Capri: Compiler and Architecture Support for Whole-System Persistence

Jungi Jeong[§], Jianping Zeng and Changhee Jung
Purdue University

I. Introduction

Advanced non-volatile memory (NVM) technologies, such as Intel's Optane PMem, provide both high-density and in-memory persistence, realizing the full potential to unify the main memory and storage devices. This leads to the advent of persistent memory programming for so-called partial-system persistence (PSP), e.g., Optane's *app-direct mode*, where DRAM is used as the main memory while NVM serves as a persistent heap. In PSP, programmers must delineate a piece of code which requires crash consistency and explicitly manage both volatile and non-volatile data (objects) using dedicated interfaces such as `pmalloc` or persistent transactions [1]. However, the persistent memory programming is difficult and often necessitates custom data structure design and application-specific recovery code tailored to particular data structures, thus being limited to a small set of programs such as in-memory index structures/databases or key-value stores [2], [3].

II. Motivation

Unfortunately, this limitation hinders most users from readily taking advantage of both high-density and in-memory persistence of NVM. With that in mind, as an alternative to the *app-direct mode*, Intel proposes a *memory mode*, where DRAM is vertically integrated as a cache on top of NVM. In particular, since NVM here is used as the high-density yet volatile main memory, it does not provide in-memory persistence at all. This implies that in the memory mode, users have no choice but to put up with data loss in case of power failure, unless they resort to the app-direct mode at the expense of the persistent memory programming difficulty.

To solve the problems, this paper studies whole-system persistence (WSP) that simultaneously enables high-density and in-memory persistence and satisfies the following requirements. First, WSP must be able to restore the entire system on failure recovery *no matter how deep the volatile cache and memory hierarchy is* as in off-chip DRAM cache as Optane's *memory mode*. Second, failure recovery should be offered *to any programs* (instead of being limited to in-memory databases and key-value stores) *in a software-transparent manner*, which is desired as a variety of recent works confirm that persistent programming is error-prone [4].

A. Limitation of state-of-the-art approaches

The key approach to achieving whole-system persistence is to flush all data in volatile media (e.g., register files, CPU caches, and DRAM) into non-volatile memory before the impending power failure. For example, Narayanan et al. proposed to use residual energy and persist all volatile status when power is about to be cut off [5]. Similarly, Intel recently announced extended ADR (eADR) support that includes on-chip caches in the persistent domain [6]. However, it turns out that eADR must secure an excessive amount of residual energy to persist the deep cache hierarchy of HPC manycore processors, which gets worse for the off-chip DRAM cache as in the memory mode of Intel Optane. Apart from that, eADR does not protect other volatile states such as register files and internal buffers in the processor pipeline. This limitation makes it practically impossible to realize whole-system persistence at low cost.

B. Overall Design

To this end, this paper proposes Capri, a compiler and architecture co-design scheme that achieves *region-level whole-system persistence*. Capri partitions programs into a series of regions, where each region boundary serves as a recovery point. For this purpose, the Capri compiler inserts region boundary instructions to delineate the *region formation*. Furthermore, it instruments register-checkpointing instructions required for resuming the power-interrupted programs from the last committed and persisted region boundaries after failure recovery. Then, all store instructions within a region are carefully handled by hardware, which leverages the *hardware non-volatile proxy buffer* as the safety net to prevent partial updates. The proxy buffer guarantees that modified data in a region are not released to the non-volatile main memory before the region is completed.

Capri's region formation and proxy buffer showcase a novel interplay between compiler and architecture, enabling whole-system persistence *without program changes* while *simultaneously leveraging high-density and in-memory persistence*. That is, the Capri compiler uses the number of stores as a region criterion, i.e., the resulting regions contain no more than the threshold number of stores therein (e.g., 256 by default, including both regular and checkpointing stores). This threshold determines the hardware proxy buffer size and prevents overflow, rendering hardware design simpler.

Challenge #1: Stale Read Prevention. Previous studies with two data paths to NVM (e.g., the regular data path through caches and the new persist path from proxy buffers) have decided to drop dirty cache blocks on their last-level cache eviction (from the regular path) to simplify the sequential persist order [7]–[10]. However, such a design decision complicates the NVM read operations—whenever searching data in NVM,

[§]Now at Google.

it must look up the proxy buffer simultaneously, increasing both read latency and energy consumption. Such *indirect read* has been mitigated in various ways, e.g., HW bloom filter [7], [8], cache coherence [10], or speculation [9], but they all come at the cost of significant hardware and software complexity.

