SoftPM: Software Persistent Memory

Yuanchao Xu*†, Wei Xu†, Kimberly Keeton†, David E. Culler†
*North Carolina State University, †Google

***Abstract*—Hardware persistent memory (HardPM) offers a promising alternative to DRAM, but the mass adoption necessary to realize its cost advantages remains elusive, especially without broad application demand. An alternative, long-understood approach to fast persistence is to utilize the battery-backed DRAM that is deployed in hyperscalar data centers [5]. We present SoftPM, a Software Persistent Memory design that manages vulnerable DRAM-resident updates to ensure that data is persisted in the event of a power outage. SoftPM supports a user-directed mode by leveraging application persistency models (e.g., logging), a transparent mode that relies on kernel page fault support, and an explicit model that gives the application direct control over persistence. Our implementation significantly outperforms HardPM and hybrid HardPM-DRAM alternatives. By providing a general-purpose solution that leverages existing infrastructure, we hope to spur wider adoption of fast local persistence.**

I. Introduction

Hardware persistent memory (HardPM) has long been proffered as a promising supplement or substitute to DRAM, offering higher density, better scaling potential, lower idle power, and non-volatility, while retaining byte addressability [1]. A rich set of studies explores how to better utilize HardPM or integrate it into existing systems to improve performance, including crash-consistent, cache-friendly, concurrent data structures [3], [9]; transaction logging designs [6]; system designs [4], [8], [10]; etc. To leverage HardPM, most of these designs are tailored to overcome its high latency, low bandwidth, and asymmetric read/write performance. Some augment the system with hybrid HardPM-DRAM to provide closer-to-DRAM performance plus persistence.

Large-scale well-engineered data centers have used battery backup for decades. After a power outage, the battery provides power for the data center to save the current execution state, e.g., flushing dirty DRAM content into nonvolatile storage. Battery-backed DRAM has clear advantages over HardPM. It has real DRAM performance, while HardPM only has one-third of DRAM's peak read bandwidth, one-fifth of its peak write bandwidth, and three times the latency. Without these performance obstacles, a battery-backed DRAM approach may not need (or benefit from) the complexity of HardPM-motivated software optimizations. Reusing HardPM achievements, but removing unnecessary complexity, leads to a cleaner system design, providing substantial performance improvement over potential HardPM designs.

However, battery-backed DRAM has an inherent size constraint. A battery has limited capacity, enabling the data center machines to run for a limited time, perhaps minutes, after a power outage. The dirty content in whatever DRAM is used to implement the *persistence domain* must be savable to some nonvolatile or remote media within this time, and that process has finite bandwidth. We define this bandwidth-time product as the *save window* and the portion of the persistence domain that has not been committed to some truly persistent media the *vulnerable subdomain*. A SoftPM system must be designed so that the size of the vulnerable subdomain never exceeds the capacity of the save window. Prior NVDRAM work [5], [7] explored this question from the temporal and spatial locality perspective. However, they have two inefficiencies: (1) they overestimate the vulnerable subdomain, as it assumes all dirty pages are related to application persistency. (2) Without reusing and reexamining HardPM optimizations, their work does not distinguish useful HardPM optimizations and unnecessary HardPM optimizations, leading to the underutilization of NVDRAM.

We explore the design and implementation of a general-purpose Software Persistent Memory (SoftPM) system, managing the limited vulnerable subdomain by leveraging application persistency models. First, we investigate application persistency models and divide them into three categories: static, append-only logs, and large in-memory persistency. Static and append-only logs are two typical persistency models in in-memory databases. These applications declare a small region in NVDRAM or DRAM as write buffers for logs. Most HardPM studies assume a large in-memory persistency model, in which applications perform in-place updates and reduce or even remove logs to improve performance. Second, we design and implement three management schemes (transparent, explicit, and user-directed) with simple userspace interfaces that pair with the three persistency models. Our user-directed design leverages the read-only hints from applications to asynchronously flush logs into storage to limit log size within the vulnerable subdomain. Our transparent design recognizes dirty pages and limits total dirty size within the vulnerable subdomain by actively flushing pages that are unlikely to be written to storage. Applications can be ported to SoftPM systems with a few code changes (100 lines of code changes in Redis [2]). Third, we re-evaluate HardPM studies to provide HardPM insights from a SoftPM perspective. We also evaluate simplified designs by removing unnecessary optimizations. Results show that the throughput of SoftPM Redis is 26%-107% higher than the Hybrid design in various YCSB workloads.

II. Design

Figure 1 provides an overview of our user-directed SoftPM design and the transparent SoftPM design. An application creates/registers a region of address space belonging to the persistence domain through a system call.

An application using the append-only log persistency model allocates log entries in DRAM and appends log records for write requests into the allocated entries. In the steady state, the application ensures enough older log entries reach storage through fsync before handling new requests. If a crash happens, the application replays these logs on an early version in storage to reach a consistent data state.

