LightPC: Hardware and Software Co-Design for Energy-Efficient Full System Persistence

Sangwon Lee[§], Miryeong Kwon[§], Gyuyoung Park, Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST),
http://camelab.org

***Abstract*—We propose LightPC, a lightweight persistence-centric platform to make the system robust against power loss. LightPC consists of hardware and software subsystems, each being referred to as open-channel PMEM (*OC-PMEM*) and persistence-centric OS (*PecOS*). OC-PMEM removes physical and logical boundaries in drawing a line between volatile and non-volatile data structures by unshackling new memory media from conventional PMEM complex. PecOS provides a single execution persistence cut to quickly convert the execution states to persistent information in cases of a power failure, which can eliminate persistent control overhead. We prototype LightPC's computing complex and OC-PMEM using our custom system board. PecOS is implemented based on Linux 4.19 and Berkeley bootloader on the hardware prototype. Our evaluation results show that OC-PMEM can make user-level performance comparable with a DRAM-only non-persistent system, while consuming 73% lower power and 69% less energy. LightPC also shortens the execution time of diverse HPC, SPEC, and In-memory DB workloads, compared to traditional persistent systems by 4.3×, on average.**

I. Introduction

Long-running applications in server-class computing domains such as data centers and high-performance computing often face unexpected power loss [1, 2]. A wide spectrum of server services employs journaling and checkpoint-restarts to make the system robust and crash consistence on a power failure [3–6]. While such persistence mechanisms can control a crash by recovering their committed transaction, they in practice suffer from significant performance degradation because of several aspects such as data replications, I/O operations serialized by a single thread, and lock/barriers for data management [7–10]. In addition, journaling and checkpointing waste system resources and power to handle persistence control requests [11, 12].

Instead, we can leverage a persistent memory module (PMEM) supporting 10× higher storage capacity with excellent non-volatile capabilities, offered by 3D-Xpoint [13, 14], which is a variation of phase change memory (PRAM) [15–17]. However, modern systems employing PMEM unfortunately undergo unexpected performance and latency variation [18–20]. For example, [18] reports that the processor latency with PMEM is non-deterministic and significantly varies, which is 3× worse than a legacy system using DRAM. One of the reasons behind the unexpected latency variation is that the memory complex and DIMM-level microarchitecture of PMEM are self-contained, a complicated system similar to high-performance SSDs, not like a DRAM DIMM [19, 20].

In addition, PMEM is not free from crash consistence management, which degrades the overall performance if the target system considers persistence as first-class citizens. PMEM employs internal DRAM buffers and leverages local-node external DRAMs as a memory cache to catch up with DRAM performance. Thanks to the hybrid design, PMEM satisfies diverse demands of server and application markets. However, it requires a support of fault-tolerant software libraries that periodically flush the buffers and synchronize the execution states between DRAM and PMEM [21, 22]. This unfortunately incurs more than 100% space overhead and significantly diminishes performance [23, 24]. In cases where the systems insist on making in-memory data durable and consistent across multiple processes running on their cores, we observe that PMEM makes user-level performance 8.7× worse and consumes 1.6× more power, compared to the DRAM-only legacy system. We will discuss the details of the observation in Section II-B.

We propose a **Light**weight **P**ersistence-**C**entric platform (*LightPC*) to make the systems resilient against power failures. The insight of this work is simple and brand-new, but it is not incarnated in a real system yet; if one can make all data structures and execution flows persistent by reconsidering a vertical design of the current memory subsystems, it addresses the crash control overhead (on a power failure) and eliminates frequent memory flushes/synchronizations imposed by the PMEM's hybrid design. LightPC supports lightweight orthogonal persistence by removing physical and logical boundaries to draw the line between persistent and non-persistent data structures. To this end, we need to first make the PRAM latency deterministic and minimize the performance impact induced by DRAM-related hardware modules. In addition, we require ensuring a solid CPU control mechanism to appropriately suspend/resume many non-persistent states residing in CPU-side storage and make them durable, such that multiple processes/threads running across different cores can keep continuing their work without a negative impact on the power failures.

In this work, we use a hardware and software cooperative technique that consists of two major components: i) *Open-Channel PMEM* (OC-PMEM) and ii) *Persistence-Centric OS* (PecOS). OC-PMEM waives non-persistent memory paths and reduces PRAM access penalties, while PecOS quickly turns execution states of running processes to be persistent.

[§]These authors contributed equally to this work.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527397

Specifically, OC-PMEM unshackles new memory media from conventional PMEM complex. In parallel, it lets the reliability and parallelism management be coupled with the memory subsystem. OC-PMEM minimizes non-persistent states in the datapath, thereby exhibiting deterministic performance. On the other hand, PecOS modifies Linux task scheduler and power management backplane to provide a single *execution persistence cut* where multiple cores can make sure their states are persistent upon a power failure. It guarantees that on-the-fly memory operations and non-persistent execution states (e.g., per-process control, peripheral contexts, etc.) can be safely stored in a persistent space within a period much shorter than a standard power hold-up time. Later, PecOS almost immediately revives and executes all the offlined processes on CPU from the persistence cut when the power is recovered. LightPC does not insist on a source-level modification thereby being transparent to and well harmonized with emerging non-volatile memories.

The following summarizes our contributions:

• **Open-Channel architecture for PMEM.** We introduce open-channel architecture to achieve deterministic performance of PMEM, which consists of two major hardware modules, persistent support module and bare-metal PRAM DIMM channels. The former manages resource conflict and reliability issues being under the computing complex, thereby removing the DIMM-side firmware and internal volatile memory components from the memory access path. The latter is designed towards exposing the PRAM media to the host, but increasing parallelism to service on-the-fly requests as soon as possible.

• **Transparent full persistence mechanism.** In a real system, making the execution states persistent at anywhere of multi-core execution stages is not an easy task. PecOS offers the persistence cut by implementing a multi-step offlining and onlining mechanism, called Stop-and-Go (SnG). SnG first makes sure that all system contexts are safely stored back. SnG then stops all processes and lockdowns them across multiple cores. Once it ensures an unchanging environment for all the cores, SnG suspends peripherals by freezing their context and dumps each core's volatile states into a persistent space.

• **Practical non-volatile computing and prototypes.** Examining full system persistence requires fine power source control and practical system space exploration across different hardware and software barriers, which are well not captured by simulation and emulation studies. We prototype LightPC's computing complex and OC-PMEM using our custom system board, while PecOS is implemented on Linux 4.19. The prototype also includes multiple bare-metal PRAM channels connected to the processor through a multi-point network [25].

Our evaluation results on real systems show that HPC, SPEC, and In-memory DB workloads running on LightPC consume 73% lower power and exhibit performance comparable to those operating on a non-persistent system. LightPC also shortens the latency of such workloads, compared to persistent systems by 4.3×, on average, while saving power by 3.6×.

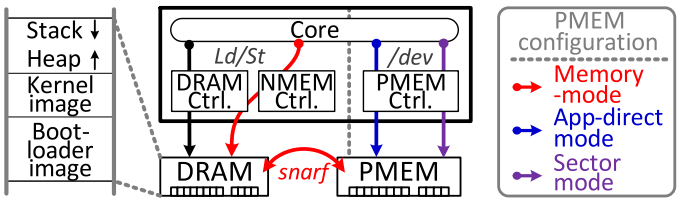


Fig. 1: Computing complex having PMEM.

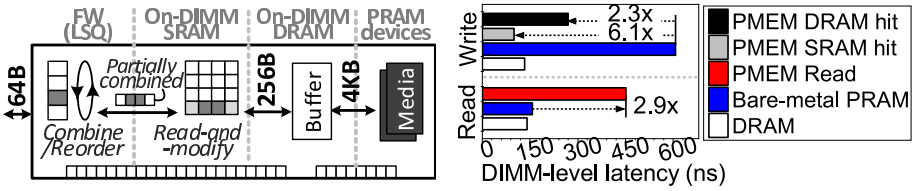
II. Background

A. Hardware Basics

Optane PMEM [13] is the fundamental ground to support data persistence in modern computers. PMEM is not a simple DIMM technology, but a comprehensive system solution to cover diverse user requirements [26, 27]. Thus, PMEM offers block storage (*sector mode*), non-persistent working memory (*memory mode*), and non-volatile memory (*app-direct mode*) that should be configured at the system initialization time [28].

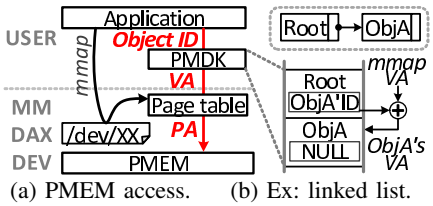
PMEM hardware complex. Figure 1 shows the computing complex of PMEM. PMEM places a couple of local-node DRAM DIMMs and PMEM DIMMs together in memory system. To appropriately manage two different memory technologies, it employs three types of memory controllers, PMEM controller, DRAM controller, and *near memory cache* (NMEM [29]) controller. While DRAM controllers manage local-node DRAM DIMMs, PMEM controllers independently interface with PMEM DIMMs. NMEM controller handles both of the memory controllers to cache PMEM data into DRAM. To minimize caching overhead, NMEM controller overlaps the latency imposed by data transfers between the local-node DRAM and PMEM, using a shared memory interface, called *snarf* [30]. The memory mode PMEM uses all those three controllers and exhibits DRAM-like performance, but it drops all the advantages of non-volatility brought by PRAM. On the other hand, the app-direct mode and sector mode PMEMs disable DRAM and NMEM controllers such that it exposes PMEM DIMMs to CPU as persistent storage. Specifically, with these two modes, PMEM DIMMs are excluded from the working memory space and mapped to a device file mounted at `/dev`. Thus, the computing complex can support the data persistent pool with different types of persistence control support, but all applications and kernel threads running on the system yet reside on the local-node DRAM DIMMs, not PMEM DIMMs.

PMEM DIMM architecture. Industrial documents indicate that PMEM DIMM includes complicated internal architectures than DRAM DIMM without detailed information [19, 26, 27]. Figure 2a shows PMEM datapath and internal architecture by performing reverse engineering of a 256GB PMEM DIMM product [19]. PMEM DIMM employs a load-store queue (LSQ) that reorders incoming requests and performs write combining if it is possible to form a 256B request (which is the physical access granularity of DIMM-level PRAM media). To handle 64B cacheline requests, PMEM DIMM integrates a set of SRAM and DRAM modules with PRAM devices into the device (as a two-level inclusive cache) [19, 27]. SRAM



(a) Internal architecture. (b) Latency comparison w/ bare-metal PRAM.