Solution #1: Undo+Redo Logging. Capri eliminates the indirect read problem by allowing NVM updates from the *both* dirty cache writeback and the proxy buffer. Therefore, memory loads and stores happen in the same way as the commodity architecture. However, dirty cache writeback may break correctness by persisting regions out of order. To preserve crash consistency with this distinctive architecture, Capri uses *undo+redo logging* that keeps data of both before and after the update. Thus, even if the dirty cache writeback persists regions out of order, the undo value (e.g., before the update) can safely restore the previous state across a power failure.

Challenge #2: Checkpoint Overheads. By its nature, whole-system persistence (in line with persistent memory programs [7], [9]–[11]) should come with performance overheads—compared to volatile execution—and complex hardware changes to control the persist order. For example, with WSP, all store instructions in a region must be reflected inside the non-volatile main memory before proceeding to the next region. Furthermore, register-checkpointing stores incur non-negligible pressure to NVM, leading to a substantial slowdown. Therefore, the primary design goal of Capri is to *lengthen the region size* as much as possible to lessen checkpointing stores. The longer regions are desirable since they reduce the number of checkpointing stores and unburden the pipeline by less dynamic instruction counts.

Solution #2: Compiler Optimizations. Capri reduces checkpoint overheads via 1) region size extension and 2) unnecessary checkpoints removal. First, we found that although a large number (e.g., 1k stores) is given as a threshold, many of the resulting regions contain fewer stores due to *short loops* in programs. In light of this, Capri presents *speculative loop unrolling* that unrolls the loop even if iteration counts are unavailable at compile time. In this way, the Capri compiler significantly extends the region sizes for short loops in programs. Second, Capri leverages existing compiler analysis to reduce checkpoint overheads further [12], i.e., it removes register-checkpointing stores if their register values can be reconstructed by other register values at recovery time. Finally, Capri rearranges checkpointing stores to prevent repeated checkpoints of the same register inside the loop body.

III. Experimental Methodology and Evaluation Results

For evaluation, we used the full-system simulation mode of a cycle-accurate architecture simulator (gem5). Notably, we re-compiled the entire Linux Kernel with our Capri compiler to include the OS in the whole-system persistence domain. Our experiments demonstrate that Capri accomplishes lightweight whole-system persistence only causing 0%, 12.4%, and 9.1% performance overheads in a geometric mean for SPEC2017, STAMP, and Splash3 benchmarks, respectively. Although a naive approach may slow down the benchmark up to 2X, our novel architecture and compiler interaction achieves very low performance overheads. Consequently, Capri makes it possible to accommodate all programs as first-class citizen in the world of persistence.

IV. Conclusion

To leverage both high-density and in-memory persistence benefits of NVM, the users of Intel Optane are forced to select the app-direct mode over the memory mode. As a result, only a handful of applications can resort to the both benefits at the expense of persistent programming difficulty. To address the limitation, this paper introduced Capri, a compiler/architecture co-design scheme for region-level whole-system persistence. Unlike partial-system persistence, Capri makes any programs failure-atomic without source code change while letting them enjoy both high-density and in-memory persistence simultaneously. This guarantee is particularly promising for the Optane users since Capri can free them from all the headaches of persistent programming including notorious crash consistency bugs. To achieve this, the Capri compiler generates recoverable regions while Capri architecture guarantees their execution to be crash-consistent. Furthermore, Capri's compiler and architecture optimizations enable lightweight whole-system persistence (5.1% average slowdown), thereby offering all programs high-performance persistence with increased memory space.

References

[1] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.

[2] "Pmem redis." https://github.com/pmem/redis/tree/3.2-nvml.

[3] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory," in *HotStorage*, 2017.

[4] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *ASPLOS*, 2019.

[5] D. Narayanan and O. Hodson, "Whole-system persistence with non-volatile memories," in *ASPLOS*, 2012.

[6] "eadr: New opportunities for persistent memory applications." https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html.

[7] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS*, 2017.

[8] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *MICRO*, 2018.

[9] J. Jeong and C. Jung, "Pmem-spec: Persistent memory speculation (strict persistency can trump relaxed persistency)," in *ASPLOS*, 2021.

[10] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *MICRO*, 2016.

[11] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *ISCA*, 2020.

[12] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight gpu register file protection," in *PLDI*, 2020.