Persistent Processor Architecture

Jianping Zeng
Purdue University
West Lafayette, IN, USA
zeng207@purdue.edu

Jungi Jeong*
Purdue University
West Lafayette, IN, USA
jungijeong@purdue.edu

Changhee Jung
Purdue University
West Lafayette, IN, USA
chjung@purdue.edu

ABSTRACT

This paper presents PPA (Persistent Processor Architecture), simple microarchitectural support for lightweight yet performant whole-system persistence. PPA offers fully transparent crash consistency to all sorts of program covering the entire computing stack and even legacy applications without any source code change or recompilation. As a basis for crash consistency, PPA leverages so-called store integrity that preserves store operands during program execution, persists them on impending power failure, and replays the stores when power comes back. In particular, PPA realizes the store integrity via hardware by keeping the operands in a physical register file (PRF), though the stores are committed. Such store integrity enforcement leads to region-level persistence, *i.e.,* whenever PRF runs out, PPA starts a new region after ensuring that all stores of the prior region have already been written to persistent memory. To minimize the pipeline stall across regions, PPA writes back the stores of each region asynchronously, overlapping their persistence latency with the execution of other instructions in the region. The experimental results with 41 applications from SPEC CPU2006/2017, SPLASH3, STAMP, WHISPER, and DOE Mini-apps show that PPA incurs only a 2% average run-time overhead and a 0.005% areal cost, while the state-of-the-art work suffers a 26% overhead along with prohibitively high hardware and energy costs.

ACM Reference Format:
Jianping Zeng, Jungi Jeong*, and Changhee Jung. 2023. Persistent Processor Architecture. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3613424.3623772

1 INTRODUCTION

Nonvolatile memory (NVM) technologies such as ReRAM [3, 13], 3D XPoint [39], PCM [68, 121, 125, 139], and STT-MRAM [14, 47, 66, 74, 114] have emerged as alternatives to DRAM. Thanks to their byte-addressability, high areal density, and in-memory persistence, they are to be used as nonvolatile main memory (NVMM)—also known as persistent memory (PMEM). That is, they can transparently replace DRAM to accommodate persistent applications with large memory footprint and obviate the need for serializing data in a block device to survive power failure.

*Now at Google.

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623772

However, it is not easy to make this obvious use case (*i.e.,* transparent NVMM) in reality. For example, while Intel Optane persistent memory (PMEM) [22, 62, 69, 144, 151] provides the transparent use of PMEM called *memory mode* where DRAM is used as the last-level cache atop PMEM, *the Optane manual states that the PMEM works as volatile memory [50]. The Optane persistent memory is not persistent at all*; this is mainly due to the difficulty of maintaining crash consistency in the memory mode[1]. As a result, under the memory mode, users have no choice but to risk the loss of all PMEM data in case of power failure.

Although PMEM offers *app-direct mode* where DRAM is used as main memory and PMEM serves as persistent heap [50], it pawns off the hard work of persistent programming on users, trading the transparency for in-memory persistence. In this partial-system persistence (PSP) model [11, 20, 21, 25, 35, 75, 128, 146], users must delineate a part of code that requires persistence, rewrite the data structures used therein with crash consistency and memory persistency [32] in mind, and often devise application-specific recovery code tailored to the data structures [40, 49, 70, 79, 106, 127]. Besides, PSP requires dedicated PMEM allocation interfaces such as `pmalloc` [23], rendering already error-prone persistent programming more complex [29, 91–93, 96, 99, 110]. While using transactions [10, 20, 72, 84, 140] or failure-atomic sections [11, 46, 51, 85] mitigates the programming complexity, the resulting persistent program is slower than the original one due to the undo/redo logging involving persistence barrier (`clwb` and `sfence` for x86).

Given the limitations of PSP and the demand for transparent use of PMEM without sacrificing the in-memory persistence and crash consistency, there is an increasing interest in whole-system persistence (WSP) [57, 107] which covers all sorts of applications—rather than being limited to a small set of PSP application domains such as in-memory index structures/databases and key-value stores. That is, *WSP is agnostic to program semantics yet capable of recovering any kind of program from power failure no matter when it occurs!*

One naive approach to WSP is flushing all volatile states (register files, SRAM caches, and DRAM cache) to PMEM when power is about to be cut off. For example, Narayanan *et al.* [107] propose to use residual energy in uninterruptable power supply (UPS) and persist all volatile data before impending power failure, which requires a considerable amount of energy to be secured for flushing. In a similar vein, Intel's extended asynchronous DRAM refreshing (eADR) flushes the entire cache contents to PMEM upon power failure using a backup battery. However, eADR also leads to significant energy cost requiring a bulky supercapacitor of 3400 mm^3 [4]; this situation gets even worse for a deeper cache hierarchy that is

[1]Since PMEM here is transparently used as main memory without any code change, it is solely the architecture's responsibility to flush data through the deep cache hierarchy (L1~DRAM caches) and keep PMEM states consistent across power failure for correct recovery.

driven by ever-increasing working sets of data-intensive applications [12, 71]. Apart from the inability to persist other volatile states such as registers, eADR cannot guarantee crash consistency for PMEM's memory mode—as it is unaffordable to reserve a sufficient amount of energy for flushing the data of large DRAM cache to PMEM; typical servers in data centers are equipped with more than 1TB DRAM. Given all this, it has been practically impossible to achieve WSP on the cheap.

To this end, this paper presents *Persistent Processor architecture* (PPA), the first of its kind to realize transparent, lightweight, and performant WSP without recompilation for all program embracing legacy software whose source code is unavailable. We found that crash inconsistency is caused by unpersisted stores left behind power failure and can be corrected by replaying (persisting) them in the wake of the power failure. Suppose the program commits 3 stores (*strA*; *strB*; *strC*) in a row, and due to cache replacement, the youngest *strC* is persisted in PMEM before the older ones. Although this violates the program semantics if a power outage occurs while others are cached, it is possible to fix the inconsistency by replaying *strA* and *strB*—unpersisted before the outage—when power comes back. We can even relax this for simple hardware implementation, *i.e.,* rather than tracking the (un)persistence of each individual store, PPA instead replays all 3 committed stores and resumes the interrupted program following the last committed instruction in the wake of the outage.

To achieve that, it is essential to preserve the registers of stores (for replay) and other committed instructions (for resumption of the interrupted program) across power failure. The implication is two-fold: (1) PPA should prevent store registers from being overwritten; this is so-called store-integrity [152]. (2) Both store registers and other committed instruction registers must be able to survive power outage, *i.e.,* PPA should save the registers on the outage—using a tiny capacitor whose energy is six orders of magnitude smaller than what eADR requires—for the replay and the resumption in the wake of the outage.

In particular, PPA realizes the *store integrity* in the core microarchitecture at a low cost. The key insight is that the values of store registers are retained in the corresponding physical registers[2] until they are deallocated. For example, once the architectural register $r0$ of a store is renamed to a physical register $p0$, PPA can retrieve $r0$ by reading the value from $p0$ unless it is remapped and overwritten by another instruction. To preserve the physical registers to which architectural registers of stores are renamed, PPA proposes to delay the deallocation of the physical registers—though the reorder buffer (ROB) already commits the store instructions[3]. Recall that out-of-order cores have a lot more physical registers than architectural ones to minimize the stalls caused by the lack of physical registers [33]; a physical register file (PRF) tends to be underutilized most of the time since only a part of instructions in ROB (30% in our experiments), *e.g.,* loads and ALU operations, define new registers. Prior work also observes this phenomenon, which leads to the advent of simultaneous multi-threading (SMT) [102, 103, 136–138, 156], PRF bank switching [118], and physical register inlining [83]. The takeaway is that due to PRF underutilization, PPA can delay the deallocation of store registers with minimal run-time overhead.

Such register-renaming-based store integrity is a building block of PPA enabling *region-level persistence*, where store integrity is ensured within each region (epoch) [59] for crash consistency as well as *lightweight yet performant WSP*. PPA dynamically delineates the regions, performing region-level persistence and physical register reclamation across their boundaries; whenever PRF runs out, PPA starts a new region (epoch) with a persist barrier, which ensures the committed stores of the prior region have already been written to PMEM and reclaims those physical registers mapped by the stores. To persist the stores of each region efficiently, PPA uses asynchronous writeback overlapping them with the execution of other instructions in the region as prior work [9, 54, 56, 60, 111, 130]. It turns out that the region size is long enough to fully hide the store persistence latency, thanks to the large PRF of modern out-of-order cores. If any region is interrupted by a power outage, PPA checkpoints minimal architectural states, *e.g.,* a part of PRF and hardware structures related to register renaming [42]. In the wake of the outage, PPA restores those checkpointed states, replays the committed stores of the interrupted region, and resumes the program from the last commit point before the outage—rather than rolling back to the beginning of the interrupted region—for correct and efficient recovery.

To evaluate PPA, we test it with 41 applications from SPEC CPU2006/2017 [8, 43], SPLASH3 [123], STAMP [101], WHISPER [105], and DOE Mini-apps [63, 135]. The experimental results show that PPA incurs only an average of 2% run-time overhead compared to the baseline (running original applications on PMEM's memory mode lacking in-memory persistence and crash consistency support). In summary, PPA makes the following contributions:

- PPA is the first lightweight yet performant whole-system persistence that introduces minor modifications on the hardware, *e.g.,* 2 registers and 1 queue, and only needs a tiny capacitor of 21.7 μJ, unlike eADR that requires a supercapacitor of 550mJ.
- PPA outperforms the complex state-of-the-art compiler and architecture codesign approach [57] in terms of all aspects, such as run-time performance, energy requirement, and hardware cost.
- PPA treats the underlying cache hierarchy as a black box, thus being suitable for current/future caches with an arbitrary depth of the hierarchy, *e.g.,* CXL (Compute Express Link) based far persistent memory [34, 53, 61, 97, 98].
- PPA only incurs an average of 2% run-time overhead and 0.005% areal cost, which we believe paves the way to practical whole-system persistence for all, driving the revival of persistent memory production with its cost-effectiveness.

2 BACKGROUND AND MOTIVATION

2.1 Register Renaming

Register renaming [26, 42, 44, 108], serving as a basis for PPA's store-integrity region formation, provides a way to eliminate false

[2]In an out-of-order processor, architectural registers are renamed to physical registers via register renaming (see Section 2.1 for details).

[3]More precisely in the context of a unified PRF (Section 2.1), PPA does not deallocate the physical registers of stores even after ROB commits them redefining their architectural registers.

register dependence and thus enables more instruction-level parallelism (ILP). To efficiently rename architectural registers, out-of-order processors are equipped with a unified PRF as in Alpha 21264 [65], MIPS 10K [38], ARM Cortex A-series out-of-order cores [142], RISC-V SonicBOOM [157], and modern Intel processors from Pentium 4 onwards [131]. For renaming an instruction, the processor picks a register from a Free List (tracking free physical registers) and maintains such a mapping from architectural register to physical one in a register alias table (RAT), *i.e.,* any data access to the architectural register is referred to the corresponding physical register by consulting the RAT. Once ROB retires the instruction, the processor puts the mapping to a commit rename table (CRT) for facilitating exception handling and debugging.

In particular, the physical register can only be reclaimed to the Free List when a later instruction redefining the associated architectural register gets retired from ROB—because the physical register value is no longer used thereafter.