Fig. 2: PMEM DIMM's internal architecture and latency variation analysis.



(a) PMEM access. (b) Ex: linked list.

Fig. 3: Persistent management.

mainly performs 256B-sized read-and-modify operations, while DRAM is used for address translation and buffering the request as 4KB to take the benefit of parallelism. There is no official document to justify why 4KB operation is a matter, but we conjecture that such 4KB-sized buffering helps to reduce metadata sizes such as mapping table and wear-level counts as well as to provide compatibility with sector mode (block storage) operations. All these internal components are managed by *PMEM firmware*, similar to modern SSDs. Even though the current configuration of PMEM can satisfy diverse demands, the complicated internal architecture and combination of multiple controllers/firmware unfortunately introduce varying and non-deterministic latency.

Latency variation. As there is no publicly available PMEM to freely modify, we build a hardware prototype that can compare PMEM performance at DIMM-level and bank-level by using real 2x nm crosspoint bare-metal PRAMs. We integrate an emulation module [19] atop the PRAMs for the DIMM-level performance analysis. Figure 2b shows read and write latency variation with random accesses. For better comparison, we also include DRAM read/write latency in the analysis. Thanks to internal DRAM/SRAM buffers, writes on DIMM-level are shorter than those on bare-metal PRAMs by 2.3× ∼ 6.1×. However, DIMM-level reads take 2.9× longer time than bare-metal PRAM. Note that the internal buffering architecture can hide the long write latency of PRAM, thereby making PMEM writes even better than DRAM writes to some extent. However, even though the read latency of bare-metal PRAM is very similar to that of DRAM (only 1.1% difference), the reads cannot be directly served from the underlying PRAM because of the multi-buffer and PMEM firmware. Since the up-to-date data may exist in either SRAM or DRAM, it should pay the cost for lookup and each level of inclusive cache memory accesses. This imposes varying read and write latency and also makes DIMM-level performance non-deterministic.

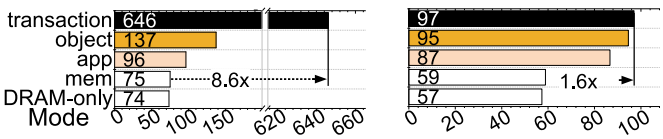
B. Persistent Management

PMEM with app-direct mode still needs runtime support to map its device file to a virtual address and to make PMEM DIMMs persistent from the processor viewpoint.

Data persistence control. Figure 3a shows how to use PMEM as persistent pool from an application angle. As Linux system exposes PMEM as device file (`/dev/XX`), it requires *direct access* (DAX) that directly maps data to application address space over a memory-mapped file (`mmap`) of Linux memory management (MM) [31, 32]. The address translation overhead of DAX is negligible since DAX simply calculates physical addresses (*PAs*) by adding the corresponding virtual addresses (*VAs*) to an offset in the target file. To enable data persistence, we need to use persistent object management library (*libpmemobj* [33]) of PMDK [34] on top of PMEM. The main idea of PMDK's data persistence is to manage application's data as objects and use persistent pointer (i.e., object ID), not VA of process. For example, Figure 3b shows how an application can persistently store a linked list by leveraging libpmemobj. Each entry of a linked list can be an object, and the VA of linked list's root object is same with the VA of memory-mapping by Linux's MM. If an additional list entry is created, libpmemobj calculates new object's offset from the root object address and uses it as an object ID. As libpmemobj uses offset-based pointers, the application should calculate VA per memory access, which introduces frequent software interventions and overhead. There is also a similar approach that stores VAs associated with their process to each object [35, 36]. Note that libpmemobj does not enforce a flush and synchronization in default, therefore, it requires explicitly declared transaction lines to make objects durable.

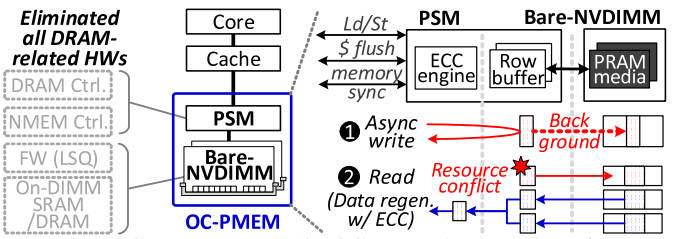
Performance overhead. In this analysis, we prepare four combinations of the PMEM solution and compare them with `DRAM-only` executing all applications on the local-node DRAM. `mem-mode` and `app-mode` setup PMEM as DRAM caching memory mode and app-direct mode with DAX, respectively. `object-mode` uses PMDK's libpmemobj atop `app-mode` to manage multiple address objects within a testing process, while `trans-mode` makes all changes durable by explicitly managing the object with transactions. Specifically, `trans-mode` wraps the operation code blocks (insert/delete) that access global and heap memories around PMDK's transaction APIs (`TX_BEGIN`/`TX_END`). We use Intel's platinum-class node that contains 1.5TB Optane PMEM with 190GB local-node DRAM and evaluate 17 HPC, SPEC, and In-memory DB workloads [37–41]. Figures 4a and 4b analyze the average latency and power, respectively. The results are only for memory subsystem power (excluding all other computing resources), measured by LIKWID [42].

Thanks to the DRAM caching and snarf, there is almost



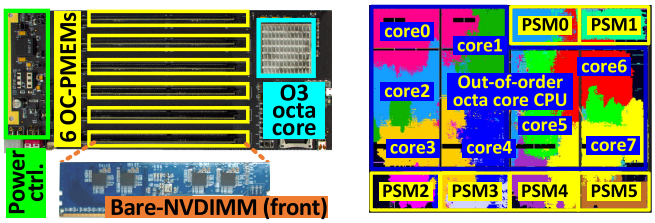
(a) Data:exec time(s). (b) Data:power(W).

Fig. 4: Performance of persistence control.



(a) Our OC-PMEM. (b) OC-PMEM's I/F and services.

Fig. 5: Overview of LightPC's hardware subsystem.



(a) System prototype. (b) Implementation view.

Fig. 6: Customized system prototype of OC-PMEM.

no difference between `DRAM-only` and `mem-mode` (under only 1.3%). Even though DAX has negligible overhead in the address translation at the application-level, `app-mode` still makes the execution time and power 28% longer and 47% higher than `mem-mode`, respectively, on average. This is because it requires extra internal DRAM/buffer lookup and device-level address translation for the load/store. When we insist on data persistence for multi-processes, `object-mode` increases the latency and power consumption by 1.8× and 1.6×, respectively. `object-mode` requires a set of initializations for the object root and header as well as runtime involvement with multiple objects' address translation. If ones make the data persistence durable, `trans-mode` further increases the application latency, compared to DRAM-only by 8.7×, on average. For the transaction, libpmemobj uses `pmem_persist()` to flush caches and synchronize memory. Since the cached data and their PAs are prohibited to access from users, it enforces the CPU cache controllers iteratively visit the cache blocks for the VA range given by `pmem_persist()`.

III. Lightweight Persistence-Centric Platform

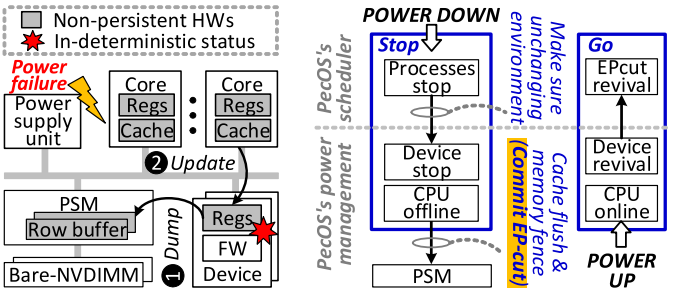
A. Hardware Subsystem Overview

To remove the overhead imposed by internal buffers, PMEM firmware, and device file-based operations, we expose all bare-metal PRAM devices to the computing complex while leaving minimum hardware logic such as parallelism and tail latency management in the controller side. This new system design, called *open-channel persistent memory* (OC-PMEM), can eliminate all DRAM-related hardware from the memory path such that it keeps volatile resources as small as the OS can quickly turn their states to be persistent. To this end, as shown in Figure 5a, OC-PMEM implements two major components: i) *persistent support module* (PSM) and i) *bare-metal PRAM DIMM channel*, referred to as Bare-NVDIMM. PSM manages resource conflict and reliability issues, while Bare-NVDIMMs are designed toward increasing parallelism to perform cache flushes as soon as possible. All running processes can issue memory requests to OC-PMEM through PSM, just like a traditional memory. In other words, the applications and OS kernel directly run on PMEM, which makes their stack, heap, and code regions almost persistent during the executions.

Basic philosophy. The biggest challenge to build DRAM-replaceable non-volatile memory (NVM) subsystem is to catch up with the read performance rather than write performance of DRAM. There are many studies to hide the NVM's write latency by allowing asynchronous operations with employment of intermediate high-performance memory for buffering, caching, and aggregation [16, 43]. However, the read latency cannot be easily shortened using such approaches when we aim to use NVMs as working memory; Unlike writes, CPU needs to bring the data before it operates, which makes asynchronous reads impractical. Read-ahead or prefetching may help, but the memory accesses should be fully sequential or speculative.

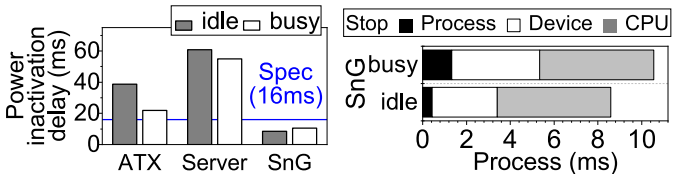
As shown in Figure 5b, PSM serves requests directly from Bare-NVDIMM to take advantage of high read speed of low-level PRAM devices (unlike PMEM). PSM applies the same strategy for writes but uses a simple row buffer for each PRAM device, which is widely used for DRAM [44]. The row buffer is different from the system-level buffering/caching schemes, as *it only removes the conflict latency* imposed by multiple writes targeting a specific region. With assistance of our memory subsystem, CPU waits for write completion only if OS faces a power failure. In our architecture, the writes are normally served as asynchronous, and their latency can be tolerable. The issue of our direct service is read targeting the location of OC-PMEM where writes are already performing. Since it only has a single row buffer for each PRAM device, the processor can read the place where the row buffer is closed, which can be observed in concurrent memory accesses. To address this, PSM serves the read requests by regenerating the target data from ECC rather than waiting to complete writes issued previously. In addition to conventional read/write interfaces, PSM exposes cache dump and memory synchronization interfaces, each flushing entire cache spaces and row buffers, respectively.

