Persistent Processor Architecture

Jianping Zeng
Purdue University

Jungi Jeong
Purdue University

Changhee Jung
Purdue University

1 INTRODUCTION AND MOTIVATION

Nonvolatile memory (NVM) technologies such as ReRAM, 3D XPoint, and STT-MRAM have emerged as alternatives to DRAM. Thanks to their byte-addressability, low cost per bit, and in-memory persistence, they are to be used as nonvolatile main memory (NVMM)—also known as persistent memory. 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.

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) 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 in this mode.* Although PMEM offers *app-direct mode* where DRAM is used as main memory and PMEM serves as persistent heap, 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, users must rewrite their program with crash consistency and memory persistency in mind, and often devise application-specific recovery code tailored to the data structures. Besides, PSP requires dedicated PMEM allocation interfaces such as `pmalloc`, rendering already error-prone persistent programming more complex. Importantly, the resulting persistent program is slower than the original one due to the undo/redo logging involving persistence barriers.

Given the limitations of PSP and the demand for transparent use of PMEM without sacrificing the in-memory persistence and crash consistency, this paper presents *Persistent Processor architecture* (PPA). PPA is the first of its kind to realize transparent, lightweight, and performant whole-system persistence (WSP) [2] 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 on power back.

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 [3]. (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 for the replay and the resumption in the wake of the outage.

2 PPA OVERVIEW

Figure 1 depicts how PPA leverages ample physical registers in out-of-order cores to preserve store registers, thus achieving WSP. 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.

3 DYNAMIC REGION FORMATION

In contrast with prior region-level persistence [3], PPA builds regions dynamically without user intervention, recompilation, and significant performance loss. PPA 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). Figure 1 shows that 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.

4 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 1), 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 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 performance due to long enough regions and thus causing negligible stalls at region boundaries.

5 STORES INTEGRITY ENFORCEMENT

Figure 1 shows how PPA ensures store integrity on the fly during the pipeline execution. Upon retiring *str r*0, [100] (○ in the figure)

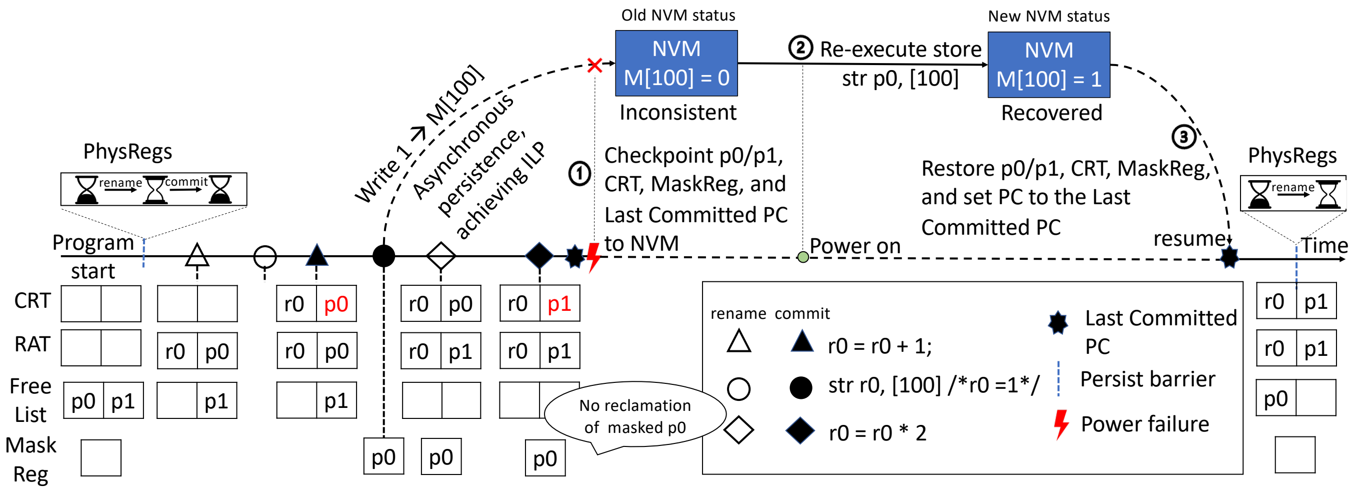

Figure 1: PPA overview; for store integrity, $p0$ is not recycled even after the multiplication commits

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 [3], thus hiding the store persistence latency.

6 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 1 (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 [2] and eADR's demand a significantly large bulky Li-thin battery or supercapacitor. When the power comes back, PPA first replays all committed stores behind the failure, *e.g., str* $p0$, $[100]$ in Figure 1 (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.*

7 EVALUATION

We use the cycle-accurate simulator gem5 to model an 8-core x86_64 Skylake-X processor with *two integrated memory controllers*, each

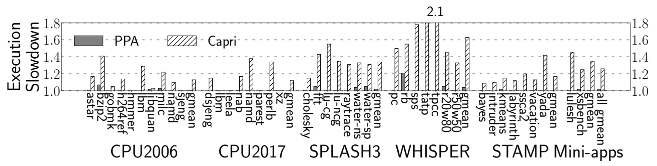

Figure 2: Normalized slowdown of PPA to PMEM's memory mode; lower is better

of which manages a DRAM as an off-chip direct-mapped cache as with PMEM's memory mode. As shown in Figure 2, PPA incurs an average of 2% overhead, while the state-of-the-art work Capri [1] incurs a 26% overhead even with complicated hardware design, *e.g.,* 54KB battery-backed hardware buffers.

8 CONCLUSION

This paper proposes PPA, the first microarchitectural approach to WSP. As a basis for crash consistency and lightweight WSP, PPA realizes region-level store integrity in the out-of-order core pipeline. 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.

REFERENCES

[1] 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.
[2] 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.
[3] 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.
[4] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture.* 1075–1091.