2.2 PSP vs WSP

PSP has been a *de facto* standard for server-class systems backed with Intel Optane persistent memory (PMEM) to ensure the crash consistency of their user applications. However, this paper argues that PSP is inferior to WSP for 3 reasons: high performance overhead, programming/maintenance burden, and the risk of losing all system-level states upon power failure.

First, the *app-direct mode* of PMEM cannot take advantage of the deep cache hierarchy despite the ever-increasing data footprint of PSP applications. Our experiment (Section 7.2) indicates that due to the inability to leverage DRAM as a cache, even an ideal PSP design is significantly (up to 2.4x and 1.39x on average) slower than the *memory mode* of PMEM for memory-intensive applications. Second, PSP is not transparent and requires programmers either to redesign their data structures with persistence and recoverability in mind—incurring severe bugs during development [29, 91–93, 96, 99, 110] and maintenance costs in the future [5, 124, 130, 154]—or to leverage transactions for mitigating the programming burden[4]. Third, PSP can only recover the states of user applications and hence puts operating systems at the risk of losing their entire states upon power failure, while WSP like PPA can ensure that the entire system states are consistent across power failure; see Section 5 for details.

Not only does WSP eliminate PSP programming and maintenance costs, but it also makes persistent applications faster with the DRAM cache. Of course, for those using PMEM's memory mode to leverage the deep cache hierarchy, WSP offers them persistence and crash consistency without hurting the transparency and performance. This is particularly beneficial for HPC applications (e.g., Mini-apps) whose states must be saved to storage on a regular basis. We believe that lightweight persistence/recoverability, *e.g.,* PPA, can enable performant application-level resilience—related to one of the nation's exascale challenges [58, 113, 160]—by obviating the need for expensive periodic global checkpointing to storage.

[4]Either way, the resulting performance overhead is so significant that PSP cannot be used for those who expect similar performance to that of running their applications in PMEM's *memory mode.*

2.3 Region-Level Persistence for WSP

Prior techniques [18, 152, 161] recently investigate region-level persistence to provide crash consistency in energy harvesting systems (EHSs) [15, 19, 86, 95] where WSP is the norm. These techniques partition the program into a series of regions (akin to recoverable epochs) where their boundaries serve as recovery points. Either compiler [18, 152, 161] or hardware (this work) is responsible for the region formation and the persistence of each region. In particular, each region should ensure that all its stores are persisted before the next region starts so that the program can be recovered by restarting the power-interrupted region upon power back.

However, such a region-level persistence scheme incurs a non-negligible performance overhead, since the program must wait at each region boundary for the preceding region to persist its stores, *i.e.,* pausing until they are all written back to nonvolatile memory (NVM). While the prior work leverages ILP to overlap the persistence latency with the execution of other instructions, they still cause significant performance degradation—especially in the presence of a more deep cache hierarchy—because their regions are too short to fully hide the long latency with ILP.

2.4 Store Integrity for Performant WSP

The key observation PPA builds upon is that we can safely recover the system states by *replaying stores that are potentially unpersisted before power outage.* Although this principle has been investigated and adapted by many prior approaches as a concept of atomic stores with logging them all [9, 18, 56, 84, 140], the prior schemes suffer from the problem of doubling NVM stores—known as *write amplification.*

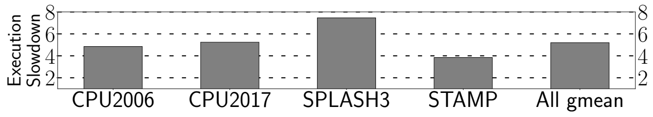


Figure 1: ReplayCache's slowdown to the baseline (running original applications on PMEM's memory mode)

To achieve high-performance WSP, we make another observation that crash inconsistency is essentially caused by the mismatch between the program order of committed stores and the order in which their cache blocks are written back to NVM. To be specific, a younger store might be evicted (persisted) to NVM while the older ones are cached; if power failure happens before their persistence, NVM status becomes inconsistent across the failure on which the data of the older stores are lost since they have not been persisted. This finding inspires us to recover the inconsistent NVM status by rewriting **only** those potentially unpersisted stores to NVM in the wake of the power failure—unlike traditional undo loggings that checkpoint **all** stores. The upshot is that no matter which random order of persisted stores is across power failure, it is always possible to correctly recover by replaying all committed stores left behind the failure and resuming from the last commit point. Zeng *et al.* show that store replaying needs compilers to prevent the store registers from being clobbered by following redefinitions, which requires a special register allocator; they call this *store integrity* in their energy harvesting work, ReplayCache [152], and use compiler-based region-level persistence to divide the

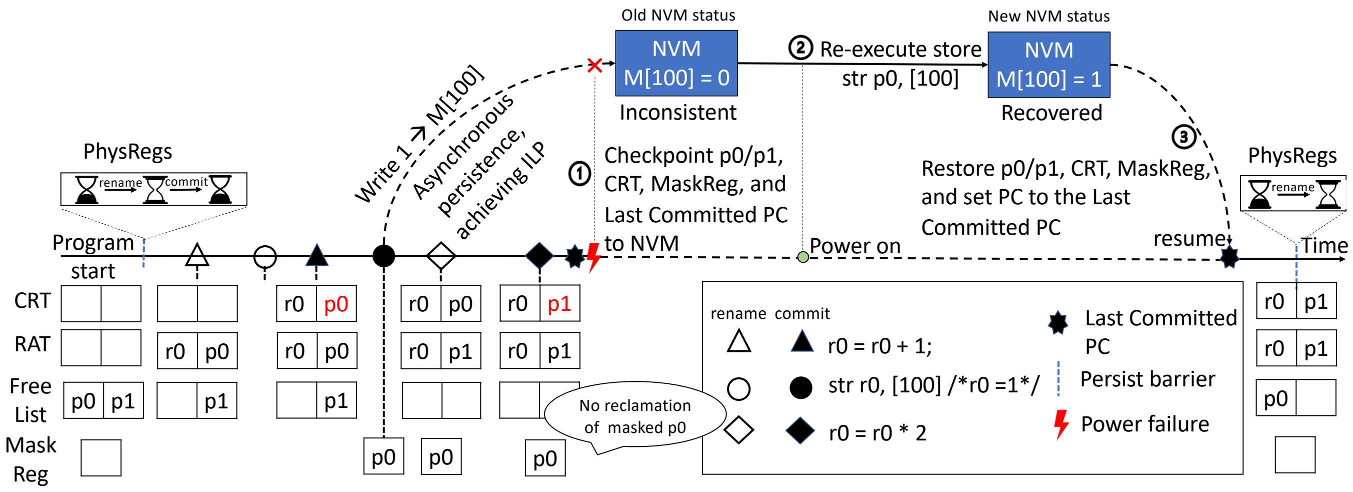


Figure 2: PPA overview; for store integrity, $p0$ is not recycled even after the multiplication commits

program to a series of regions where store integrity is enforced to guarantee crash consistency.

Unfortunately, ReplayCache incurs too much performance overhead (5x average slowdown as shown in Figure 1) when used to achieve WSP for server-class cores; see Table 2. The reason is 2-fold: (1) ReplayCache's regions are so short (average 12 instructions in regions) that they cannot accomplish enough ILP to hide the region-level persistence latency through multi-level caches. That is mainly due to the inherent issues of ReplayCache's compiler analyses, *e.g.,* function calls/loops, scarce architectural registers, and energy-aware region splitting for avoiding *stagnation* [17, 18] in EHSs. Hence, the short region leads to frequent pipeline stalls at each region boundary serving as a persist barrier; (2) ReplayCache inserts `clwb` after each store to write it back to NVM, which doubles the instruction count and places high pressure[5] on store queue whose overflow stalls the pipeline as well. Unlike ReplayCache, PPA achieves performant WSP for server-class cores causing only a 2% overhead (Section 7.1).

3 PPA OVERVIEW

PPA aims to achieve a lightweight WSP that works for a deep cache hierarchy, where DRAM cache is used as in PMEM's memory mode, without sacrificing the transparency (*i.e.,* keeping the entire software stack as is and obviating the need for recompilation) and the performance. PPA adopts store integrity for crash consistency, but its novel hardware design for the integrity enforcement makes it possible to realize a performant WSP at a low cost. In particular, PPA leverages ample physical registers in out-of-order cores to preserve store registers; it dynamically delineates the region (epoch) boundary whenever physical registers run out. In this way, sufficiently long store-integrity regions serve as the basis for failure recovery, thus effectively hiding the store persistence latency.

Figure 2 depicts how PPA realizes WSP based on register renaming of a modern out-of-order core[6]. In the figure, commit rename table (CRT), register alias table (RAT), and Free List are existing microarchitectural components. CRT keeps the mapping from an architecture register to a physical register for committed instructions, while RAT records that for in-flight instructions. The free list maintains free registers for later renaming use. PPA proposes *MaskReg*, a bit vector, to record which physical register is used by prior committed stores and therefore should not be remapped (overwritten) by the following redefinitions.

In Figure 2, upon renaming a destination architectural register $r0$ (*i.e.,* △ $r0 = r0 + 1$), the processor removes a physical register $p0$ from the free list and puts the mapping from $r0$ to a physical register $p0$ into RAT as usual. Thus, for renaming the following store (◯), *i.e., str* $r0$, [100], the reference to $r0$ is replaced by $p0$. Once the addition instruction commits (▲), making the defined value of $r0$ architecturally visible, the processor puts the mapping $r0 \longrightarrow p0$ in CRT as usual. In particular, on the commit of the store (◯), PPA starts to track $p0$ in MaskReg, watching it for store integrity. When the following redefinition of $r0$ is renamed (◊), the multiplication instruction obtains $p1$—not $p0$ since it is already masked—from the free list with RAT updated accordingly. Additional pipeline details are deferred to Section 3.3.

3.1 Dynamic Region Formation

Similar to prior techniques [18, 152], PPA also provides region-level persistence. However, what makes PPA stand out from them is its ability to build regions dynamically without user intervention, recompilation, and significant performance loss. PPA instead leverages an existing microarchitectural feature to deliver the region formation with the store integrity enforced at a low cost. In particular, PPA considers the number of free physical registers to decide when to place a region boundary (persist barrier). As shown in Figure 2, PPA places the boundary (barrier) when no free physical register is available at the renaming stage of the out-of-order pipeline (⧗). Once PPA ensures at each region boundary that the committed stores of the finished region are all persisted, it reclaims their physical registers with MaskReg cleared—before starting the next region, as shown at the left bottom of the figure.

[5]The `clwb` instruction occupies a store queue entry.
[6]We assume a server-class Intel Skylake core [30] though PPA can be generalized to other out-of-order cores.