Prototype and baseline performance. While simulations are a powerful tool to validate the architectural concepts and designs, one of the goals of LightPC is to explore a practical design for power-failure free computing and make it real. Unfortunately, industry intellectual properties of memory controllers (integrated in CPU) are prohibited from accessing and modifying. Thus, we integrate PSM into a RISC-V based out-of-order octa-core CPU [45], and connect Bare-NVDIMM to the PSM in our customized system prototype. Unlike PMEM, which uses an asynchronous interface (DDR-T), we connect Bare-NVDIMM through conventional synchronous links (DDR), considering its deterministic latency characteristics (exposed to PSM). Figures 6a and 6b show the system prototype and the implementation view of OC-PMEM enabled CPU in a 28 *nm* FPGA [46], respectively. We use six Bare-NVDIMM and PSM to compose OC-PMEM (Figure 6a), and the eight cores are directly connected to PSM via a system memory bus without



(a) Need of PecOS. (b) PecOS's Stop-and-Go.

Fig. 7: Overview of LightPC's software subsystem.



(a) Holdup time. (b) SnG latency breakdown.

Fig. 8: Validation of PecOS's SnG.

employment of DRAM-related components (Figure 6b). The "baseline" performance of our OC-PMEM enabled system is only 12% different from that of a non-persistent "DRAM-only system", on average. We will give details of our implementation and comprehensive analysis in Sections V and VI.

B. Software Subsystem Overview

While OC-PMEM offers data persistence and allows the host to use the conventional memory interface, there are several non-persistent states, managed by cache and registers on multiple cores (Figure 7a). Our PSM may also hold outstanding requests, which are not yet completely written to the Bare-NVDIMMs. To address this, we introduce a *persistence-centric OS* (PecOS) that provides a *single execution persistence cut* (EP-cut) where the system safely re-executes it without losing contents. The single EP-cut can remove unnecessary cache flushes and memory synchronization while offering execution persistence. However, it is challenging to make all information of the processes running on multiple cores persistent in a single shot. For example, even though we successfully dump all CPU registers and flush all outstanding I/O requests including memory, caches, and peripherals to OC-PMEM, certain processes can further change the device states before completely running out the power inactivation delay time. Note that a sleeping process can be scheduled in a brief space of time, thereby making the machine state non-deterministic. In addition, we should quickly make the states of processes running on multiple cores be persistent in an all synchronized and coherent manner, such that they can be re-executed from the EP-cut.

Stop-and-Go. To address these, PecOS employs *Stop-and-Go* (SnG), which is triggered by any power event signal and turns non-persistent states to persistent within a power inactivation delay time. As shown in Figure 7b, all procedures to generate the EP-cut are called *Stop*, and processing from the point where the system should restart, referred to as *Go*. When a power interrupt arises, SnG makes all necessary volatile runtime persistent and confirms the corresponding connected devices stopped in *a power hold-up time* (the duration between when the interrupt arises and the point before the power rails fall out of specification to 95% of their nominal value). To this end, SnG first stops all processes and lockdown them across multiple cores. During this phase, SnG visits all sleeping processes and wakes them up by considering CPU load balancing. As SnG is bounded in the core, invoked by the power interrupt, called *master*, it gives *inter-process interrupt* (IPI) to other cores (*workers*) assigned to handle the waken process. SnG keeps iterating this process until there is no remaining sleeping process. In parallel, SnG's interrupt handler of such workers reschedules the just-waken process to their run queue. During this rescheduling, SnG just makes the target process yield all the time, such that it cannot be scheduled anymore. Note that, since all the cores will have similar numbers of tasks, including the running and waken processes, stopping the tasks can be achieved as much as the system can do in a balanced manner.

Once it ensures an unchanging environment for the multiple cores, SnG stops necessary devices and peripherals by collaborating with the corresponding drivers. SnG then dumps each core's volatile states to a designated space of OC-PMEM. Since some CPU registers can be invisible to even kernel such as IPI, power-down, and security registers, SnG jumps to the system bootloader and stores all the registers to the OC-PMEM. The master, in parallel, sends IPI for each worker to make them offline one by one. In this time, each core dumps caches and is suspended till the cache flushes complete. Lastly, the system bootloader flushes the master's caches and performs memory synchronization such that there are no outstanding requests before making all cores offline.

When the power is recovered, SnG examines whether the system initialization request is related to a cold boot or a power recovering case. If it is the latter case, SnG loads system contexts from OC-PMEM and re-executes all processes from the EP-cut. We will explain SnG's details in Section IV.

Offlining speed. We implement PecOS based on Linux 4.19 and validate SnG on our OC-PMEM system prototype. We test the actual power hold-up time of a standard ATX power supply unit (PSU) [47] and a Dell's server-class PSU [48] and the results are shown in Figure 8a. For this evaluation, we prepare two different configurations, i) busy and ii) idle. While the busy configuration makes the processor fully utilized by occupying the processor with heavy threads while the idle is the normal system that runs kernel and shell applications.

We observe that the standard ATX and server PSUs offer $22ms$ and $55ms$ even in cases where the processor is fully utilized, which is longer than the hold-up time that ATX specification declares ($16ms$). Even though there is enough room for SnG, SnG is implemented to make the target system persistent even in the worst case ($16ms$) by obeying the standard time that ATX documented. Figure 8b decomposes the SnG latency into process stop, device stop, and offline procedures. Everything for busy and idle systems is in the range of 8.6∼10.5 ms, which are 46% and 34% shorter than the worst-case power

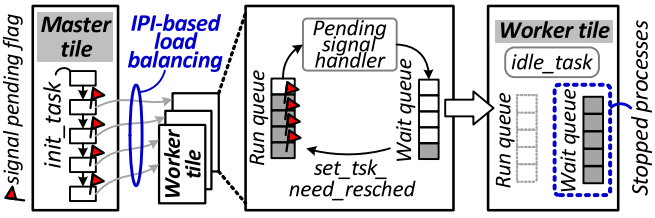


Fig. 9: Details of Drive-to-Idle.

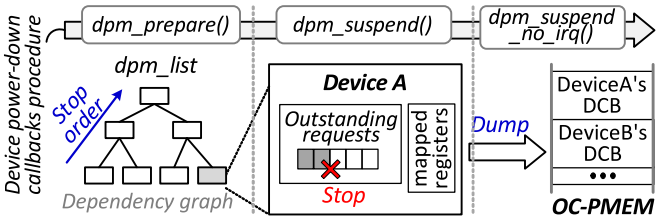


Fig. 10: Details of stopping devices in Auto-Stop.

hold-up time. SnG validation is also performed on the prototype that we used in Section III-A. Note that the number of the busy system's user and kernel processes is around 72 and 48, respectively (in total 120 processes). In addition, the current prototype includes all default device driver packages, which reflect an excessive computing and complex system scenarios.

IV. Details of Persistence-Centric OS

PecOS's EP-cut is given by two different phases of Stop, i) *Drive-to-Idle* and ii) *Auto-Stop*. While Drive-to-Idle makes the execution environment immutable, Auto-Stop dumps device states to the OC-PMEM, cleans up its cache and row buffer, and completely powers down all the processor cores.

A. Drive-to-Idle

All runtimes and execution states are on OC-PMEM, and only a part of them are in CPU caches and row buffers in our architecture. However, execution persistence cannot be given by just flushing the cache with a memory fence. Although SnG does not need checkpoint or system images, some processes can update the memory states even after the cache flushes. This makes the volatile and non-volatile incoherent as well as introduces execution disparity in all running processes on the system. Further, user processes can interfere with the devices that the system needs to suspend even after the cache flush and memory fence. For example, if a user process running on the second core accesses the devices that SnG is suspending, it makes the system non-deterministic.

To address these challenges, SnG's Drive-to-Idle makes sure that no process will further change before drawing the EP-cut. When the power event signal triggers SnG through an interrupt handler, the core first seizing such the signal becomes master. Drive-to-Idle running on the master sets an atomic system-wide persistent flag and then traverses alive process control blocks, PCBs (`task_struct`), derived from the init process (`init_task`). During this time, as shown in Figure 9, Drive-to-Idle sets a mask (`TIF_SIGPENDING`) for user processes while assigning sleep tasks across different cores in a balanced way. The master then issues IPI to each worker such that the master-side process examination can work in parallel with driving the just-waken tasks to idle. For the user processes, the worker gives a fake signal to them such that they can handle all remaining signals from the kernel-mode stack using entry assembler (`entry.S`). On the other hand, Drive-to-Idle executes other sleep tasks to handle pending work, but it makes them context switch out as soon as possible from their associated run queue by referring `set_tsk_need_resched()`. Whenever a process is switched out, Drive-to-Idle makes the process uninterruptible (`TASK_UNINTERRUPTIBLE`) and removes it from the run queue such that the process cannot further have a change. Once Drive-to-Idle removes all tasks from the run queue, it replaces each worker's running process with an idle task. In the meantime, the architectural states per process (including all thread's registers) are stored on PCB.

Note that these Drive-to-Idle procedures are all performed in parallel across different cores by carefully considering load balancing. Once the master is ready to place the idle task to its run queue, Drive-to-Idle signals workers and synchronizes all the cores to be in idle. Drive-to-Idle has no cache flush and memory fence operations; thereby the latency to be in idle (process stop) accounts for only 12% of the total Stop latency, as we analyzed in Figure 8b.

B. Auto-Stop.

Stopping devices. After Drive-to-Idle, Auto-Stop disables devices/peripherals, and it stores necessary information into device control block (*DCB*), as shown in Figure 10. Since the device configurations vary based on different environments, we leverage a standard *device power management* (`dpm`) mechanism [49] while manually handling the peripherals such as SPI and GPIO. Auto-Stop at the master visits each device driver, listed up in `dpm_list`, and it calls the registered callback functions that device driver implemented. The callbacks of `dpm` consist of a series of power down steps; the first callback `dpm_prepare()` that prevents device probe from the system-level. `dpm_suspend()` stops the outstanding I/O requests, disables interrupts, and powers down the device. After quiescing the target, it allows to store device states by having `dpm_suspend_noirq()`. As there may be dependency among devices, SnG calls them in the order that `dpm` regulated. SnG then reads peripheral-related memory regions, which are not physically located to OC-PMEM, but memory mapped. Auto-Stop also writes the peripheral information to DCB and flushes the master core's caches. Note that since this device stop requires visiting all device drivers and handling `dpm` callbacks, it is a time-consuming task whose latency is longer than Drive-to-Idle. As shown in Figure 8b, the device stop processes take 38% of the end-to-end SnG latency when all cores are busy.