Figure 1 (a) shows the user-directed SoftPM design and how it naturally fits this model. In user-directed SoftPM design, we

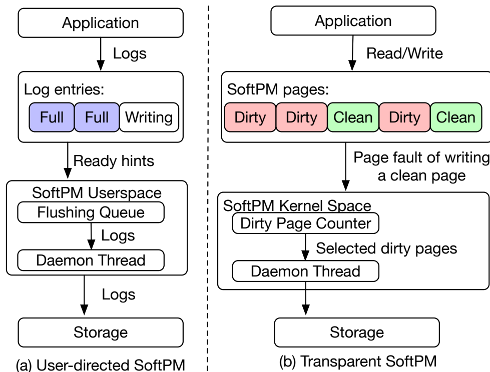

Fig. 1. (a) User-directed SoftPM and (b) Transparent SoftPM

simply place log entries in the mmapped SoftPM region, and SoftPM manages this region by asynchronous flushing while preserving the same crash-consistency requirements. We leverage that a fully filled log entry will not be overwritten again by the application. We provide an interface for the application to notify SoftPM that a log entry is complete and ready to be written back to storage by the daemon thread. The SoftPM system ensures that it will occur even in the advent of a power failure (or upon reboot on a non-power-related crash).

Figure 1 (b) shows the transparent SoftPM design for large in-memory persistency model, as used in most HardPM studies. SoftPM sets write-protected bits for all pages of mmapped SoftPM region. An application write to a clean page leads to a page fault. We implement a SoftPM page fault handler in the kernel to unset the write-protected bit and increase the dirty page counters. But, if this event would cause the vulnerable subdomain to exceed the save-window, it delays resumption till the constraint clears by using a kernel daemon thread to flush a selected dirty page in the foreground. This daemon thread also proactively performs background flushing of selected dirty pages into the storage to maintain the save-window capacity requirement and reduce the need for foreground flushes.

III. Preliminary Results

We perform our experiment on a two-socket Xeon server equipped with a 28-core Intel Xeon Platinum 8273CL@2.20GHz. Each socket is populated with six 32-GiB 2666MHz DDR4 DRAM DIMMs and six 512 GiB Optane DC DIMMs. The machine is equipped with a 4 TB SSD. All client threads and the server thread are running on different cores in one socket.

Table I summarizes the persistence options we evaluate. We use Redis with an append-only log persistency model, which batches updates for a second and then appends them to a (persistent) file. The SSD 1s and the SSD record represent per-second and per-record frequencies to flush and fsync() updates to the file. SoftPM-T represents Redis with transparent SoftPM, and SoftPM-U represents Redis with user-directed SoftPM.

Figure 2 (a) shows that SoftPM schemes outperform HardPM and Hybrid schemes. In the write-heavy workload (YCSB A), the SoftPM-T scheme is 45%-107% faster than the HardPM scheme and is 45%-84% faster than the Hybrid. In the mostly read workload (YCSB B), the SoftPM-T scheme is 30%-69% faster than HardPM and is 26%-49% faster than the Hybrid. The

TABLE I
Persistency and Optimizations of 6 Redis Schemes

	Persistent logic		Data locations			
Schemes	Logging	Persist logs	Main data	Log	Pers. logs	Pers. data
SSD 1s	Yes	Yes	DRAM	DRAM	SSD	
SSD record	Yes	Yes	DRAM	DRAM	SSD	
Hybrid	Yes	No	DRAM	HardPM	HardPM	
HardPM	No	No	HardPM			HardPM
SoftPM-T	No	No	SoftPM			SSD
SoftPM-U	Yes	Yes	SoftPM	SoftPM	SSD	

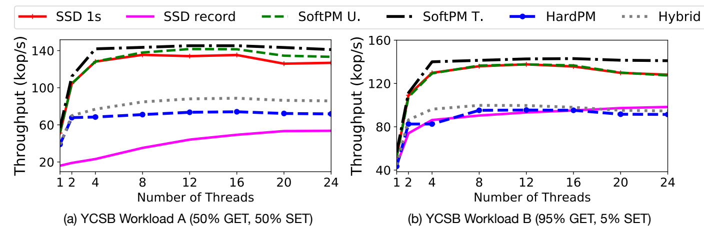

Fig. 2. (a) YCSB workload A and (b) YCSB workload B

Hybrid scheme puts indexing data structures and keys in DRAM, but it does not fully mitigate the low performance of HardPM.

IV. Conclusion and Future Work

Instead of waiting for an industry-wide transformation of deploying HardPM (e.g., hardware, system software, applications, etc.), SoftPM is potentially available using technology already provisioned in data centers today. SoftPM also sets the performance bar that future HardPM and its ecosystem will need to beat. SoftPM also points out that it is important to pursue research using this foundation rather than just overcoming that peculiarities of current HardPM technology.

References

[1] "Intel optane persistent memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, online; accessed February, 2022.

[2] "Redis," https://redis.io/, online; accessed February, 2022.

[3] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[4] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database system," in *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. ACM, 2017.

[5] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 613–626, 2017.

[6] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 385–398, 2016.

[7] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 401–410.

[8] J. Shi, "Exadata with persistent memory: An epic journey." https://www.snia.org/sites/default/files/PM-Summit/2020/presentations/11_PMEM_Jia_Shi_final_PM_Summit_2020_v2.pdf, online; accessed February, 2022.

[9] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, 2015, pp. 167–181.

[10] W. Zhang, S. Shenker, and I. Zhang, "Persistent state machines for recoverable in-memory storage systems with {NVRam}," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1029–1046.