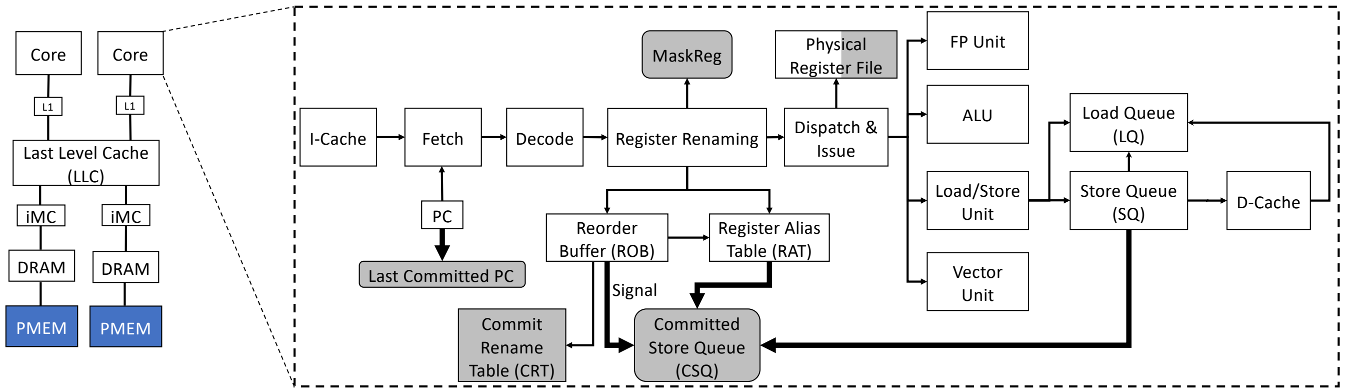


Figure 3: PPA with Intel's memory mode; rounded rectangles corresponds to new components, while thick lines to new signal or data paths; shaded parts are JIT-checkpointed upon an outage (PPA checkpoints only those registers masked by MaskReg/CRT)

3.2 HW-Based Asynchronous Store Persistence

Although prior software-logging-based PSP techniques guarantee consistent NVM status across power failure, they incur significant performance overhead because of a persist barrier (*e.g.,* `clwb` and `sfence` in x86). In contrast, PPA does not block the pipeline execution while stores are being persisted to NVM. That is, once the data being stored is merged into the L1 data cache (◯ in Figure 2), the L1 data cache controller immediately asynchronously writes back the resulting dirty cacheline to NVM in the background, keeping the pipeline busy with other instruction executions in the meantime.

To ensure all stores prior to the end of a region are already persisted in NVM before committing following instructions, *PPA treats every region boundary (the last instruction of each region) as a special persist barrier.* Therefore, the core pipeline waits until the acknowledgment of persisting the region's all prior stores in NVM is received by the core before entering the next region. While stalling the pipeline can lead to a slowdown due to the wasted cycle time, our experimental results show that our hardware-based store persistence has a minimal impact on the pipeline performance due to long enough regions (see Section 7.5) and thus resulting in negligible stall cycles at the end of regions (see Section 7.3).

3.3 Dynamic Enforcement of Stores Integrity

Figure 2 shows how PPA ensures store integrity on the fly during the pipeline execution. Upon retiring $str\ r0, [100]$ (◯ in the figure) whose $r0$ was renamed to $p0$, PPA masks $p0$ in MaskReg to notify it is occupied by the store, which makes the target register of the following multiplication instruction renamed to $p1$ (◊) instead of $p0$. Unlike conventional cores, upon retiring the multiplication (♦ $r0 = r0 * 2$) with updating CRT with $r0 \rightarrow p1$, PPA does not reclaim the physical register $p0$ which is associated with $r0$'s prior definition $r0 = r0 + 1$—though its value can no longer be used due to the retirement of the multiplication overwriting $r0$. That is because $p0$ is masked as a committed store register in MaskReg, and it should be preserved in case of power failure so that the store can be replayed in the wake of the failure. In this way, PPA not only guarantees store integrity in each region but also achieves performant WSP with a much longer region size than the compiler-based prior work [152], thus hiding the store persistence latency.

3.4 Checkpoint and Recovery Protocol

To achieve correct program execution across power outage, all the store registers preserved by our register renaming trick must survive power failure. For this reason, PPA should maintain necessary microarchitecture status such as CRT across the outage. Also, in the wake of power failure, PPA should be able to resume the program right after the last commit point behind the outage.

In light of this, PPA exploits just-in-time (JIT) checkpointing to save minimal architectural states—*e.g.,* physical register $p0$, CRT, and the last committed PC as shown in Figure 2 (①)—to a designated checkpoint storage in NVM, when power is about to be cut off. Owing to its simplicity, PPA only requires a tiny capacitor to secure energy for JIT checkpointing, while Narayanan's [107] and eADR's demand a significantly large bulky Li-thin battery or supercapacitor [4] (Section 7.13). When the power comes back, PPA first replays all committed stores behind the failure, *e.g.,* $str\ p0,\ [100]$ in Figure 2 (②), and restores other checkpointed states such as CRT (③). Then, PPA resumes the interrupted region from the latest uncommitted instruction following the *last committed PC* to continue program execution. More details are deferred to Section 4.5 and Section 4.6.

4 PPA IMPLEMENTATION DETAILS

Figure 3 shows PPA's microarchitecture with its 3 newly added components; *Last Committed Program Counter (LCPC), Store Operands Mask Register (MaskReg),* and *Committed Store Queue (CSQ).* The *LCPC* register keeps the PC of the last committed instruction so that a power-interrupted program can resume thereafter in the wake of power failure. Note that PPA does not save or recover architectural status related to speculation, such as in-flight instructions in ROB. The *MaskReg* comprises as many bits as the PRF size. Each set bit of *MaskReg* indicates that the corresponding physical register has been used as an operand of any committed store in the current region and thus prevents those physical registers from being updated by the following instructions of the region. Finally, the *CSQ* is a circular FIFO queue for tracking committed stores per region. When a store retires from ROB, a pair of (1) the index of the source physical register and (2) the destination physical address of the store is inserted into the rear position of *CSQ.*

Actions of PPA across an Outage: Upon a power outage, PPA has 5 components (shaded in Figure 3) JIT-checkpointed in NVM:

CSQ, LCPC, CRT, MaskReg[7], and the physical registers tracked by CSQ/CRT. When power comes back, PPA (1) restores the checkpointed registers, MaskReg, CRT, LCPC, and CSQ from NVM, (2) scans CSQ entries from front to rear re-executing the stores committed before the outage[8], (3) populates RAT with the restored CRT, and (4) resumes the power-interrupted program right after LCPC.

Note that once a region is persisted at the boundary, *i.e.,* the pipeline receives an acknowledgment that all the committed stores of the region have been persisted to NVM, PPA clears both the CSQ and the MaskReg—reclaiming the store registers masked therein—before starting the next region. Thanks to the long region size (Section 4.1) and the asynchronous writebacks (Section 4.3), PPA effectively hides the store persistence latency at each region end.

4.1 Enforcing Store Integrity Efficiently

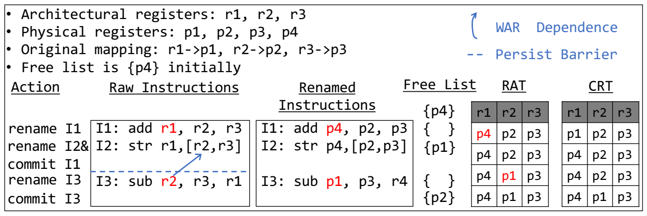


Figure 4: Impact of register renaming on the region length

At first glance, forming store-integrity-preserving regions seems easy, *i.e.,* placing a region boundary right before the redefinition of store registers to preserve their values within a region. For example, placing a region boundary (persist barrier) after store *I*2 in Figure 4 ensures store integrity and post-crash consistency but yields short regions because of a write-after-read (WAR) dependence on store register *r*2. A sophisticated compiler approach might form relatively longer regions by renaming the redefinitions of previous store registers—unless architectural registers run out—as in ReplayCache [152]. However, the prior approach [152] to store integrity still generates short regions due to a limited number of architectural registers, *e.g.,* 16 general-purpose registers in x86. The crux of the problem is that ReplayCache pays for persist barrier overheads so often at each end of such short regions.

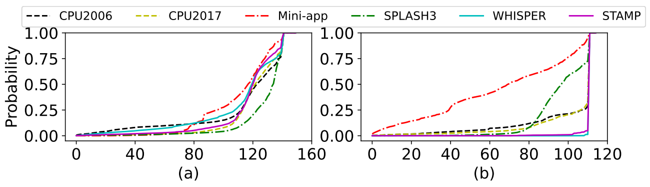
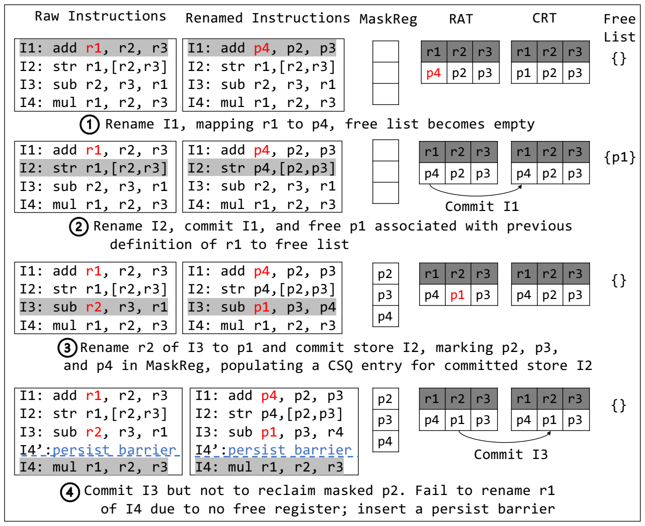


Figure 5: (a): CDF of free integer registers; (b) CDF of free floating-point registers

Fortunately, the out-of-order cores already have the ability to eliminate WAR dependence with register renaming [42], *e.g.,* renaming *r*2 to *p*1 for the subtraction *I*3 in Figure 4. That way the pipeline can execute *I*3 without redefining a register *r*2, thus obviating the need for the persist barrier after the store *I*2—unlike the prior approach [152] that needs the barrier to persist the store before *I*3—forming longer regions than it can.

Note that the number of physical registers is much larger than that of architectural registers and that they are often idle. Figure 5 shows that CDFs of free integer and floating-point registers—sampled every cycle[9]—respectively. For example, 75% of the program execution cycles, the baseline core has 138/110 integer/floating-point registers not utilized for CPU2006. As such, PPA's region size can be sufficiently long to hide the persistence latency of stores in each region without incurring slowdown a lot.

4.2 Forming Longer Regions at a Low Cost



Raw Instructions
Renamed Instructions
MaskReg
RAT
CRT
Free List
I1: add r1, r2, r3
I2: str r1,[r2,r3]
I3: sub r2, r3, r1
I4: mul r1, r2, r3
I1: add p4, p2, p3
I2: str p4,[p2,p3]
I3: sub p1, p3, p4
I4': persist barrier
r1 r2 r3
{}
{p1}
① Rename I1, mapping r1 to p4, free list becomes empty
② Rename I2, commit I1, and free p1 associated with previous definition of r1 to free list
③ Rename r2 of I3 to p1 and commit store I2, marking p2, p3, and p4 in MaskReg, populating a CSQ entry for committed store I2
④ Commit I3 but not to reclaim masked p2. Fail to rename r1 of I4 due to no free register; insert a persist barrier
Commit I1
Commit I3
p2
p3
p4

Figure 6: Dynamic region partitioning by physical register file size; the free list shows its status after the action

With the above observation in mind, PPA proposes a minimal change to the instruction pipeline so that it enforces store integrity during the execution of each region. That is, PPA dynamically partitions program to a series of regions by placing a region boundary, *i.e.,* a persist barrier, upon a pipeline stall at the renaming stage; if a register-defining instruction cannot be renamed due to the lack of a free physical register in free list, PPA injects a persist barrier **right before** the instruction (see Figure 6 for details). Once the pipeline retires the persist barrier at the boundary of a region, *i.e.,* all its stores are already sure to have persisted, PPA acknowledges the renaming stage to reclaim the physical registers masked by MaskReg, clears it, and resumes the pipeline to start the next region.