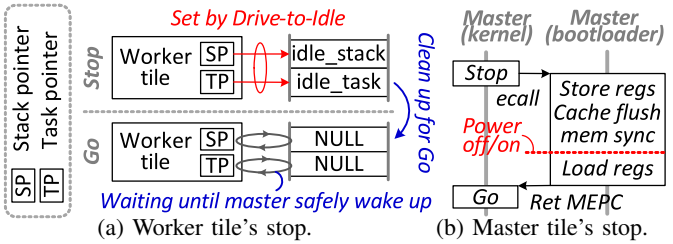
Drawing the execution persistence cut. When PecOS's Go re-executes all processes across different cores, PecOS should appropriately control the operation sequence of the cores. It also needs to know where the cores can re-run with which context. Specifically, in the cases where the power is recovered, all of the cores are ready to run by referring empty

execution pointers such as a kernel task pointer and stack pointer (i.e., `__cpu_up_task/stack_pointer`). However, PecOS's Drive-to-Idle placed them with the addresses associated with the per-core idle process (Figure 11a). Since all the data are persistent in LightPC, the master can lose the control of operation sequence when Go takes over the EP-cut. Thus, Auto-Stop cleans these kernel pointers such that all the cores can be synchronized after the system is recovered. Each core's Auto-Stop then dumps the cache and confirms pending memory requests (of PSM) are completed by issuing a memory fence request. In the last process of power down (from the worker's viewpoint), Auto-Stop informs the master that it is ready to be offline and makes sure that all workers can be safely powered off. Since there is a set of registers prohibited to access from kernel (cf. Section III-B), exception call is required before the master becomes completely offline. When the master receives the offline reports from all workers, Auto-Stop raises a system-level exception to switch the context from kernel to bootloader (Figure 11b). From the bootloader side, Auto-Stop stores such registers to OC-PMEM's reserved area in a form of bootloader control block (*BCB*). To finalize the EP-cut drawing, Auto-Stop also records the return address where Go will re-execute the system, called *machine exception program counter* (MEPC) to BCB. When the system is powered on again, it is required to check whether such the system booting is related to SnG or a cold boot. Auto-Stop clears the system-wide persistent flag (made by Drive-to-Idle) and stores a commit at this final stage to BCB with cache dump and memory synchronization. As this phase is performed per core and includes multiple cache/memory operations, it consumes most of SnG latency.

C. Go Implementation

Go implements two-phase operations; one is to synchronize the execution sequence of multiple cores, and another is to re-execute stopped processes with device revivals. When the power is recovered, Go is loaded as the bootloader and checks the Stop commit. If commit exists, Go restores BCB including bootloader- and kernel-related registers into the master. Otherwise, it passes the system control to kernel's entry point (`start_kernel`). Go also configures the interrupt handlers and boosts the master's execution mode to the highest-level such that the master can manage all other workers. To recover the system, the master's Go then powers up all workers and reconfigures all their registers before beginning the stopped processes. By this juncture, the workers do not execute any process, but wait for the task assignment by referring `__cpu_up_task/stack_pointer`. The master's Go places an idle process on OC-PMEM to each worker's kernel task pointers and gives IPI to them one by one. Go then jumps to kernel by referring MEPC (restored from BCB) that indicates the EP-cut where kernel-side Go should re-execute.

Then, Go at the master revives the devices by referring `dpm` callbacks in the inverse order of device suspension. `dpm_resume_noirq()` restores the device states from OC-PMEM and enables the corresponding device driver to receive interrupts. `dpm_resume()` and `dpm_complete()` allow



(a) Worker tile's stop. (b) Master tile's stop.

Fig. 11: Details of EP-cut drawing procedure.

the target devices to recover the contexts or reinitialize them (if it needs). In addition, Go restores the MMIO regions by reading the device contexts from OC-PMEM. All master and workers prepare the ready-to-schedule status by restoring the virtual memory space and flushing TLB. Finally, Go schedules other kernel process tasks in first and then user-level process tasks by checking up each core's wait queue and changing `TASK_UNINTERRUPTIBLE` to `TASK_NORMAL`.

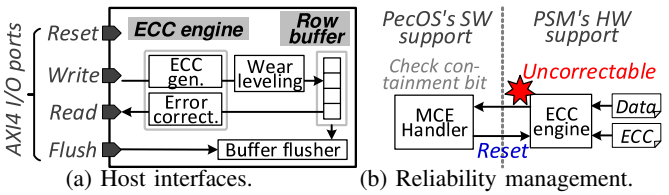
Note that, since PCBs that SnG stored by Drive-to-Idle contain all execution environment and registers, including page table directory pointer, it can restore them and load the virtual address space to MMU per process when the kernel scheduler makes the process tasks run; the processes can re-execute from the exact point where the EP-cut indicates.

V. Open-Channel PMEM Implementation

A. Persistent Support Module

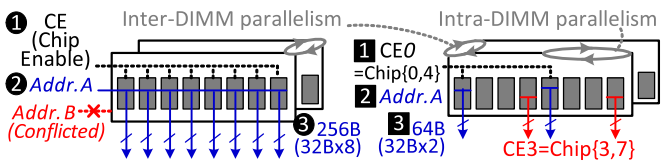
As shown in Figure 12a, OC-PMEM's PSM exposes write, read, flush, and reset ports (to processor), which are straightforward to be implemented in a conventional memory bus or crossbar switch. In this study, PSM is integrated to the processor complex over advanced extensible interface (AXI4 [50]) as our prototype is implemented based on RISC-V architecture, but it can be integrated into other typical front-side buses such as HyperTransport [51] or direct media interface [52]. While OC-PMEM is geared towards making PMEM lightweight, it is required to manage the shortcoming of long writes and reliability. PSM couples error management and resource conflict resolution with Bare-NVDIMMs.

Reliability management. The requested data are directly transferred between the processor caches and Bare-NVDIMM without internal SRAM/DRAM cache, but for writes, PSM's ECC encodes the data and stores them along with the original data. PSM implements a simple, but efficient wear-leveling algorithm, inspired by Start-Gap [53], which is designed for PRAM-based main memory systems. This algorithm keeps the number of writes served for the entire OC-PMEM ranges, and if it is greater than a threshold, the wear-leveler updates the address space by shifting a 64 byte block in company with a static randomizer. For reads, PSM's ECC checks the data correctness and returns the data with an error containment bit if it is corrupted (Figure 12b). If the containment bit is detected, the host raises *machine exception error* (MCE). Based on the system demand, the MCE handler can be implemented in the various ways, which is our future work. The current version of LightPC is to reset OC-PMEM and reinitializes the system for



(a) Host interfaces. (b) Reliability management.

Fig. 12: OC-PMEM's persistent support module.



(a) Conventional DRAM-like. (b) Our Bare-NVDIMM.

Fig. 13: Comparison of DIMM layout.

a cold boot by interacting with the reset port, which is used for wiping out all the memory spaces. Since cache flushes and memory fence are mapped to the flush port, PSM refers to the flush port per memory service. If there is, PSM blocks incoming requests and makes sure that all pending requests are served from Bare-NVDIMMs, which guarantees that there is no early-return request on the row buffer.

Conflict management. PSM implements two key components to address the latency issue caused by writes. Specifically, in contrast to DRAM, the PRAM writes are slower than its reads by 4∼8× at the processor-side viewpoint [16, 54, 55]. Since PecOS waits for the write completion only if there is a cache flush or memory fence issued to OC-PMEM, the write latency can be mostly tolerable. However, the long write latency still matters if i) writes are issued to a specific region and ii) a write blocks following read services. Note that the reason why PRAM writes require long latency is caused by cooling off PRAM's thermal core [56]. Thus, PSM does not need to wait for the completion if we preserve the time as long as the PRAM core is in a stable state (i.e., *early-return*). However, overwrites and read-after-writes prevent PSM from ensuring the cooling time. To address the first case, PSM employs a row buffer for each PRAM device, which is in this work implemented by a Block RAM (BRAM) FPGA intellectual property. The row buffer is assigned to a page that the processor has just requested, and if there are following writes to the same page, they can be just aggregated by the row buffer and served in the next. Note that the second case is more problematic as most applications exhibit more reads than writes, and such read-after-write operations make the *early-return writes* mostly useless. To address this, PSM reads other media along with ECC except for the target and serves the requested data by *reconstructing* it from other media and ECC. This allows PSM to perform reads/writes as no-blocking services with its best. Specifically, we use an XOR-based ECC to minimize the en/decryption overhead. PSM allocates XOR gates for each device as many as its input size (32B), and writes the each with XORing two device inputs half and half in a pipeline manner. In the meantime, if there is a conflict half, the read data can be regenerated by those parallel XOR gates, simultaneously. While the XOR-based ECC takes 2.5% area of LightPC CPU, its en/decryption can be performed in a cycle (as it is fully combinational logic) and it does not need metadata to figure out corresponding media location (statically mapped). In addition, it is used for correcting errors incurred by large granularity faults along with wear-leveling.

B. Bare-Metal PMEM Channels

As shown in Figure 13a, one can design a rank of Bare-DIMM, similar to DRAM. In this case, the PRAM devices (conventionally, eight) within a rank are all connected through a single *chip enable* (CE). Since the DRAM per device (bank) input granularity is 8B, all the DRAMs in the rank can serve 64B cacheline service in parallel [57]. This *DRAM-like NVDIMM* design well operates with DRAMs as a 64B-sized cacheline request is served by a rank (8B*8), and if there are further requests, they can be served by different ranks, called inter-DIMM parallelism. While we conjecture that the emerging Optane DIMMs adopt DRAM-like NVDIMM design to improve the degree of memory parallelism, unfortunately, they cannot efficiently handle a PRAM-based working memory. Since the input granularity of PRAM per device is bigger than DRAM (32B) [58], if we implement the DRAM-like channel for Bare-NVDIMM, the default access size becomes 256B (the same with that of PMEM DIMM). It thus requires read-and-modify to bridge disparity caused by different access granularities between a cacheline and a DIMM. As the DRAM-like channel needs to enable all PRAM devices per rank, a 64B cacheline-sized request wastes many PRAM resources per service, thereby making more incoming requests suspended.

