threads	1	1, 2, 4	1, 2, 4

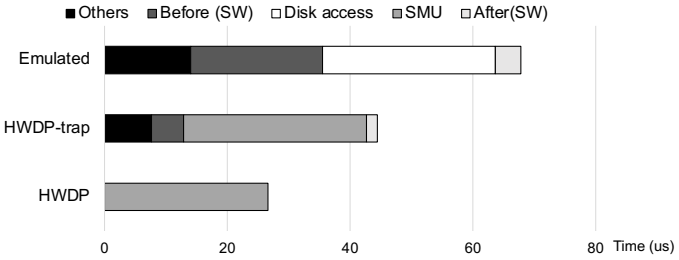

Fig. 8. Page fault handling time analysis.

In this evaluation, we conduct a comparative performance analysis between OS-based demand paging (OSDP) and hardware-based demand paging (HWDP). To thoroughly assess the benefits and performance enhancements offered by the hardware-based page fault mechanism, particularly when offloaded to hardware, we examined three distinct implementations of hardware-based demand paging:

- **Emulated:** This implementation emulates the functionality of hardware demand paging using the OS's page fault handler, replicating the process in software.
- **HWDP-trap:** Similar to conventional page faults, this implementation traps page faults but processes them in hardware if possible, by issuing commands to the SMU from the OS's page fault handler.
- **HWDP:** This implementation bypasses OS traps, with the MMU directly issuing commands to the SMU to handle page faults.

B. Page Fault Latency Analysis

Figure 8 presents an page fault latency analysis. To isolate the latency attributed solely to page fault processing, all background threads in the kernel were disabled, and the experiments were conducted to avoid page refills. The analysis breaks down page fault timing into before(SW), disk access, after(SW), and others. Before(SW) encompasses the time from entering the kernel's page fault handler to just before the device (SMU or NVMe SSD) is activated. Disk access time, in the emulated scenario, spans from writing to the NVMe SQ doorbell to just before writing to the NVMe CQ doorbell. The SMU's operation time is represented in the HWDP and HWDP-trap scenarios. After(SW) covers the period from the device access completion to the return from the fault handler. The other category encapsulates the duration of context switching and additional software execution time, exacerbated by resource contamination from OS page fault handling. In the emulated scenario, of the total 69.1us, disk access accounted for 28.1us, leaving 41.0us for non-disk

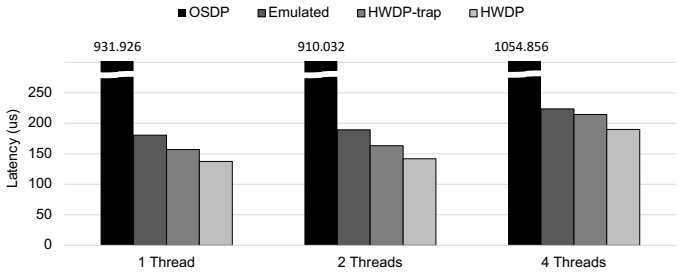

Fig. 9. Average latency of FIO for small access size.

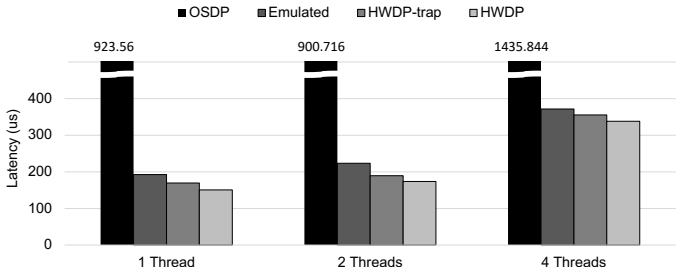

Fig. 10. Average latency of FIO for large access size.

access activities, indicating that a significant 66.6% of page fault handling time was consumed by processes other than disk access. In the HWDP-trap scenario, only 16.0us (34.9%) of the total 45.8us was spent on non-hardware access activities. Conversely, the fully hardware-driven HWDP managed to process page faults in just 27.9us, highlighting the efficiency of complete hardware-based demand paging in minimizing page fault latency.

C. Application Performance

Figure 9 displays latency results from executing the FIO workload in scenarios where the access size is kept below DRAM capacity, eliminating the necessity to offload pages back to the disk. The experiment adhered to the *small* category parameters depicted in Table II. Notably, the performance enhancements are less marked in the 4-thread scenario, presumably due to thread contention, yet a performance improvement of up to 6.78× (representing an 85.4% reduction in latency) compared to OSDP was noted.

Figure 10 illustrates latency results from conducting the FIO workload in conditions where the access size surpasses DRAM capacity, necessitating data eviction from memory. This experiment conformed to the *large* category specifications listed in Table II. Findings indicate that HWDP yields a latency improvement ranging from 4.24× to 6.12× over OSDP, and from 1.09× to 1.27× over the emulated scenario, varying by the number of threads. The most significant performance gains are observed with a smaller number of threads. In scenarios featuring 4 threads, matching the number of CPU cores, page evictions extended the OS's page fault handling time and intensified thread contention, diminishing the overall performance uplift.

V. Conclusion

This paper delves into the architecture required for implementing a previously proposed hardware-based demand paging technique on real hardware and executes this implementation on an FPGA platform utilizing RISC-V architecture. Additionally, we adapted the RISC-V OS kernel to support this innovative approach, conducting comprehensive performance evaluations on the system. Through this exploration, we not only demonstrated the practicality of realizing the proposed technique on actual hardware but also showcased that utilizing the proposed structure significantly enhances page fault handling and data retrieval, closely mirroring disk access times. Notably, our research marks a pivotal advancement in integrating memory management techniques directly into hardware, showcasing a practical method to markedly reduce latency associated with page faults in systems where rapid data access is paramount. This breakthrough resulted in up to an 85.4% performance boost in the FIO 4KB random read, highlighting the efficiency of the hardware-based demand paging solution in enhancing system speed and performance.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[2] Chipsalliance, "Rocket-chip," hhttps://github.com/chipsalliance/rocket-chip.

[3] "Flexible I/O tester," https://github.com/axboe/fio, 2016.

[4] Intel, "Intel Optane SSD DC P4800X/P4801X," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf, 2018.

[5] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, "A case for hardware-based demand paging," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1103–1116.

[6] 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 *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, 2019, pp. 603–616.

[7] "NVM express," https://nvmexpress.org.

[8] D. A. Patterson and J. L. Hennessy, *Computer organization and design*. Newnes, 2013.

[9] Samsung, "Samsung Z-SSD SZ985," https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf, 2018.

[10] Sifive, "Sifive Freedom," https://github.com/sifive/freedom.

[11] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 33–46.

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