Figure 6 shows a step-by-step example of how to perform dynamic region formation while preserving registers of stores. We assume a 4-bit *MaskReg* for total 4 physical registers $p1 - p4$. Initially, *MaskReg* is empty, and $p1 - p3$ are occupied by previous definitions of register $r1 - r3$, and a free list contains only $p4$. When renaming $r1$ of the addition instruction $I1$ at step ①, PPA maps

[7]MaskReg should be JIT-checkpointed as well to prevent store registers therein from being recycled even after power comes back.

[8]Even if some stores might have already been persisted, there is no harm to execute them again as each store is idempotent [27, 48, 67, 87, 88, 90, 155].

[9]We measure the number of free physical registers every cycle at the renaming stage of an OoO core (Table 2).

$r1$ to the only free physical register $p4$ and updates the RAT and the free list accordingly. Then, at step ②, when renaming a store instruction $I2$, the references to architectural registers are replaced by physical registers as usual. At the same time, the pipeline retires $I1$ instruction, deallocating $p1$ associated with $r1$’s previous definition (not shown in the figure) and updating the CRT with $r1 \longrightarrow p4$. At step ③, the pipeline renames $r2$ of $I3$ to $p1$, with RAT and the free list correspondingly updated, and commits the store $I2$ setting the bits of $p2 - p4$ in *MaskReg* to 1[10] and populates a CSQ entry in its back position for the committed store $I2$.

In particular, at step ④ where the pipeline commits $I3$ and renames $I4$, PPA takes different actions from the traditional out-of-order pipeline that allows $p2$ to be remapped. PPA does not deallocate physical register $p2$ associated with $r2$’s previous definition (not shown in the figure), though $I3$ commits redefining $r2$. This is because $p2$ is masked in *MaskReg* as a store operand. However, at this moment, there is no physical register in the free list, which makes PPA fail to rename the register $r1$ of $I4$. Thus, PPA injects a persist barrier here as a region boundary right before instruction $I4$. Once the barrier gets retired, PPA reclaims all masked physical registers $p2 - p4$ to the free list, clears *MaskReg*, and starts a new region allowing them to be reused therein.

Full CSQ as an Implicit Region Boundary: If CSQ becomes full, PPA cannot accommodate stores anymore, thus being unable to replay them for power failure recovery. Thus, PPA treats this event as a virtual region boundary where it waits for all prior stores to be persisted. Once the core receives the acknowledgment of persisting all prior stores, PPA starts a new region with CSQ and MaskReg cleared. Our experiment (Section 7.9) shows that a 40-entry CSQ rarely overflows, thus incurring a minimal performance impact.

4.3 Region-Level Asynchronous Persistence

In addition to preserving store registers for their integrity in each region, PPA also ensures that its stores are persisted to NVM before moving on to the next region. Instead of leveraging cacheline writeback instruction (`clwb` in x86) that has a lot of drawbacks, as shown in Table 1, *e.g.,* occupying a store queue entry for each store, requiring inter-core snooping, and not being able to flush data from core to NVM main memory through a DRAM cache above, PPA leverages the asynchronous store writeback which effectively takes it off the critical path [9, 54, 56, 60, 111, 130]. That is, when the data being stored is merged into L1 data cache after cache coherence transactions are completed, an asynchronous store persistence operation is generated in the write buffer (WB)[11] of L1 data cache and then issued by its controller. The implication is 2-fold: (1) the store persistence happens in the background while the core continues the execution of following instructions, achieving ILP; (2) once a store persistence operation is issued, all other cores already have up-to-date memory data.

Unlike `clwb`, *i.e.,* a cacheline writeback instruction that occupies a store buffer entry, PPA instead uses a counter register in the L1 data cache controller to record the number of stores being persisted, rather than tracking each individual store with `clwb`; the

[10]While *MaskReg* could record all operand registers of each store, we opt to keep only a data register as an optimization. See Section 4.6 for more details.
[11]WB already exists in Intel processors sitting between L1D and L2 cache for buffering dirty cacheline eviction.

Table 1: Comparison between PPA and CLWB

	Store Queue Occupied	Single Store Tracking	Snooping	Reaching NVM
CLWB in x86	✓	✓	✓	✗
PPA	✗	✗	✗	✓

counter increases for each store performed and decreases every time the controller receives the acknowledgment of the writeback completion. In particular, to lower write traffic towards NVM, PPA performs *persist coalescing* [130] on the WB for the data being persisted. That is, a younger store being persisted is merged with the old *unpersisted* one of the same address sitting in the WB. This is correct because persist barriers ensure that the WB’s data—to be persisted—are from the same region, and the stores of the following regions are not performed yet.

When the counter hits zero, the controller tells the core that all prior stores in a region are persisted to NVM, allowing both CSQ and *MaskReg* to be cleared. In this way, PPA determines if the pipeline needs to stall at the region boundary by simply comparing the counter with zero. Although such a stall might slow down the pipeline by waiting for the counter to be zero at each region boundary, it turns out that the performance impact is not significant. The reason is that the region-level persistence latency is fully overlapped with the execution of other instructions in the long regions dynamically formed by PPA (Section 7.3). Moreover, PPA’s asynchronous store writeback does not generate coherence traffic—since each core is responsible for its own writeback—thus reducing the persistence latency further.

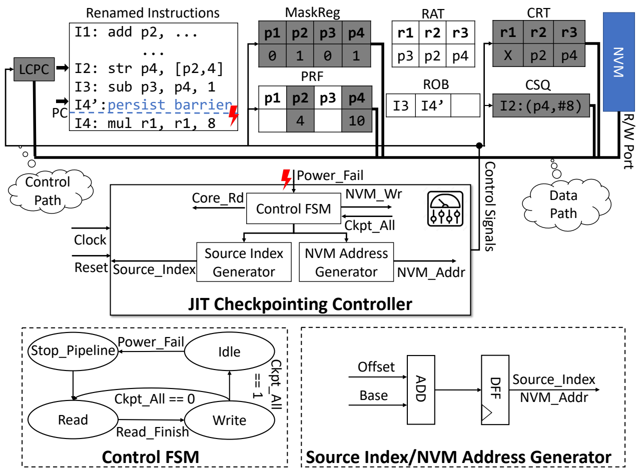


Figure 7: JIT Checkpointing logic; gray parts are checkpointed before impending power failure

4.4 Lightweight Hardware for Recovery

To achieve highly energy-efficient checkpointing and recovery, PPA needs to checkpoint only essential architectural statuses, *e.g.,* a part of physical registers used by committed stores or linked with committed instructions in the interrupted region, committed stores of the region, CRT, and program counter (PC) of the latest committed instruction, upon power failure. With checkpointing such minimal states, we can still restore consistent memory status by re-executing those stores and then resume the program execution following the latest committed instruction.

To facilitate this, PPA proposes a simple yet hardware-efficient FIFO queue called committed store queue (CSQ) and Last Committed PC (LCPC). Each CSQ entry keeps the source physical register index and the destination (physical) address of committed stores in program order, and LCPC gets updated with the PC value after committing an instruction. Note that CSQ and LCPC do not affect the existing pipeline's timing logic at all because they are out of the critical path. More importantly, CSQ is organized with a read/write port eliminating an expensive CAM structure, making a large CSQ realistic; we only need 40 CSQ entries at most as shown in Section 7.9 though. During normal program execution, the port is used to populate a CSQ entry in its rear position and to checkpoint the entire CSQ to NVM upon power failure. Finally, PPA clears CSQ at each region boundary as with MaskReg emptied, *i.e.,* when all committed stores in the finished region become persistent in NVM, before moving on to the next region.

4.5 Just-In-Time (JIT) Checkpointing on Power Failure

To ensure correct program recovery across power failure, PPA should checkpoint necessary states when power is about to be cut off. Figure 7 shows how such just-in-time checkpointing works with its circuitry implementation; upon the delivery of `Power_Fail` signal, PPA saves the contents of its 5 structures to NVM, *i.e.,* MaskReg, commit rename table (CRT), committed store queue (CSQ), a part of PRF, and last committed PC (LCPC). Note that PPA only checkpoints those physical registers marked by CRT or CSQ entries in that neither free registers ($p1$ in the figure) nor uncommitted registers ($p3$ defined by $I3$) affect correct program recovery. Similarly, PPA does not have to checkpoint any other status of in-flight instructions, *e.g.,* their RAT and ROB entries. This is because PPA can resume the execution of power-interrupted program from the latest uncommitted instruction following LCPC, when power comes back.

As with prior work on JIT checkpointing [36, 94, 95, 120, 126, 133, 143] developed for energy-harvesting systems [6, 16, 19, 152] to realize power failure recovery, PPA implements a controller that governs checkpointing and recovery[12] operations, according to each signal delivered on power failure and its wake-up. As shown in the middle of Figure 7, the controller consists of 3 components: (1) *Control Finite State Automaton (FSM)*, (2) *Source Index Generator (SIG)*, and (3) *NVM Address Generator (NAG)*. *FSM* is responsible for generating control signals to checkpoint PPA's 5 structures, *i.e.,* MaskReg, CRT, CSQ, PRF, and LCPC, into their storage in NVM. During the checkpointing process, *FSM* triggers *SIG* and *NAG* that share the same logic—shown in the bottom right of the figure—for the sum of the inputs `Base` and `Offset` to determine (1) what to be checkpointed and (2) where to save in the NVM, respectively.

It is worth noting that PPA activates its checkpointing controller only on power failure, and therefore it is out of the critical path most of the time as long as power is on, *i.e.,* PPA does not have to optimize the controller's circuitry for latency. This allows PPA to keep the controller's hardware design simple by sequentially checkpointing PPA's 5 structures[13] one entry at a time. To illustrate, as shown at the bottom left of Figure 7, *FSM* is triggered upon `Power_Fail` to transit from *Idle* stage to *Stop_Pipeline* stage, where PPA stops the core pipeline to preserve the contents of the 5 structures. Then, *FSM* moves to *Read* stage, raising the read signal `Core_Rd` on the control path so that the entry indexed by *SIG* can be read in each of 5 structures across which `Base` and `Offset` are properly updated. Upon the delivery of `Read_Finish` signal, *FSM* enters `Write` stage enabling the write signal `NVM_Wr` to write the data to the NVM address generated by *NAG*. Once the writing is done, *FSM* either goes back to *Read* stage or exits to *Idle* provided if `Ckpt_All` is activated, *i.e.,* all 5 structures are completely checkpointed.

To realize the above sequential checkpointing while maintaining a low hardware complexity, PPA exploits the existing non-temporal path [24] in x86 processors to deliver data to NVM—other than introducing a new data path. This indicates that PPA checkpoints its 5 structures at an 8-byte granularity as with their entry size (Section 7.12). Likewise, *FSM* reads PRF and CRT at an 8-byte granularity, which seems possible given that they are implemented with SRAM [38, 132, 157]. The takeaway is that the aforementioned JIT-checkpointing logic is lightweight, *i.e.,* a few hundred logic gates, keeping the overall hardware cost of PPA minimal (Section 7.12).