Considering one of our goals (removing volatile resources as much as possible), we implement Bare-NVDIMM with a dual-channel design, that groups every two PRAM devices and shares a CE per group. In contrast to the DRAM-line Bare-NVDIMM design, our Bare-NVDIMM is well harmonized with the last-line cache operations. Specifically, as shown in Figure 13b, a 64B cacheline-sized request is right away served by a dual-channeled PRAM (32B*2) while remaining other PRAM devices being affordable to serve incoming memory requests. Thus, sequential accesses can take the benefits on interleaving requests across PRAM devices in a rank (*intra-DIMM parallelism*), while it keeps memory accesses across the rank boundary (random accesses) being interleaved similar to the DRAM-like DIMM design.

VI. Evaluation

Methodologies. We configure the memory subsystem of our prototype (Section III-A) into three: i) *LegacyPC*, ii) *LightPC-B*, and iii) *LightPC*. These three options have the same computing complex with OC-PMEM, but LegacyPC additionally employs DRAM as a hybrid design. LegacyPC is managed by Linux 4.19, which maintains all processes and data structures in DRAM. As the baseline of our implementation, LightPC-B places all the processes and data on OC-PMEM, but it

CPU configuration			
CPU	Freq(GHz)	I$,D$	
8 RV64 cores 7-stage O3	0.4(FPGA) 1.6 (ASIC)	16KB	
Bare-NVDIMM (vs. DRAM)			
#DIMM	Capacity	Read latency	Write latency
6	2x	1.1x	4.1x [61]

TABLE I: Configurations.

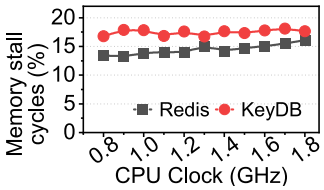


Fig. 14: CPU stall analysis.

handles read-after-writes just like what conventional memory controllers do. LightPC, on the other hand, implements all advanced PRAM management schemes such as early-return writes and data reconstructions to realize non-blocking services (cf. Section V-A). Note that, although LightPC's PRAM address space is 2× greater than that of LegacyPC's working memory (DRAM), for better comparison, we configure all Linux and applications to run with LegacyPC's memory space (i.e., no paging and swap memory caused by processes).

For execution persistence analysis, we implement four orthogonal persistence mechanisms: i) SnG, ii) system images, and iii) application-level checkpoint-restarts and iv) system-level checkpoint-restarts. SnG is implemented with LightPC-B and LightPC, whereas other three mechanisms are implemented in LegacyPC, referred to as *SysPC*, *A-CheckPC*, and *S-CheckPC*, respectively. SysPC stores non-persistent data and execution states into OC-PMEM in cases where there is a sleep signal. A-CheckPC that we implemented based on distributed multi-threaded HPC checkpointing [59] selectively stores stack and heap variables at the end of each function, whereas S-CheckPC is implemented based on Berkeley lab HPC checkpoint/restart (BLCR [60]) that periodically dumps the target thread virtual memory structure at kernel-level (per second). Compared to SysPC, A-CheckPC and S-CheckPC are more tolerable for a power failure. The important characteristics of the aforementioned platforms are explained in Table I.

Benchmark, power, and crash control. We select five HPC applications; two cryptographic algorithms (AES/SHA) for HPC key infrastructure [37], and three for proxy application estimations on parallel programing (miniFE/SNAP) and a parallel algebraic multigrid solver (AMG) [38–40]. We also select eight data-intensive applications from SPEC2006 [41],

Type	Category	Workload	Memory read (#)	Memory write (#)	#Read/ #Write	# Row buffer hit	D$		Multi thread
							read hit	write hit	
Computing intensive	Crypto	AES	21.7M	4.5M	4.8	1	99.5%	98.9%	
		SHA512	6.3M	438K	14	1	99.9%	99.9%	
	HPC	miniFE	419M	37.3M	11	3.9K	93.3%	99.4%	✓
		AMG	513M	46.7M	11	116K	84.1%	89.8%	✓
		SNAP	370M	137M	2.7	54K	97.9%	99.0%	✓
	SPEC CPU2006	perlbench	239M	38.9M	6.1	892	80.2%	81.3%	
		bzip2	123M	47.2M	2.6	774	94.6%	54.4%	
		gcc	360M	81.3M	4.4	70K	99.0%	98.4%	
		mcf	578M	1.7M	345	10K	93.4%	95.5%	
		astar	789M	296M	2.7	20K	96.2%	98.7%	
		cactusADM	428M	36.8M	12	9.1K	96.1%	94.1%	
		dealII	352M	26.7M	13	229K	75.8%	97.5%	
		wrf	345M	80.1M	4.3	1.2K	96.2%	94.2%	
Memory intensive	In-memory DB	Redis	377M	60.4M	6.2	37K	97.9%	99.1%	✓
		KeyDB	195M	75.7M	2.6	51K	97.7%	99.0%	✓
		Memcached	354M	57.3M	6.2	12K	95.3%	98.5%	✓
		SQLite	187M	14.9M	13	126	78.1%	98.4%	✓

TABLE II: Benchmark characterization.

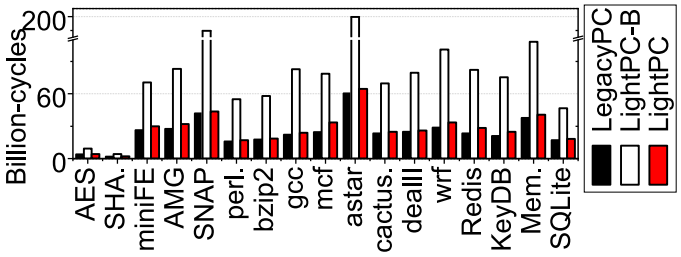


Fig. 15: Analysis of in-memory execution latency.

and evaluate memory-intensive applications such as Redis [62], KeyDB [63], Memcached [64], and SQLite [65].

To make the applications work upon our hardware prototype, we port and reimplement them based on RISC-V ISA. We execute HPC and In-memory DB workloads with multithreading, while Crypto and SPEC are evaluated with a single thread because of their benchmark characteristics. Note that all the workloads are executed upon our system already running tens of kernel threads. The important behaviors, such as the ratio of load/store and memory access behaviors, are analyzed in Table II. When power is turned off, most FPGAs lose their hardware programming file (i.e., bitfile). Instead of use of the bitfile downloaded, our custom board stores memory configuration (`.mcs`) to on-board ROM and automatically restores the hardware programming in bootup. For the tests of power failure, we physically remove AC power sources from the power supply.

Even though the FPGA frequency is slow (400MHz), our Synopsys frontend and backend simulations show that the prototype RTL of our octa out-of-core processor [66] can successfully operate at 1.6GHz. Note that the real-world applications are not a memory stress testing tool. The memory latency effects for the application are not diminished at the testing frequency. To make sure of this, we pick two memory-intensive applications and scale the frequency of a server Intel Xeon octa-core processor from 0.8 to 1.8 GHz. While the DRAM operates at 2.4GHz, user-level experiences on varying frequencies (in terms of memory stalls) show a similar trend.

A. Non-Persistent Computing Performance

The potential issue of using OC-PMEM as primary working memory can be performance degradation compared to a DRAM-only system. We analyze application latency and power/energy behaviors by comparing LightPC-B/LightPC and LegacyPC, which uses only DRAM for the working memory.

In-memory execution latency. Figure 15 compares the execution time of user-level applications for OC-PMEM only and DRAM only use-cases. While the read and write latency of PRAMs are slower than those of DRAM 1.1× and 4.1×, respectively, the completion time of each benchmark is not too much different from each other; overall, LightPC is slower than LegacyPC by only 12%, on average. We believe that there are two root causes. First, DRAM speed (frequency) itself is not strictly related to the application-level performance, but memory-level resource conflicts are a more important parameter to decide user experience, which is also observed by [67]. Second, the write penalty of PRAM is relatively

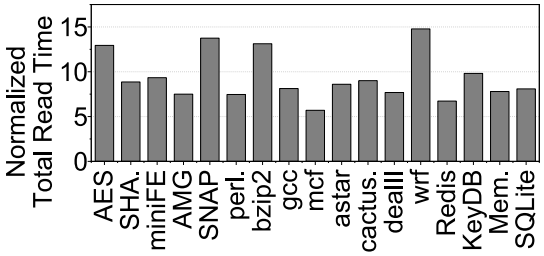


Fig. 16: LightPC-B's read latency.

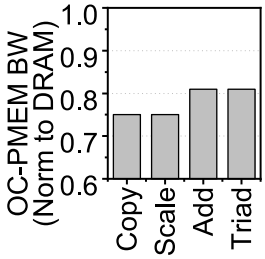


Fig. 17: Bandwidth.

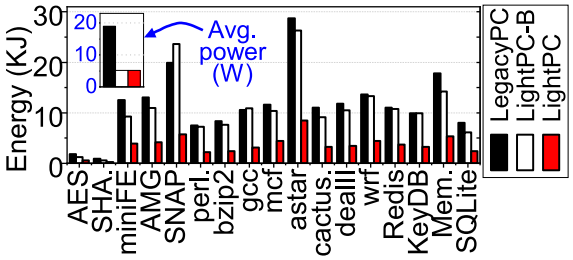


Fig. 18: Power and energy.

significant, but, most applications on the load-store architecture in nature exhibit more loads than stores. For the workloads we tested, the number of loads is 27× greater than that of stores, on average (Table II). Considering the benefits brought by LightPC (in terms of persistence, power and energy, which will be explained shortly), we believe that 12% slow application-level latency for in-memory execution is tolerable in many usages.

By comparing LightPC with its baseline (LightPC-B), we can estimate how much our PSM can improve the user-level experiences in details. As shown in the figure, LightPC exhibits 2.8× shorter average execution times across all the workloads that we tested. For the SNAP and astar applications, LightPC, in particular, outperforms its baseline by 4.1×, on average. This is because the number of memory write requests is larger than any other applications (more than a few million instructions), while the applications exhibit heavy reads (2.7 × more than the writes). In contrast, the performance improvement of LightPC is limited for SHA (SHA512). This is because, even though the read ratio of SHA is higher than that of the above two applications, SHA exhibits a small number of writes requests than all the applications that we tested.