4.6 Power Failure Recovery Protocol

To achieve correct program recovery, in the wake of power failure, PPA restores MaskReg, CRT, and checkpointed physical registers by reloading their data from NVM as an opposite operation of the JIT checkpointing. PPA then re-executes those potentially unpersisted stores by reading the CSQ entries checkpointed in NVM. To be specific, for each CSQ entry, PPA gets both the data value by retrieving the restored PRF with an index of the checkpointed physical register and the destination address. That way, PPA writes the data value to the target address. Finally, PPA resets the PC to the instruction following the LCPC to continue the program execution.

5 INTERACTION WITH OS

This section describes how PPA interplays with the rest of the computing stack, such as the operating systems (OS), to enable system-level crash consistency.

Handling I/O Operations: To the best of our knowledge, supporting irrevocable operations such as I/O remains an open problem. PPA can be extended to have a battery-backed buffer for crash-consistent I/O operations. In this way, PPA considers any store to the buffer as persisted.

Context Switching: PPA treats context switching as is without any special consideration. In particular, PPA does not differentiate between kernel code and user program thanks to the benefits of WSP. While keeping context switching as is, PPA still guarantees correct process (de)scheduling and resumption. That is because PPA ensures that the architectural states, *e.g.,* stores and architectural registers, of a descheduled process are crash-consistent by following PPA 's JIT checkpoint and recovery protocol. That being said, PPA might have an indirect impact on performance, provided that

[12]Recovery operations are not shown in Figure 7 as they are the opposite of checkpointing operations.

[13]The order in which the structures are visited does not affect the correctness of checkpointing and recovery.

a region boundary is introduced during context switching. In reality, such a case rarely occurs because PPA forms reasonably long regions (see Section 7.5), keeping the frequency of encountering region boundaries low. Even if the case occurs, *i.e.,* PRF runs out in the middle of the context switching, and the resulting region boundary incurs the region-level persistence overhead, PPA can still minimize the stall cycles at the boundary leveraging the asynchronous store persistence, *e.g.,* only a few stall cycles occur on average (see Section 7.3). It turns out that they are negligible compared to typical context switching overhead (*e.g.,* 5-20 μs) [134, 141, 145]. Consequently, the context-switching performance would practically be the same with PPA.

Interrupt Handling and System Calls: There is no special treatment of PPA for Interrupt handling[14] and system calls—that rely on trap instructions (`syscall` in x86_64)—for the same reason above. That is, PPA guarantees that any architectural state is consistent across power failure. As such, PPA can resume interrupt handlers and system calls exactly from the power failure point without rollback. For an interrupt handler that encounters power failure in the middle of the execution, PPA can recover all committed but unpersisted stores and architectural registers and resume the handler from the last commit point in the wake of the failure.

6 DISCUSSION

Recovery for Multi-Cores: To guarantee correct recovery for multi-threaded applications on multi-core processors, we assume data-race-free (DRF) applications as required from the C/C++11 onward. DRF implies that conflicting accesses should be explicitly ordered by a synchronization primitive, *e.g.,* serializing them in a lock-protected critical section or leveraging an RMW (read-modify-write) instruction. PPA treats all synchronization primitives, including atomics and fences, as a region boundary so that their actions comply with PPA's original recovery protocol in case of power failure; for each synchronization primitive running on a core, it cannot be committed until all stores of its region are sure to have been persisted to NVM with the CSQ of the core emptied. For example, the stored data before a lock release can exist in the CSQ of at most one core. The implication is two-fold: (1) there cannot be multiple pending stores to the same address in the CSQs of different cores due to the absence of data races; (2) thus, we may replay stores in the cores' CSQs in an arbitrary order, which still achieves correct recovery—because each core's CSQ entries are disjoint with any other core's CSQ entries. That is, PPA can restore consistent NVM states of DRF applications—though it lets each core perform the recovery protocol (Section 4.6) individually—without maintaining the recovery order among the cores.

Memory Consistency Model: Although PPA is evaluated with X86 ISA (total store ordering), it works well for other consistency models, *e.g.,* relaxed memory ordering (RMO) in ARM and RISC-V, because PPA leaves load/store unit (LSQ) as is by proposing a tiny CSQ. One might think of gating those retired stores in store buffer (SB) without merging them to L1 cache as an alternative. However, it complicates the hardware design and limits the performance optimizations of RMO for 3 reasons: (1) region-level persistence prohibits inter-region store coalescing and out-of-order store write-back from SB to L1 data cache; (2) it is hard to enlarge the SB size for hiding long memory latency. That is because SB's CAM searching structure is expensive, and it must provide data within L1-hit time, which would otherwise complicate the scheduling loads with variable latency; (3) data being stored exists in both SB and PRF, wasting the energy to checkpoint the same data twice.

In-Order Cores and ROB-Style Register Renaming: Our design can be easily extended to provide WSP for both cores by accommodating data values (rather than indexes to PRF as in the current PPA) and destination addresses of committed stores in the CSQ as usual. Across power failure, the CSQ entries can be checkpointed and thus restored to recover inconsistent NVM status via replaying.

Multiple Memory Controller (MC) Support: PPA naturally supports multiple memory controllers without any hassle. This is because PPA only moves on to the next region once all stores of the prior region are persisted in NVM with the help of region-level persistence (Section 4.3); this makes it impossible to persist a younger store (in program order) destined to a near MC before the older one to a far MC, if the two stores are separated in different regions. Even if the stores exist in the same region and its power failure exposes the possible ordering violation, PPA replays them all together with other stores of the power-interrupted region in the wake of the failure. Consequently, either way PPA prevents crash inconsistency from occurring in the presence of multiple MCs.

7 EVALUATION AND ANALYSIS

All programs are compiled with -O3 flag and are statically linked. We use the Clang/LLVM 13.0.1 compiler [76, 77] to build the baseline binaries with default compilation flags. We implement the same ReplayCache region formation in the same compiler to build store-integrity binaries with disabling ReplayCache's energy-aware region splitting to enlarge the region size as much as possible.

Table 2: Microarchitectural Parameters

Component	Configuration
Full System Mode	Ubuntu 18.04 and Linux Kernel 5.4.46
Processor	8-core 4-width x86_64 OoO processor at 2GHz. Unified PRF, ROB/IQ/SQ/LQ/Integer PRF/Floating-Point PRF: 224/97/56/72/180/168
L1I	private 32KB, 8-way, 64B block, 3 cycles
L1D	private 64KB, 8-way, 64B block, 4 cycles, write back
L2	shared 16MB, 16-way, 64B block, inclusive, 44 cycles, write back
DRAM Cache (LLC)	shared direct-mapped, 4GB, DDR4 2400 8x8
PMEM	32GB, Read = 175ns/Write = 90ns, 16-entry WPQ [147, 148], 2.3GB/s write bandwidth [148]
CSQ	40-entry FIFO queue

We use the cycle-accurate simulator gem5 [7] to model an 8-core (one thread per hardware core) x86_64 Skylake-X processor with *two integrated memory controllers*, each of which manages a DRAM as an off-chip direct-mapped cache as with PMEM's memory mode. Table 2 shows the details of the microarchitectural parameters.

To measure the impact of PPA on varying programs, we choose 6 benchmark suites, *e.g.,* CPU2006/2017 [8, 41, 43, 82], SPLASH3 [123], STAMP [101], WHISPER [105], and Mini-apps [63, 135], which represent different application domains from CPU performance benchmarks, shared-memory multi-core systems, transactional applications, key-value stores, to memory-intensive programs.

[14]We use the term interrupt to describe software exception and hardware interrupt.

Table 3: Data inputs for DOE Mini-apps and WHISPER apps

Application	Short Description	Simulation Data Input	Memory Footprint
LULESH [63]	High instruction and memory-level parallelism.	-s 100	664MB
XSBench [135]	Stress memory system with little computations.	-s small	241MB
PC [73]	Update in hash-table.	8 100000	196MB
RB [73]	Insert/delete nodes in a red-black tree.	8 100000	166MB
SPS [73]	Swap random entries of an array.	8 200000	264MB
TATP [73]	`update_location` transaction.	8 100000	287MB
TPCC [73]	`add_new_order` transaction.	8 100000	110MB
r20w80 [100]	Memcached with 20% reads and 80% writes	-m 1000 -t 8	189MB
r50w50 [100]	Memcached with 50% reads and 50% writes	-m 1000 -t 8	189MB

We simulate the entire SPLASH3/STAMP/WHISPER program in the full system (FS) mode of gem5 with 8 cores by default. To stress the memory system and demonstrate the benefits of enabling DRAM as a cache, we use reference inputs to simulate SPEC CPU applications and the data inputs specified in Table 3 for Mini-apps and WHISPER. Additionally, we modify the source code of WHISPER applications to increase the key/value sizes, keeping their data footprint large enough; see Table 3. Similarly, we follow the prior work [105] using Memcached 1.6.18 [100] as a server and `memaslap` from libMemcached 1.0.18 [2] as a client to initiate 8 threads sending 10000 requests to the server. For each memaslap request, we test two ratios of read-to-write operations: 20/80 and 50/50 for write-intensive and read-intensive. In particular, we set the key and value sizes of Memcached to 64 bytes and 1KB, respectively. We follow the same way as prior work [27, 28, 80, 89, 122, 129, 153] to fast forward the first 5 billion instructions and then simulate the next 1 billion instructions with a detailed CPU model.

7.1 Run-time Overhead Analysis

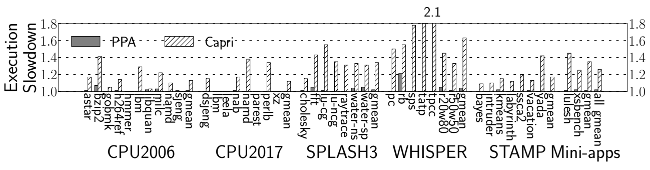


Figure 8: Normalized slowdown of PPA and Capri to the baseline (original binaries running on PMEM's memory mode); lower is better; 40 CSQ entries

As a comparison, Figure 8 presents run-time overheads of PPA and the state-of-the-art WSP—Capri [57] which incurs high hardware costs due to the separate FIFO persist path between the core and NVM and the complex undo+redo logging structures; see Table 6 for the comparison. To be practical, we set the persist path bandwidth of Capri to 4GB/s instead of its original unrealistic 32GB/s[15]. PPA incurs an average of 2% overhead, while Capri incurs a 26% overhead due to its 11x shorter regions than that of PPA; see Section 7.5. Note that PPA only incurs a slightly high overhead for `rb` of WHISPER due to the relatively higher write traffic towards NVM, as confirmed in Figure 15 and Figure 18.

We also compare PPA and PMEM's memory mode to the DRAM-only system with a 32GB DRAM. Figure 9 depicts that PPA and the

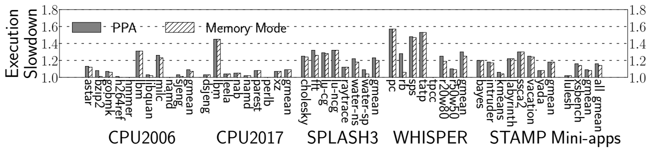


Figure 9: Normalized slowdown to a DRAM-only system with 32GB volatile memory; lower is better

memory mode are 16% and 14% slower than the system only with a 32GB DDR4 DRAM, respectively. The results are encouraging in that PPA's cost of making the DRAM-only system persistent is comparable to the run-time overhead of PMEM's memory mode that does not offer persistence. In particular, `lbm` and `pc` incur *e.g.*, 44% and 58% overheads, respectively. That is because they have poor locality and thus the DRAM cache only increases the critical path of their memory accesses with a lot of misses.

7.2 Comparison to Partial-System Persistence

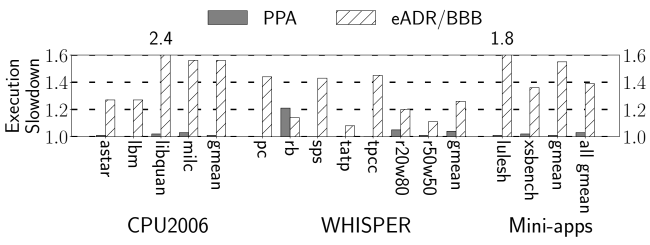


Figure 10: Normalized slowdown of PPA and eADR/BBB (ideal PSP) to the baseline (running original program on PMEM's memory mode); lower is better

To demonstrate the benefits of enabling DRAM as a cache for the applications with high L2 miss rates (ranging from 18% to 100%), we compare PPA to an optimized version of BBB [4] whose performance is close to that of eADR, representing the upper-bound performance of a PSP scheme. Figure 10 shows that PPA incurs only an average of 3% run-time overhead for these programs, while BBB/eADR slows down the programs by 1.39x on average and up to 2.4x for `libquantum`. Notably, PPA underperforms BBB/eADR slightly for `rb`. The reason is two-fold: (1) PPA leads to higher contention in WPQ (Section 7.7) due to the store persistence; (2) `rb` exhibits high locality (4% L2 miss rate) and thus has less write traffic towards NVM for the baseline.

7.3 Analysis of Stall Cycles at Region End

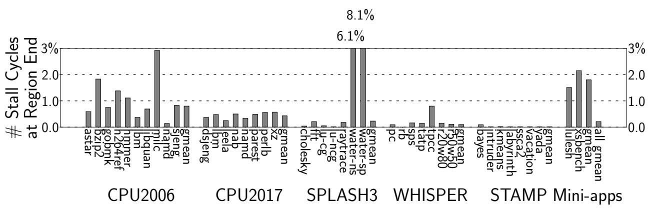


Figure 11: Stall cycles at the end of regions as a percentage of their execution time; lower is better

[15]We get Capri's source code and figure out its default persist path bandwidth is 32GB/s.

Figure 11 shows the average ratio of the stall cycles occurred at the end of each region to the execution cycles of that region. Thanks to the sufficiently long region size (*i.e.,* high ILP for hiding store persistence latency), PPA only increases the stall cycle ratio of the baseline (PMEM's memory mode) by 0.21% on average, showcasing why PPA incurs a low run-time overhead, *i.e.,* 2% on average. Figure 11 also shows why PPA incurs a relatively higher overhead for `water-ns` and `water-sp`; the reason is that, as shown in the figure, these two applications have more stall cycles, *i.e.,* 6.1% and 8.1%, respectively due to their shorter regions and more stores therein (see Figure 13).

7.4 Impact on PRF Pressure

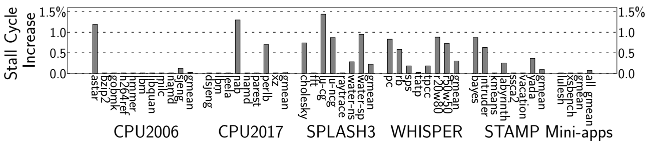


Figure 12: Increase in stall cycles at the renaming stage when the core is out of physical registers; lower is better

For both the baseline (PMEM's memory mode) and PPA, we measure the number of stall cycles due to the lack of physical registers in the renaming stage of the simulated core. Figure 12 highlights that PPA incurs negligible extra stall cycles (0.07%) on average compared to the baseline. The reason is two-fold: (1) The core pipeline stall caused by running out of free registers rarely occurs due to the sufficient amount of free registers (see Figure 5). (2) Although the stall happens, PPA tends to spend minimal cycles at the end of regions (see Figure 11) and thus quickly deallocates their reserved registers for later use.

7.5 Dynamic Region Characteristics

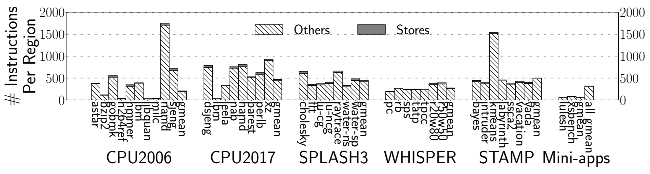


Figure 13: Average number of stores and others in regions

To demonstrate why PPA incurs such a low run-time overhead, we measure the number of stores and others in each region. As shown in Figure 13, each region has 301 other and 18 store instructions on average thanks to the abundant free registers, while Capri's average region size is only 29. As a result, PPA has enough room to keep the pipeline busy while asynchronously persisting the data being stored to NVM without waiting at each region boundary. Note that some applications, *e.g.,* `bzip2` and `libquantum`, have smaller region sizes due to their heavy register usage.

7.6 Sensitivity to Deeper Cache Hierarchy

To evaluate the sensitivity to deeper cache hierarchy, *i.e.,* 3-level SRAM caches atop DRAM cache, we add a shared 16MB 16-way

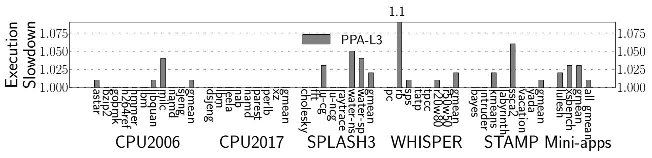


Figure 14: Normalized slowdown of PPA to the baseline when using L3 cache atop DRAM cache; lower is better

set-associative L3 cache of 44-cycle hit latency to both PPA and the baseline (PMEM's memory mode). We also alter the existing L2 cache in Figure 2 to a private L2 with 14-cycle hit latency and 1MB. Figure 14 shows that PPA incurs a negligible overhead (1%) even when the L3 cache is used atop DRAM cache thanks to PPA's sufficiently long region size (see Section 7.5) that can cover the extended store persistence latency through the hierarchy.

7.7 Sensitivity to WPQ Size

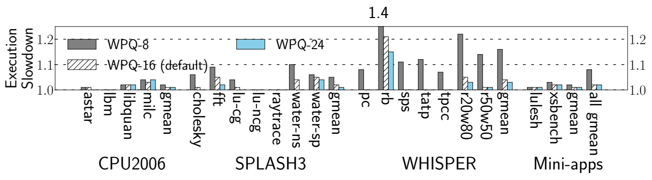


Figure 15: PPA's normalized slowdown varying WPQ size from 8 to 24; lower is better

To see the impact of the NVM write pending queue (WPQ) on the performance of PPA, we vary the WPQ size from 8 to 24 for memory-intensive applications of CPU2006/Mini-apps and multi-threaded applications. As shown in Figure 15, PPA still incurs a low overhead (8%) though the WPQ size decreases to 8. This is because many applications exhibit high L2 write miss rates indicating already high pressure on the WPQ for the baseline. As such, the negative effect of extra write traffic caused by PPA's store writeback is amortized. Note that PPA incurs a higher overhead for some applications, *e.g.,* `rb`, `water-ns`, and `water-sp`., as setting WPQ size to 8. The reason is two-fold: (1) they have low L2 miss rates indicating low run-time execution time for the baseline; (2) the store writeback leads to high pressure on WPQ due to more generated write traffic to it. Fortunately, the extra write traffic can be absorbed by enlarging the WPQ size to the default (16).

7.8 Sensitivity to PRF Size

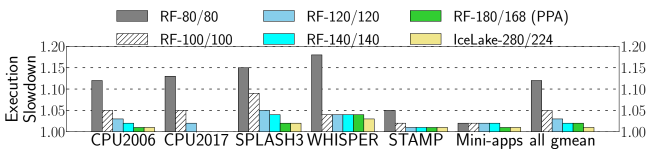


Figure 16: PPA's normalized slowdown varying RF sizes; lower is better

To show how PRF size affects PPA's performance, we vary the PRF size from 80/80 to 280/224 (integer/floating-point PR count). As shown in Figure 16, PPA incurs less overhead with a larger PRF.

Note that even with the smallest PRF size of 80/80, PPA still forms sufficiently long regions and thus incurs an average of only 12% overhead owing to the underutilization of the PRF size. Interestingly, the benefit of the large PRF diminishes once its size increases beyond the default. This is because the default PRF setting already has enough amount of free registers to form long regions covering the persistence latency. Notably, with PRF size 80/80, PPA incurs about 30% run-time overhead for some programs, *e.g.,*`hmmer`, `lbm`, `lu-cg`, and `tpcc`, since (1) PPA requires at least 65/68 integer/floating-point registers for their normal execution, and (2) the programs have intensive memory writes, ending up with putting high pressure on the PRF.

7.9 Sensitivity to CSQ Size

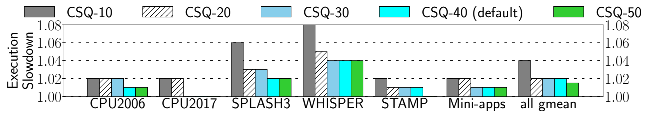


Figure 17: PPA's normalized slowdown with varying CSQ size; lower is better

To investigate the proper size of the CSQ, we vary the CSQ size from 10 to 50. As shown in Figure 17, the CSQ size has a minimal impact on PPA's performance since there are an average of only 18 stores in each region (see Figure 13). In light of this, we set the CSQ size to 40 by default such that the core pipeline encounters as less pipeline stalls as possible caused by the CSQ overflow; it is cheap to enlarge the size of the CSQ to 40 because of its simple structure.

7.10 Sensitivity to PMEM Write Bandwidth

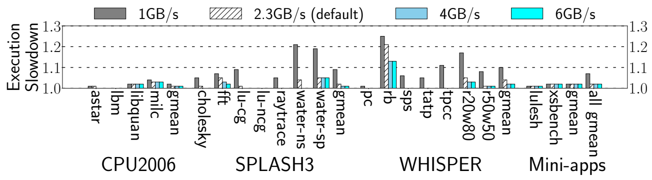


Figure 18: Normalized slowdown of PPA with varying NVM write bandwidth; lower is better

To show how PMEM write bandwidth affects PPA's performance, we vary the NVM write bandwidth from 1GB/s to 6GB/s for memory-intensive CPU2006/Mini-apps, SPLASH3, and WHISPER benchmarks. To be practical, PPA sets the default bandwidth to 2.3GB/s according to the empirical Intel PMEM analysis [148]. As shown in Figure 18, PPA still incurs an average of only 7% overhead even for 1GB/s write bandwidth. Once the write bandwidth goes up beyond the default, PPA keeps its performance overhead as low as 2% thanks to the long regions hiding the potential pipeline stalls upon full WPQ. It is worth noting that PPA incurs a relatively higher overhead for SPLASH and WHISPER programs with 1GB/s bandwidth. This is because different threads of these multi-threaded applications always compete for the shared WPQ and the lower bandwidth exacerbates the competition. Note that some applications, *e.g.,* `water-ns`, `water-sp`, and `rb`, are more sensitive to the write bandwidth due to their inherent less memory writeback traffic (*i.e.,* they exhibit high locality).