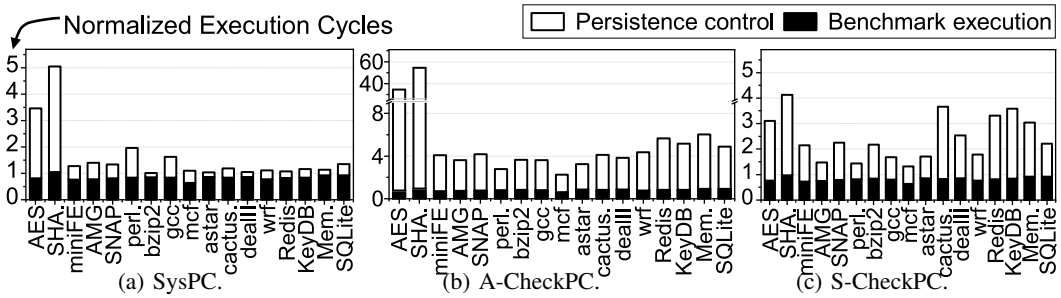
Removing head-of-line blocking. To dig deeper, we further analyze the performance improvement of PSM's non-blocking services at the memory level. Figure 16 normalizes the low-level read latency of LightPC-B for the entire application execution to that of LightPC. One can observe from this figure, the performance improvement of LightPC, compared to the baseline read latency, ranges from 7× to 14.8 ×. In particular, LightPC improves the memory-level read performance of the wrf (weather forecast model) by 14.8×, on average. Since the wrf recursively uses the prediction history for the next forecast prediction, it introduces many reads, heading to the places where the wrf wrote in the previous steps. This application behavior exhibits many head-of-line blocking issues, which can be addressed by LightPC. On the other hand, the mcf has relatively less performance improvement compared to other applications. Since the number of writes is significantly smaller than that of reads, mcf has fewer chances to read data that have been just written, compared to other workloads.

In general, 9× read latency reduction, on average, is the matter to improve user-level experiences across all the applications that we tested. Note that reads are more critical than writes as the following instructions should wait for the data arrivals, whereas the writes data can be asynchronously written to the underlying memory. The reason behind the big read latency difference between LightPC and LightPC-B is the long write latency being in services at the same memory location. Since LightPC-B employs no mechanism for non-blocking services, such read-after-write requests are all suspended, thereby leading to head-of-line blocking issues. In contrast, LightPC does not stall any instructions to for a write completion and directly serves the reads by constructing new data using the data of other side memory banks with ECC.

Potential memory bandwidth impacts. Considering a user focusing on bandwidth, we also evaluate a synthetic benchmark (STREAM [68]) designed to measure sustainable memory bandwidth for LegacyPC and LightPC. Figure 17 analyzes bandwidth on the target platform's memory hierarchy by normalizing the results of LightPC to those of LegacyPC. As shown in the figure, while the memory-level bandwidth behaviors of LightPC are a little bit worse than the characteristics of its application execution time, the bandwidth difference between LegacyPC and LightPC is still in a reasonable range (78%, on average). This is because STREAM synthetically moves the data from one to another location, which reduces the impact of caches and makes (uncached) writes exceed real-world workloads' reads. Specifically, their average cache hit ratio and read ratio are reduced by 5% and 94%, compared to those of real-world workloads that we tested (cf. Table reftab:bench). Note that, since Add and Triad perform element-wise operations on more than two arrays, they exhibit more reads than Copy and Scale (copying one array to another) thereby being closer to LegacyPC.

Power and energy. Figure 18 shows the power and energy values of the platforms we tested for the aforementioned in-memory executions; the power values are embedded in the left corner of the figure. Both LightPC and LightPC-B consume only 28% of the total system power, consumed by LegacyPC, on average. Specifically, LegacyPC consumes 18.9 Watts, whereas LightPC and LightPC-B consume only 5.3 Watts. This is because PRAM itself exhibits power consumption lower than DRAM, but more importantly, LightPC has no burden to manage DRAM refresh and system power.

While the power consumption behaviors are similar between LightPC and LightPC-B, LightPC-B only reduces the energy by 8.2% compared to LegacyPC. This is because read-after-writes and long write latency make application execution time longer (as we discussed) such that it loses the gain brought by OC-MEM. On the other hand, the energy efficiency of LightPC is overall 69% better than LegacyPC. Although the difference



(a) SysPC. (b) A-CheckPC. (c) S-CheckPC.

Fig. 19: Benchmark execution cycle overhead.

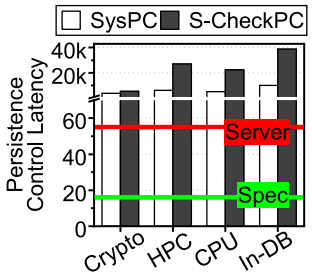


Fig. 20: Persistence control time of SysPC.

of execution completion time between LegacyPC and LightPC is in a reasonable range (12% on overage), the power consumption behaviors of LightPC outperform LegacyPC as high as 73% and offer an energy-efficient computing environment without a volatile buffer and external storage.

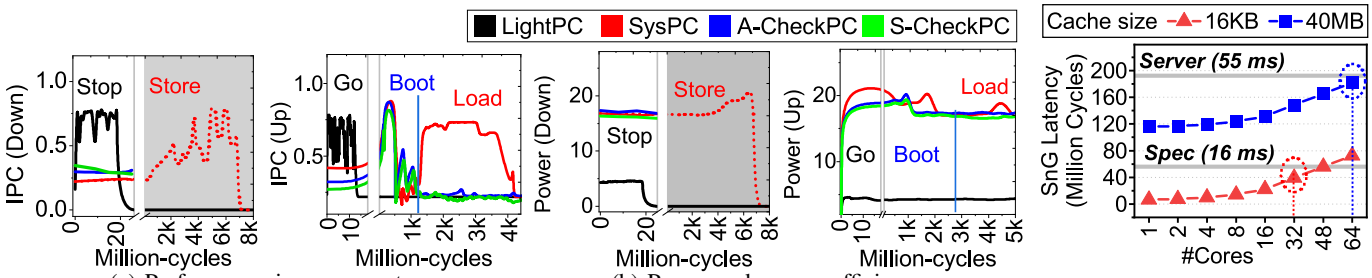
B. Performance of Persistent Computing

Figures 19a, 19b, and 19c show the performance of SysPC, A-CheckPC and S-CheckPC, normalized to that of LightPC, respectively, when executing the benchmarks with a power down. These figures also decompose the execution cycles into two, benchmark execution and persistence control. The execution latency of A-CheckPC for all the benchmark suites is over 134B cycles, while that of SysPC is only 60B cycles, on average. This is because, even though A-CheckPC selectively dumps stack and heap variables that each function uses, it frequently migrates the data from DRAM to OC-PMEM during the execution time. Thus, before the commit is completed for each checkpoint, the benchmark execution on A-CheckPC is generally stalled. Although the size of system images is much greater than A-CheckPC's ones, SysPC exhibits 5.5× shorter latency, on average. This is because SysPC dumps the system images only at once (at the power down), whereas A-CheckPC saves the checkpoint image at each function call. Note that the checkpoint granularity of our A-CheckPC is much coarser than other checkpoint-restarts such as adaptive dynamic checkpointing [3, 69, 70] that dumps data for each basic block, but A-CheckPC exhibits such a significant overhead. While S-CheckPC cannot control the execution persistence as much as users want to manage, its periodical data dump in a self-governing manner can alleviate the overhead of A-CheckPC's persistence control. As shown in Figure 19, S-CheckPC reduces the execution latency of A-CheckPC by 73%, on average by dumpling the application's virtual memory spaces via `vm_area_struct` to OC-PMEM automatically. Specifically, the benefits of S-CheckPC are emphasized on Crypto as it just simply dumps the application's virtual memory without understanding per-encryption/decryption contexts. Nevertheless, the latency of S-CheckPC is yet 52% worse than SysPC, on average. This is because SysPC stores the images back to OC-PMEM only if there is a power failure.

In contrast, when we consider persistence as the first-class citizen, the execution latency of LightPC becomes shorter than that of SysPC, A-CheckPC, S-CheckPC by 1.6×, 8.8×, and 2.4×, respectively. This is because PecOS's SnG takes account for only 0.3% of the total execution time, on average. Figure 20 shows flush latency for SysPC and S-CheckPC. The figure also includes residual stored energy on power loss (hold-up time) for ATX and Server that we measured (cf. Figure 8). SysPC takes 172× and 112× longer flush latency compared to the hold-up times of ATX/Server, respectively. S-CheckPC requires periodically dumping the virtual memory data into OC-PMEM, which also respectively exhibits 3.5× and 1.4× longer flush operations than the hold-up times of ATX/Server. In contrast, LightPC's Stop successfully flushes outstanding requests in 12.8 ms, which is shorter than the hold-up times of ATX/Server by 33% and 21%, respectively.

C. Consistence Control Efficiency and Scalability

Performance. Figure 21a shows the time series analysis of LightPC, SysPC and A/S-CheckPCs' dynamic IPC. It also decomposes each step of consistency control in terms of cycles (x-axis). LightPC's Stop consumes 19 million cycles (mc) at the power down, and requires 12.8 mc before the benchmark execution restarts from the persistence cut. Since the latency of LightPC's Stop is shorter than the worst case of power inactivation delay by 20%, on average, which makes LightPC successfully be in fully consistence. In contrast, SysPC requires 7 billion cycles (bc) to store system images to OC-PMEM, and even after power recovery, SysPC spends 4.2 bc to load the system images, which is 358× slower than LightPC's Go. A/S-CheckPCs do not require a time to prepare the system images as they periodically dump necessary information. Note that, because of periodic dumps, the performance behaviors at power down/up times are similar each other. However, since checkpoint processes of those two consume memory bandwidth with memory synchronizations/flushes, A/S-CheckPCs can give burdens for the benchmark execution. Since A/S-CheckPCs are unavailable for dumping kernel itself and machine mode registers, a cold reboot is unavoidable once power is recovered. Thus, they show the IPC spike right after the power recovery before the benchmark resumption. Note that the average IPC for power down preparation processes of SysPC, A-CheckPC, S-CheckPC and LightPC is 0.5, 0.23, 0.30, and 0.66, whereas their power-up recovery's IPC is 0.59, 0.23, 0.19, and 0.64, respectively. LightPC exhibits better IPC as it does not have memory dump-related thread pending, which makes LightPC's SnG faster than the power inactivation delay.