7.11 Sensitivity to Thread Count

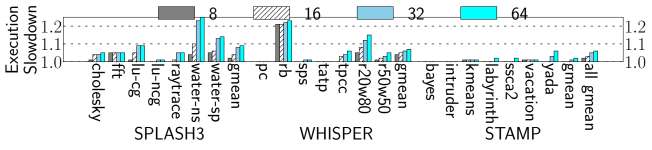


Figure 19: Normalized slowdown of PPA with varying thread count from 8 to 64 for multi-threaded apps; lower is better

To study the impact of PPA on cache coherence, we vary the thread count and scale up the NVM WPQ/shared L2 size proportionally. Figure 19 shows that the resulting performance impact is quite small; PPA still maintains high performance, *i.e.,* an average of 2%–6% overheads for 8–64 threads. PPA incurs slightly higher overheads for `water-ns`, `water-sp`, and Memcached (`r20w80`) with more threads due to the increasing stall cycles taken for thread synchronization.

7.12 Hardware Cost Analysis

PPA introduces a 64-bit LCPC register, a 348-bit vector register MaskReg due to the PRF size (348), and a 40-entry CSQ. Each CSQ entry records a pair of 9 ($\lceil log_2^{348} \rceil$)-bit index to a physical register and a 48-bit physical address. To facilitate JIT checkpointing, we round the size of PPA's proposed structures to the nearest multiple of 8 bytes such that their entry size is 8 bytes. We then use these numbers to calculate their hardware overheads (see Table 4).

Table 4: PPA's hardware overheads

	Area (μm^2)	Access Latency (ns)	Dynamic Access (pJ)
64-bit LCPC	12.20	0.057	0.00034
384-bit MaskReg	74.03	0.067	0.00029
40-entry CSQ	547.84	0.07	0.00025

We use CACTI 7.0 [104] to estimate the hardware cost of PPA's proposed hardware structures with a 22 nm process technology node. Table 4 showcases PPA's low hardware costs in terms of chip area, access latency, and power consumption. In summary, PPA's proposed hardware structures only occupy 0.005% chip area of an Intel Xeon server core (11.85 mm^2 after excluding its shared L2 cache); the core area size is calculated with McPAT [81].

7.13 Energy and Latency for JIT Checkpointing

Upon impending power loss, PPA checkpoints CSQ, LCPC, CRT, MaskReg, and a part of PRF marked by entries of CSQ or CRT in NVM. We assume 16 architectural integer registers and 32 architectural floating-point registers. Therefore, we need to checkpoint at most 88 physical registers (40 in CSQ and 48 in CRT).

Energy Consumption: We assume that the checkpointed hardware structures are based on SRAM. To estimate the energy consumption, we leverage prior work [4, 109, 117]. They measure the energy cost per memory operation by using an external power meter while executing carefully designed microbenchmarks. These microbenchmarks are used to observe the energy consumption of only data movement between core and memory and minimize the impact of other architectural optimizations and non-memory operations. It turns out that 11.839 nJ/byte is necessary for accessing data in SRAM cells and moving it from core to NVM. Therefore, we need to secure 21.7 μJ to JIT checkpoint 1838 bytes of data considering the worst case that each physical register has 128-bit data. However, the ideal PSP scheme BBB [4] and Intel's eADR require a supercapacitor of 775 μJ and 550 mJ, which are 36.5 and 25943x larger than ours, respectively.

Table 5: Comparison of Energy requirement for JIT flushing

	PPA (WSP)	Capri [57] (WSP)	LightPC [78] (PSP)
Energy Consumption	21.7μJ	0.6mJ	189mJ
Volume (SuperCap/Li-thin)	0.06 mm^3/0.0006 mm^3	1.57 mm^3/0.016 mm^3	527.8 mm^3/5.3 mm^3
Ratio to Core Size (SuperCap/Li-thin)	0.005/ 5×10^{-5}	0.14/0.0014	44.5/0.45

We leverage the prior work [4] to calculate the required size of supercapacitor [162] and Li-thin battery [119]. These two battery techniques have an energy density of $10^{-4}Wh/cm^3$ and $10^{-2}Wh/cm^3$, respectively. Table 5 shows that PPA needs a 0.06 mm^3 supercapacitor or a 0.0006 mm^3 Li-thin battery, which occupies 0.5%/0.0005% of an Intel server core (11.85 mm^2), respectively.

Checkpointing Time: PPA's JIT-checkpointing controller can persist 8 bytes of data per cycle thanks to its simple structure (see Section 4.5). According to our RTL synthesis results with TSMC 22 nm technology, the controller only requires 144 D flip-flops with 88 two-input logic gates. Therefore, the controller takes 114.9 ns to read 1838 bytes of data. Along with the write bandwidth (2.3GB/s) of PMEM [52] in our simulations, PPA needs only 0.91 μs to flush the 1838 bytes data to PMEM upon power failure.

Comparison of Energy Consumption: We calculate the energy consumption of a single core equipped with WSP Capri or PSP LightPC [78] to highlight the low energy requirement of PPA. Upon power failure, Capri flushes data in its battery-backed redo buffers (54KB per core) to NVM with 11.839 nJ per byte [4], thus costing 0.6mJ per core. Likewise, LightPC flushes volatile data of *only* user processes in architectural registers (4224 bytes of 16 GPRs and 32 XMM registers), L1D cache (64KB), and L2 cache (16MB) all the way to NVM, leading to a high energy consumption of 189 mJ; LightPC uses PCM as main memory.

8 RELATED WORK

Many prior PSP schemes [1, 4, 6, 9, 31, 37, 45, 51, 55, 64, 78, 85, 112, 115, 116, 122, 128, 146, 149, 150, 158] have offered user program persistency with crash consistency guaranteed. However, they require substantial programming burden in that users have to understand the underlying memory persistency model [73] and carefully write the code with crash consistency in mind. Moreover, the schemes often cause high run-time overhead (software approaches [140]) or significant logic complexity (hardware approaches [159]).

To this end, Narayanan *et al.* [107] propose the first WSP that flushes all volatile data, *e.g.,* architectural registers/caches/DRAM contents, to NAND flash storage upon an impending power outage. Unfortunately, the just-in-time (JIT) checkpointing of all the data requires a considerable amount of energy to be secured always, which is in need of an expensive uninterruptible power supply (UPS). To lower the energy consumption, Capri [57] proposes a crash consistency mechanism based on hardware-managed redo buffers that only require a capacitor for their JIT checkpointing. In particular, Capri compiler partitions the input program into a series of recoverable regions so that their stores never overflow the buffer. During the region execution, Capri persists the data being stored in the region by moving them from the redo buffer to NVM through the non-temporal path [24], bypassing the cache hierarchy completely. However, Capri still suffers expensive chip area/energy overheads due to per-core capacitor-backed redo buffer (each requiring 54 KB). On the other hand, ReplayCache [152], another WSP scheme for energy harvesting systems, incurs high run-time overhead with the frequent pipeline stalls at the end of compiler-formed store-integrity regions.

In summary, the overheads of the prior WSP schemes are so significant that they cannot enable a lightweight yet performant WSP. With the store integrity implemented using the simple register renaming trick, PPA achieves high-performance WSP for all at a negligible hardware cost. As shown in Table 6, PPA outperforms all prior WSP schemes in terms of all comparison criteria.

Table 6: Comparison of PPA to prior WSP approaches

	WSP [107]	Capri [57]	ReplayCache [152]	PPA
Hardware Complexity	Extremely High	High	No	Low
Energy Requirement	Extremely High	High	Low	Low
Recompilation	No	Yes	Yes	No
Transparency	Yes	Yes	Yes	Yes
Enable DRAM Cache	Yes	Yes	No	Yes
Enable Multi-MCs	Yes	No	Yes	Yes

9 CONCLUSION

This paper proposes PPA, the first microarchitectural approach to WSP. As a basis for crash consistency and lightweight WSP, PPA realizes so-called store integrity in the out-of-order core pipeline. That is, PPA prevents store registers from being overwritten and dynamically partitions program to a series of regions whose boundary is delineated when the physical register file runs out. Upon impending power failure, PPA checkpoints the minimal architectural states including the preserved store registers using a tiny capacitor. When power comes back, PPA restores the checkpointed states, replays (persists) the stores of the power-interrupted region, and resumes the program following the latest committed instruction before the failure. Experimental results with 41 applications highlight the benefits of PPA causing only a 2% average run-time overhead and 0.005% chip areal cost. We believe that PPA lays the foundation for WSP and pave the way to realizing it for all.

ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd for their valuable comments as well as Purdue CompArch research group members for early discussions on the project. We also appreciate Xianfei Jia helping with the RTL implementation of PPA's JIT checkpointing controller. This work was supported by NSF grants 2001124 (CAREER), 2153749, and 2314681.

REFERENCES

[1] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. 2022. ASAP: architecture support for asynchronous persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 306–319.

[2] Brian Aker. 2011. libMemcached - a open source C/C++ library for the memcached server. https://libmemcached.org/libMemcached.html.

[3] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[4] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 111–124.

[5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[6] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. 2022. NvMR: non-volatile memory renaming for intermittent computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 1–13.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[8] James Bucek, Klaus-Dieter Lange, et al. 2018. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 41–42.

[9] Miao Cai, Chance C Coats, and Jian Huang. 2020. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 584–596.

[10] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Commun. ACM* 51, 11 (2008), 40–46.

[11] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014), 433–452.

[12] CL Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information sciences* 275 (2014), 314–347.

[13] Yangyin Chen. 2020. ReRAM: History, status, and future. *IEEE Transactions on Electron Devices* 67, 4 (2020), 1420–1433.

[14] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 109–114.

[15] Jongouk Choi, Hyunwoo Joe, and Changhee Jung. 2022. CapOS: Capacitor Error Resilience for Energy Harvesting Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4539–4550.

[16] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. 2019. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 331–344.

[17] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 40–54.

[18] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 399–412.

[19] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2023. Write-Light Cache for Energy Harvesting Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[20] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[21] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[22] Intel Corporation. [n.d.]. Memory Optimized for Data-Centric Workloads. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[23] Intel Corporation. [n.d.]. Persistent memory programming. https://pmem.io.

[24] Intel Corporation. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual. (2023).

[25] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 271–282.

[26] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P Topham. 2000. Multiple-banked register file architectures. In *Proceedings of the 27th annual international symposium on Computer architecture*. 316–325.

[27] Marc De Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 140–151.

[28] Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–12.

[29] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–516.

[30] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[31] Alexander Freij, Huiyang Zhou, and Yan Solihin. 2023. SecPB: Architectures for Secure Non-Volatile Memory with Battery-Backed Persist Buffers. In *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA-29)*.

[32] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. 2018. Persistency for synchronization-free regions. *ACM SIGPLAN Notices* 53, 4 (2018), 46–61.

[33] Gonzalez Gonzalez, Adrián Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. 2004. A content aware integer register file organization. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 314.

[34] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[35] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A scalable and efficient persistent transactional memory. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 913–928.