(a) Performance improvement.

(b) Power and energy efficiency.

Fig. 21: SnG analysis during power-down and power-up.

Fig. 22: Scalability analysis (worst-case).

Power/energy efficiency. Figure 21b shows dynamic system power values for the power-down preparation and power-up recovery processes. As considering the execution cycles (x-axis), this figure also can be used for energy analysis. SysPC consumes 20W to hibernate the system, and unfortunately it requires unwearying power (and 19.7J) to complete the system images dump even after the power inactivation delay (until 7 bc). In contrast, LightPC makes the cores and devices offline by 19 mc, which consume 4.5W and 53mJ for the completion of Stop. A/S-CheckPCs exhibit the power consumption behaviors similar to SysPC, but they can just turn off the system at the end of power inactivation time as each checkpoint can be used as a committed execution transaction. At the power recovery, both all persistence mechanisms exhibit similar power and energy behaviors as they must restart the system; as loading system images is lighter than a cold boot, SysPC consumes 2.7% less power than A/S-CheckPCs. In contrast, Go recovers all the processes by 12.8 mc with 4.4W, thereby consuming 52mJ.

Scalability analysis. Figure 22 analyzes the scalability that LightPC can support when we consider the worst-case configuration; having the maximum number of drivers in PecOS's kernel list, `dpm_list` (730) and making all cachelines fully dirty thereby flushing the entire cache. Since the current prototype cannot unfortunately accommodate more than 8 physical cores because of its FPGA die area limit, we estimate the SnG latency for the cases where there are more than 8 cores. To make an accurate estimation, we modify LightPC with different cache sizes, do hardware instrumentation for each core, and measure the worst-case latency of each core's offlining, cache flushing, and driver/process suspending from the current prototype. As shown in the figure, SnG can successfully stop the machine having 64 cores with 40MB cache within the server PSU's power holdup time (cf. 55ms, Section III-B). We believe that this type of manycore machine will leverage server PSU rather than the ATX documented value (16ms). However, if one regulates the machine to meet 16ms, SnG can stop upto 32 cores with 16KB cache in this worst-case scenario.

VII. Related Work

Non-volatile processor for energy harvesting. As the persistence guarantee is key for non-volatile computing, there exist many studies dealing with persistence control [23, 31, 71–75] and significant analyses on PRAM [76, 77]. Non-volatile processors (NVPs) [72–74] are proposed to make energy-harvesting systems power-variation tolerant [78]. NVPs periodically dump register files, pipeline latches, and program counter into FeRAM or replace all of them with FeRAM. NVPs are a simulation-based study to explore the design space to make low-power processor persistent. However, NVPs cannot make a system fully persistent. They do not consider heap/stack memory for their consistence control as the target system is in energy-harvesting environment.

Whole-system persistence. Whole-system persistence (WSP) proposes a concept of flash backed flush-on-fail [79], which flushes caches and DRAM data into the underlying flash by utilizing ultracapacitors. This dumping process is similar to hibernation, but it can be performed from DIMM-side controllers, which can take around long time (10 seconds at most). While the ultracapacitors may guarantee memory-level persistence, it has several constraints. First, the persistence with them can be crashed if there are continuous power failures at a certain period. This is because the ultracapacitors also require a reliable charge time, which is similar to the dump time (10 seconds). If a consecutive power failure arises during the charge time, state changes made by the charge time cannot be captured. Second, flash backed solution is not a scalable solution due to ultracapacitor scaling issues [80], and even its memory capacity is limited by DRAM; the capacity of WSP can be smaller than (or equal to) DRAM. WSP also requires a complex BIOS configuration and calibration processes. Thus, it is only performed single-threaded closed-loop tests on an emulation environment.

Enhanced asynchronous DRAM refresh. Recently, Intel proposed eADR [81] that flushes the cached data to PMEM when a power event signal triggers. This process is similar to a part of Stop (flushing operation at its end). However, it needs to ensure making the system states consistent/persistent, offlining device drivers and cores in a specific sequence (i.e., EP-cut), such that the system can restore all the process/device contexts and registers when the power is recovered. Note that, since eADR has a lack of control for consistent system states, cachelines can also change and be dirty while flushing their cache without EP-cut.

VIII. Discussion and Future Work

Write endurance and reliability. The write endurance of our PRAM devices (set and reset cycles) is in the range of

$10^6 \sim 10^9$, which is the same as the most prior PRAM studies reported [17, 82]. As this is an order of magnitude worse than DRAM, unfortunately, bare-NVDIMM itself cannot meet the same reliability level that DRAM offers. Thus, LightPC may not be the right design for a specialized purpose application that heavily writes data without reads (e.g., burst buffer). However, this work still stands for PRAM as working memory because of two reasons: i) *reads are more sensitive than writes* and ii) *PRAM has no writes on destructive reads and refresh*. As analyzed in Section VI, load instructions are in general observed much more than memory stores. We believe that most instruction sets in natural have more operands to read memory (inputs) than writes (outputs). At the device-level, reads are more frequently observed and become sensitive; several layers of on-chip storage (registers, L1/L2, etc.) can absorb writes of store instructions whereas reads mostly cannot. Note that DRAM should write the in/output data for both the reads and writes because of its destructive read characteristic [83, 84]. DRAM also must write the existing data back every 64 ms to avoid its data loss caused by leakage current. Since PRAM's writes are all free from destructive reads and refresh, a direct comparison using the write endurance may not be fair.

Note that there are also several studies expecting to increase the write endurance by $10^{12} \sim 10^{13}$, which is similar to or slightly worse than that of DRAM (e.g., $10^{14} \sim 10^{16}$ [17, 57, 85]). For example, [86] uses a thin metallic linear and confined pore cell structure to reduce the unavailability of set/reset switching thereby significantly increasing the endurance on writes. [87–89] also studies PRAM's write disturbance issues caused by thermal crosstalk [90, 91] and enhances the write endurance cycles by increasing thermal boundary resistance and separating physical cells [86, 92].

Error rates and coverage. In this work, we assume that our XOR-based ECC (XCC) can manage an uncorrectable bit error rate (UBER) lower than 10^{-19}, which is similar to UBER that most storage class memories assume to guarantee. Specifically, the assumption of this work is that all the PRAM modules in two Bare-NVDIMMs are not simultaneously dead at a given time. Considering the write endurance ranging from 10^6 to 10^9, there is a study to reveal that 8-bit error correction capability (per cacheline) is needed to support 10^{-19} UBER PRAMs [93]. While this ECC coverage requirement is stronger than the chipkill or double chipkill ECCs that most DRAMs use [94–96], the correction capability of LightPC's XCC is much higher than that of the 8-bit correction. As explained in Section V-A, XCC is capable of regenerating a conflict half the data of cachelines, which can be leveraged to recover 32B data per cacheline. To make Bare-NVDIMMs more robust, we can have a finer-granule, symbol-based ECC [97–100] in addition to XCC, but this can unfortunately increase Bare-NVDIMM access latency because of its en/decryption. In this future work approach, one may use the symbol-based ECC only in cases where two or more Bare-NVDIMMs are simultaneously dead, while XCC is used for the most in an attempt to take advantage of both ECC schemes.

Wear-leveling capability. LightPC leverages StartGap [53] which is a simple but effective wear-leveling method for PRAM-based working memory. The main idea behind such a wear-leveling method is to shuffle (shifting) the entire memory address space of Bare-NVDIMM rather than maintaining mapping information between logical and physical addresses. As explained in Section V-A, such address shuffling can be simply done by statically randomizing the addresses and shifting a 64-byte block in the address space for every given number of writes (100 in default). The wear-leveler's metadata registers are thus associated with only the start/gap offsets, write threshold counter, and static randomizer's seed. LightPC stores these registers (taking less than 64B per 4TB$\sim$6TB memory) at SnG's EP-cut with other volatile states. When the system power is recovered, the metadata as part of EP-cut's information is restored, making our wear-leveler available across system boots.

While such lightweight metadata and wear-leveling management are cost-effective, there is a limit. Suppose there is an abnormal application that keeps accessing a set of specific addresses without generating any other writes. In that case, our wear-leveler cannot evenly distribute writes across all different places of PRAMs. This is because shifting a 64-byte block only happens from the end of memory space (indicated by the gap register) one by one for each epoch. Even though [53] demonstrated that the static randomizer with StartGap's address shifting could offer 97% of the theoretical maximum lifetime, we believe that it yet is vulnerable to such a malicious access pattern. We consider periodically changing the seed register value in future work, thereby enhancing the endurance level.

IX. Conclusion

We proposed LightPC that guarantees both data and execution persistences by removing all DRAM-related hardware components and software stack from the memory hierarchy and persistent control, respectively. Our non-volatile computing board prototype, which implements multiple cores and a persistent subsystem in a 28 *nm* FPGA, offers user-level performance, similar to a DRAM-only system, while consuming 73% lower power and shortening latency of diverse benchmarks compared to checkpoint-restarts by 8.8×, on average.

X. Acknowledgement

The author thanks to Jason Lowe-Power for shepherding this paper. The author also thanks anonymous reviewers of ASPLOS'22, ASPLOS'21, SOSP'21, ATC'21, MICRO'20, and ISCA'20 for their constructive feedback. This work is mainly supported by MemRay grant (G01190170). This work is also in part supported by NRF's 2021R1A2C4001773, IITP's 2021-0-00524 & 2022-0-00117, KAIST start-up package (G01190015), and KAIST IDEC. The authors also thank Donghyun Gouk (CAMEL), MemRay, and Samsung for their sample donations and evaluation/technical support. This work is protected by one or more patents. Myoungsoo Jung is the corresponding author.

References

[1] S. Di, H. Guo, R. Gupta, E. R. Pershey, M. Snir, and F. Cappello, "Exploring properties and correlations of fatal events in a large-scale hpc system," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[2] Y. Yin, J. Wu, X. Zhou, L. Eeckhout, A. Qouneh, T. Li, and Z. Yu, "Copa: Highly cost-effective power back-up for green datacenters," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[3] M. Chtepen, F. H. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids," *IEEE Transactions on Parallel and Distributed Systems (TPDS'08)*, 2008.