[36] Yizi Gu, Yongpan Liu, Yiqun Wang, Hehe Li, and Huazhong Yang. 2016. NVPsim: A simulator for architecture explorations of nonvolatile processors. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 147–152.

[37] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 466–478.

[38] Linley Gwennap. 1994. MIPS R10000 uses decoupled architecture. *Microprocessor Report* 8, 14 (1994), 18–22.

[39] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[40] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. MOD: Minimally ordered durable data structures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 775–788.

[41] Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2021. A reusable characterization of the memory system behavior of SPEC2017 and SPEC2006. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 2 (2021), 1–20.

[42] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[43] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[44] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. 2001. The microarchitecture of the Pentium® 4 processor. In *Intel technology journal*. Citeseer.

[45] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. 2023. Zhuque: Failure is Not an Option,{it's} an Exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 833–849.

[46] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*. 468–482.

[47] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.

[48] Shao-Yu Huang, Jianping Zeng, Xuanliang Deng, Sen Wang, Ashrarul Haq Sifat, Burhanuddin Bharmal, Jiabin Huang, Ryan Williams, Haibo Zeng, and Changhee Jung. 2023. RTailor: Parameterizing Soft Error Resilience for Mixed-Criticality Real-Time Systems. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[49] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*.

[50] Intel. 2020. Intel Optane DC Persistent Memory Quick Start Guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-

Quick-Start-Guide.pdf.

[51] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 427–442.

[52] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[53] JEDEC. 2022. CXL Consortium and JEDEC Sign MOU Agreement to Advance DRAM and Persistent Memory Technology. https://www.jedec.org/news/pressreleases/cxl-consortium-and-jedec-sign-mou-agreement-advance-dram-and-persistent-memory.

[54] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 525–538.

[55] Jungi Jeong and Changhee Jung. 2021. PMEM-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–529.

[56] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 520–532.

[57] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and architecture support for whole-system persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 71–83.

[58] William M Jones, John T Daly, and Nathan DeBardeleben. 2012. Application monitoring and checkpointing in hpc: looking towards exascale systems. In *Proceedings of the 50th Annual Southeast Regional Conference*. 262–267.

[59] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*. 660–671.

[60] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 361–372.

[61] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.

[62] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 105–119.

[63] Ian Karlin, Jeff Keasler, and J Robert Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[64] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 613–626.

[65] Richard E Kessler. 1999. The alpha 21264 microprocessor. *IEEE micro* 19, 2 (1999), 24–36.

[66] AV Khvalkovskiy, Dmytro Apalkov, S Watts, Roman Chepulskii, RS Beach, Adrian Ong, X Tang, A Driskill-Smith, WH Butler, PB Visscher, et al. 2013. Basic principles of STT-MRAM cell operation in memory arrays. *Journal of Physics D: Applied Physics* 46, 7 (2013), 074001.

[67] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. Compiler-directed soft error resilience for lightweight GPU register file protection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 989–1004.

[68] Nam Sung Kim, Choungki Song, Woo Young Cho, Jian Huang, and Myoungsoo Jung. 2019. LL-PCM: Low-latency phase change memory architecture. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[69] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 424–439.

[70] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. ClfB-Tree: Cacheline Friendly Persistent B-Tree for NVRAM. *ACM Trans. Storage*, Article 5 (2018), 17 pages.

[71] Martin Kleppmann. 2017. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.".

[72] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 399–411.

[73] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. 2016. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[74] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. 2018. Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 315–327.

[75] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 335–349.

[76] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[77] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[78] Sangwon Lee, Miryeong Kwon, Gyuyoung Park, and Myoungsoo Jung. 2022. LightPC: hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 289–305.

[79] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*.

[80] Lin Li, Vijay Degalahal, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. 2004. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*. 132–137.

[81] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 469–480.

[82] Ankur Limaye and Tosiron Adegbija. 2018. A workload characterization of the SPEC CPU2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 149–158.

[83] Mikko H Lipasti, Brian R Mestan, and Erika Gunadi. 2004. Physical register inlining. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 325.

[84] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017), 329–343.

[85] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for non-volatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 258–270.

[86] Qingrui Liu and Changhee Jung. 2016. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[87] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Notices* 50, 5 (2015), 1–10.

[88] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 228–239.

[89] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 25.

[90] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2017. Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2017), 32.

[91] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 487–502.

[92] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1187–1202.

[93] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 411–425.

[94] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, et al. 2015. Ambient energy harvesting nonvolatile processors: from circuit to system. In *Proceedings*

of the 52nd Annual Design Automation Conference. 1–6.

[95] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 526–537.

[96] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[97] Chris Mellor. 2022. CXL-led big memory taking over from age of SAN. https://blocksandfiles.com/2022/06/20/cxl-led-big-memory/.

[98] Chris Mellor. 2022. Redis is ready for CXL memory pooling. https://blocksandfiles.com/2022/07/20/redis-cxl-memory-pooling/.

[99] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 413–422.

[100] memcached organization. 2017. memcachd - a distributed memory object caching system. http://memcached.org/.

[101] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*. 69–80.

[102] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. 1999. Ilp versus tlp on smt. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. 37–es.

[103] Alessandro Morari, Carlos Boneti, Francisco J Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose, and Mateo Valero. 2012. SMT malleability in IBM POWER5 and POWER6 processors. *IEEE Trans. Comput.* 62, 4 (2012), 813–826.

[104] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 1 (2009), 1–24.

[105] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. *ACM SIGPLAN Notices* 52, 4 (2017), 135–148.

[106] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*.

[107] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 401–410.

[108] Vignyan Reddy Kothinti Naresh, David J Palframan, and Mikko H Lipasti. 2011. CRAM: Coded registers for amplified multiporting. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 196–205.

[109] Debasish Nayak, Debiprasad Priyabrata Acharya, and Kamalakanta Mahapatra. 2016. An improved energy efficient SRAM cell for access over a wide frequency range. *Solid-State Electronics* 126 (2016), 14–22.

[110] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–414.

[111] Tri M Nguyen and David Wentzlaff. 2018. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 507–519.

[112] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L Miller. 2019. Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 836–848.

[113] Bogdan Nicolae, Adam Moody, Gregory Kosinovsky, Kathryn Mohror, and Franck Cappello. 2021. Veloc: Very low overhead checkpointing in the age of exascale. *arXiv preprint arXiv:2103.02131* (2021).

[114] Byoungchan Oh, Nilmini Abeyratne, Nam Sung Kim, Jeongseob Ahn, Ronald G Dreslinski, and Trevor Mudge. 2022. Rethinking DRAM's page mode with STT-MRAM. *IEEE Trans. Comput.* (2022).

[115] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 142–156.

[116] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2023. Scoped Buffered Persistency Model for GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 688–701.

[117] Dhinakaran Pandiyan and Carole-Jean Wu. 2014. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 171–180.

[118] Kimish Patel, Wonbok Lee, and Massoud Pedram. 2007. Active bank switching for temperature control of the register file in a microprocessor. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*. 231–234.

[119] David Pech, Magali Brunet, Hugo Durou, Peihua Huang, Vadym Mochalin, Yury Gogotsi, Pierre-Louis Taberna, and Patrice Simon. 2010. Ultrahigh-power micrometre-sized supercapacitors based on onion-like carbon. *Nature nanotechnology* 5, 9 (2010), 651–654.

[120] Keni Qiu, Mengying Zhao, Zhenge Jia, Jingtong Hu, Chun Jason Xue, Kaisheng Ma, Xueqing Li, Yongpan Liu, and Vijaykrishnan Narayanan. 2020. Design insights of non-volatile processors and accelerators in energy harvesting systems. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 369–374.

[121] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. 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, 14–23.

[122] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 672–685.

[123] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 101–111.

[124] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. *Adms@ Vldb* 15 (2015), 61–72.

[125] Dipanjan Sengupta, Qi Wang, Haris Volos, Ludmila Cherkasova, Jun Li, Guilherme Magalhaes, and Karsten Schwan. 2015. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 317–320.

[126] Sophiane Senni, Lionel Torres, Gilles Sassatelli, and Abdoulaye Gamatie. 2016. Non-volatile processor based on MRAM for ultra-low-power IoT devices. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 2 (2016), 1–23.

[127] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[128] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-less Persist Ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 539–554.

[129] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*. 141–152.

[130] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–190.

[131] Dezso Sima. 2000. The design space of register renaming techniques. *IEEE micro* 20, 5 (2000), 70–83.

[132] CORPORATE SPARC International, Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc.

[133] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. 2016. A Ferroelectric Nonvolatile Processor with 46mus System-Level Wake-up Time and 14mus Sleep Time for Energy Harvesting Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 3 (2016), 596–607.

[134] Kun Suo, Yong Shi, Chih-Cheng Hung, and Patrick Bobbie. 2021. Quantifying context switch overhead of artificial intelligence workloads on the cloud and edges. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1182–1189.

[135] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[136] Nathan Tuck and Dean M Tullsen. 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 26–34.

[137] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 392–403.

[138] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. 1999. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 54–58.

[139] Vineet Veer Tyagi and DPCM Buddhi. 2007. PCM thermal storage in buildings: A state of art. *Renewable and sustainable energy reviews* 11, 6 (2007), 1146–1166.

[140] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[141] Daniel Waddington and Jim Harris. 2018. Software challenges for the changing storage landscape. *Commun. ACM* 61, 11 (2018), 136–145.

[142] Wei Wang and Tanima Dey. 2011. A survey on arm cortex a processors. *Retrieved March* (2011).

[143] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. 2012. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *2012 Proceedings of the ESSCIRC (ESSCIRC)*. IEEE, 149–152.

[144] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[145] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. When storage response time catches up with overall context switch overhead, what is next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4266–4277.

[146] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 346–359.

[147] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A Speculative Approach to Persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 892–907.

[148] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[149] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Hai Jin, Xiaofei Liao, and Yan Solihin. 2022. Preserving Addressability Upon GC-Triggered Data Movements on Non-Volatile Memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–26.

[150] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. 2023. SpecPMT: Speculative Logging for Resolving Crash Consistency Overhead of Persistent Memory. (2023).

[151] Mao Ye, Clayton Hughes, and Amro Awad. 2018. *Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[152] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 170–182.

[153] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. 2021. Turnpike: Lightweight Soft Error Resilience for In-Order Cores. In *The 54th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press.

[154] Jin Zha, Linpeng Huang, Linzhu Wu, Sheng-an Zheng, and Hao Liu. 2016. A consistency mechanism for NVM-Based in-memory file systems. In *Proceedings of the ACM International Conference on Computing Frontiers*. 197–204.

[155] Yida Zhang and Changhee Jung. 2022. Featherweight soft error resilience for GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–262.

[156] Yilin Zhang and Wei-Ming Lin. 2016. Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors. *Microprocessors and Microsystems* 45 (2016), 270–282.

[157] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, Vol. 5.

[158] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 421–432.

[159] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 421–432.

[160] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. 2012. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 1–6.

[161] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. 2023. SweepCache: Intermittence-Aware Cache on the Cheap. In *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*.

[162] Yanwu Zhu, Shanthi Murali, Meryl D Stoller, KJ Ganesh, Weiwei Cai, Paulo J Ferreira, Adam Pirkle, Robert M Wallace, Katie A Cychosz, Matthias Thommes, et al. 2011. Carbon-based supercapacitors produced by activation of graphene. *science* 332, 6037 (2011), 1537–1541.