[4] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[5] B. Nicolae and F. Cappello, "Blobcr: efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2011.

[6] R. Garg, G. Price, and G. Cooperman, "Mana for mpi: Mpi-agnostic network-agnostic transparent checkpointing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[7] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*, IEEE, 2013.

[8] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta, "Performance implications of periodic checkpointing on large-scale cluster systems," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, IEEE, 2005.

[9] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu, "Edo: improving read performance for scientific applications through elastic data organization," in *2011 IEEE International Conference on Cluster Computing (CLUSTER'11)*, IEEE, 2011.

[10] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, 2014.

[11] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS'16)*, IEEE, 2016.

[12] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'10)*, IEEE, 2010.

[13] Intel, "Optane DC Persistent Memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[14] N. Tanabe and T. Endo, "Exhaustive evaluation of memory-latency sensitivity on manycore processors with large cache," in *Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications (HP3C'18)*, 2018.

[15] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, 2010.

[16] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[17] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, 2010.

[18] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska, "Unexpected performance of intel® optane™ dc persistent memory," *IEEE Computer Architecture Letters*, 2020.

[19] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020.

[20] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, 2020.

[21] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *2019 USENIX Annual Technical Conference (USENIXATC 19)*, 2019.

[22] A. Demeri, W.-H. Kim, R. M. Krishnan, J. Kim, M. Ismail, and C. Min, "Poseidon: Safe, fast and scalable persistent memory allocator," in *Proceedings of the 21st International Middleware Conference*, 2020.

[23] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. Brito Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, ACM, 2017.

[24] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017.

[25] SiFive, "SiFive TileLink Specifica-

tion." https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[26] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[27] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[28] "Quick start guide part 1: Persistent memory provisioning introduction." https://software.intel.com/content/www/us/en/develop/articles/qsg-intro-to-provisioning-pmem.html.

[29] N. AbouGhazaleh, B. R. Childers, D. Mossé, and R. G. Melhem, "Near-memory caching for improved energy consumption," *IEEE Transactions on Computers*, 2007.

[30] B. Nale, R. K. Ramanujan, M. P. Swaminathan, T. Thomas, and T. Polepeddi, "Memory channel that supports near memory and far memory access," May 17 2016. US Patent 9,342,453.

[31] "Direct access for files." https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[32] R. Kateja, N. Beckmann, and G. R. Ganger, "Tvarak: software-managed hardware offload for redundancy in direct-access nvm storage," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020.

[33] Intel, "libpmemobj: A native transactional object store." https://pmem.io/pmdk/libpmemobj/.

[34] Intel, "Pmdk: Persistent memory development kit." http://www.pmem.io.

[35] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, *et al.*, "K42: building a complete operating system," *ACM SIGOPS Operating Systems Review*, 2006.

[36] D. Bittman, P. Alvaro, P. Mehra, D. D. Long, and E. L. Miller, "Twizzler: a data-centric os for non-volatile memory," in *2020 USENIX Annual Technical Conference (USENIXATC 20)*, 2020.

[37] J. Hiller, J. Amann, and O. Hohlfeld, "The boon and bane of cross-signing: Shedding light on a common practice in public key infrastructures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[38] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[39] R. J. Zerr and R. S. Baker, "Snap: Sn (discrete ordinates) application proxy," *Online: https://github. com/losalamos/SNAP. Accessed: Sep*, 2014.

[40] U. M. Yang *et al.*, "Boomeramg: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, 2002.

[41] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[42] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops (ICPP'10)*, IEEE, 2010.

[43] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[44] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2012.

[45] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.

[46] "Xilinx virtex7 fpga." https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html.

[47] S. flower, "Super flower SF-600R12A." https://www.super-flower.com.

[48] DELL, "DELL Power Supply 770-BCBD." https://www.dell.com/en-sg/work/shop/dell-mellanox-sb7800-sb7890-460-watt-power-supply/apd/770-bcbd/computer-chassis-components.

[49] "Device power management specification." https://elinux.org/Device_Power_Management_Specification.

[50] ARM, "AMBA AXI and ACE protocol specification." https://static.docs.arm.com/ihi0022/d/IHI0022D_amba_axi_protocol_spec.pdf.

[51] H. T. Consortium *et al.*, "Hypertransport i," *O Link Specification*, 2003.

[52] "Introduction to intel® architecture." https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf.

[53] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, IEEE, 2009.

[54] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, IEEE, 2010.

[55] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate

memory technologies," *ACM SIGARCH computer architecture news*, 2009.

[56] M. Le Gallo and A. Sebastian, "An overview of phase-change memory device physics," *Journal of Physics D: Applied Physics*, 2020.

[57] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *ACM SIGARCH computer architecture news*, 2009.

[58] K. Kim, S.-W. Lee, B. Moon, C. Park, and J.-Y. Hwang, "Ipl-p: In-page logging with pcram," *Proceedings of the VLDB Endowment*, 2011.

[59] W. R. Dieter and J. E. Lumpp Jr, "User-level checkpointing for linuxthreads programs.," in *USENIX Annual Technical Conference, FREENIX Track*, 2001.

[60] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, IOP Publishing, 2006.

[61] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, *et al.*, "A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth," in *2012 IEEE International Solid-State Circuits Conference*, IEEE, 2012.

[62] "Redis." https://redis.io/.

[63] "Keydb." https://keydb.dev/.

[64] "Memcached." https://memcached.org/.

[65] "Sqlite." https://www.sqlite.org/.

[66] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[67] S.-J. Cho, J. Ahn, H. Choi, and W. Sung, "Performance analysis of multi-bank dram with increased clock frequency," in *2012 IEEE International Symposium on Circuits and Systems*, IEEE, 2012.

[68] J. D. McCalpin, "Stream benchmark," *Link: www. cs. virginia. edu/stream/ref. html# what*, 1995.

[69] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[70] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, IEEE Computer Society, 2003.

[71] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*, ACM, 2011.

[72] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*, IEEE, 2015.

[73] F. Su, K. Ma, X. Li, T. Wu, Y. Liu, and V. Narayanan, "Nonvolatile processors: Why is it trending?," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, IEEE, 2017.

[74] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *2012 Proceedings of the ESSCIRC (ESSCIRC'12)*, IEEE, 2012.

[75] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, USENIX, 2016.

[76] D. Kim, S. Lee, J. Chung, D. H. Kim, D. H. Woo, S. Yoo, and S. Lee, "Hybrid dram/pram-based main memory for single-chip cpu/gpu," in *DAC Design Automation Conference 2012*, IEEE, 2012.

[77] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, 2009.

[78] V. Leonov, "Energy harvesting for self-powered wearable devices," in *Wearable Monitoring Systems*, Springer, 2011.

[79] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, ACM, 2012.

[80] B. K. Kim, S. Sy, A. Yu, and J. Zhang, "Electrochemical supercapacitors for energy storage and conversion," *Handbook of Clean Energy Systems*, 2015.

[81] "2021. intel optane persistent memory 200 series brief.." https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[82] H.-Y. Cheng, F. Carta, W.-C. Chien, H.-L. Lung, and M. J. BrightSky, "3d cross-point phase-change memory for storage-class memory," *Journal of Physics D: Applied Physics*, vol. 52, no. 47, p. 473002, 2019.

[83] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in dram for cache applications," *IEEE micro*, vol. 28, no. 6, pp. 47–56, 2008.

[84] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving dram latency with dynamic asymmetric subarray," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 255–266, IEEE, 2015.

[85] H. Aziza, M. Moreau, M. Fieback, M. Taouil, and S. Hamdioui, "An energy-efficient current-controlled write and read scheme for resistive rams (rrams)," *IEEE Access*, vol. 8, pp. 137263–137274, 2020.

[86] W. Kim, M. BrightSky, T. Masuda, N. Sosa, S. Kim, R. Bruce, F. Carta, G. Fraczak, H. Cheng, A. Ray, *et al.*,

"Ald-based confined pcm with a metallic liner toward unlimited endurance," in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 4–2, IEEE, 2016.

[87] S. Lee, M. Kim, G. Do, S. Kim, H. Lee, J. Sim, N. Park, S. Hong, Y. Jeon, K. Choi, *et al.*, "Programming disturbance and cell scaling in phase change memory: For up to 16nm based 4f 2 cell," in *2010 Symposium on VLSI Technology*, pp. 199–200, IEEE, 2010.

[88] R. Wang, L. Jiang, Y. Zhang, and J. Yang, "Sd-pcm: Constructing reliable super dense phase change memory under write disturbance," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 19–31, 2015.

[89] M. K. Tavana and D. Kaeli, "Cost-effective write disturbance mitigation techniques for advancing pcm density," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 253–260, IEEE, 2017.

[90] S. W. Fong, C. M. Neumann, and H.-S. P. Wong, "Phase-change memory—towards a storage-class memory," *IEEE Transactions on Electron Devices*, 2017.

[91] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[92] Y. Xie, W. Kim, Y. Kim, S. Kim, J. Gonsalves, M. BrightSky, C. Lam, Y. Zhu, and J. J. Cha, "Self-healing of a confined phase change memory device with a metallic surfactant layer," *Advanced Materials*, vol. 30, no. 9, p. 1705587, 2018.

[93] J. Kong and H. Zhou, "Improving privacy and lifetime of pcm-based main memory," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 333–342, IEEE, 2010.

[94] A. Devices, "Bios and kernel developer's guide (bkdg) for amd family 15h models 00h–0fh processors (2012)," *URL https://www. amd. com/system/files/TechDocs/42301_15h_Mod_00h-0Fh_BKDG. pdf*.

[95] I. X. Processor, "E7 family: Reliability, availability, and serviceability," 2012.

[96] H. P. Enterprise, "How memory ras technologies can enhance the uptime of hpe proliant servers," 2016.

[97] H.-M. Chen, S.-Y. Lee, T. Mudge, C.-J. Wu, and C. Chakrabarti, "Configurable-ecc: Architecting a flexible ecc scheme to support different sized accesses in high bandwidth memory systems," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 646–659, 2018.

[98] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics division*, vol. 11, no. 1-23, pp. 5–7, 1997.

[99] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 101–112, IEEE, 2015.

[100] J. Kim, M. Sullivan, S. Lym, and M. Erez, "All-inclusive ecc: Thorough end-to-end protection for reliable computer memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 622–633, IEEE, 2016.