SPMS: Strand based Persistent Memory System

Shuo Li*, Peng Wang†, Nong Xiao‡, Guangyu Sun† and Fang Liu*
*State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Chi
†Center for Energy-efficient Computing and Applications, Peking University, China
‡School of Data and Computer Science, Sun Yat-sen University, China
lishuo12@nudt.edu.cn wang_peng@pku.edu.cn xiaon6@sysu.edu.cn gsun@pku.edu.cn liufang@nudt.edu.cn

***Abstract*—Emerging non-volatile memories enable persistent memory, which offers the opportunity to directly access persistent data structures residing in main memory. In order to keep persistent data consistent in case of system failures, most prior work relies on persist ordering constraints which incurs significant overheads. Strand persistency minimizes persist ordering constraints. However, there is still no proposed persistent memory design based on strand persistency due to its implementation complexity. In this work, we propose a novel persistent memory system based on strand persistency, called SPMS. SPMS consists of cacheline-based strand group tracking components, a volatile strand buffer and ultra-capacitors incorporated in persistent memory modules. SPMS can track each strand and guarantee its atomicity. In case of system failures, committed strands buffered in the strand buffer can be flushed back to persistent memory within the residual energy window provided by the ultra-capacitors. Our evaluations show that SPMS outperforms the state-of-the-art persistent memory system by 6.6% and has slightly better performance than the baseline without any consistency guarantee. What's more, SPMS reduces the persistent memory write traffic by 30%, with the help of the strand buffer.**

I. Introduction

Emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM), spin-transfer torque RAM (STT-MRAM) and resistive memory (ReRAM), offer both the durability of disks/SSDs and the byte-addressability of DRAM. NVM-based persistent memory systems give applications the opportunity to directly access persistent data in main memory through load/store instructions. However, similar to disks/SSDs, persistent memory systems should ensure crash consistency [1], [2], that is the persistent data structures remain in consistent states after systems restart in the event of system failures (e.g., power losses and system crashes).

Crash consistency has been well investigated in database systems and file systems. Write-ahead logging (WAL) and copy-on-write (CoW) are two main mechanisms for ensuring crash consistency. NV-heaps [3] and Mnemosyne [4] enforce atomic transactional updates by maintaining a redo log. BPFS [5] adopts short-circuit shadow paging mechanism, a variant of CoW, to perform consistent updates. Unfortunately, both mechanisms incur significant overheads. In WAL, logs consisting of both data and corresponding metadata like the address of the data, introduce a large amount of extra memory write traffic. Unavoidably, CoW need to copy unmodified data and thus incurs unnecessary write traffic and consumes extra intra bandwidth, especially when updates are sparse.

Prior work reaches a consensus constraining persist ordering is the main mechanism to ensure consistency [5], [6], [7], [8]. However, persist ordering constraints induce significant overheads. In order to enforce persist ordering, applications usually need to explicitly execute costly cache flush (e.g., clflush, clflushopt and clwb, memory fence (e.g., mfence and sfence) and pcommit [9] instructions [8]. What's more, persist ordering constraints eliminate the opportunity to reorder or coalesce memory references to improve system performance.

Motivated by memory consistency, Pelly et al. [6] introduced *memory persistency* to describe persist ordering constraints and proposed three persistency models: strict persistency, epoch persistency and strand persistency. Specifically, strand persistency eliminates all unnecessary ordering constraints and exposes the maximum persist concurrency. From programs' point of view, a *strand* is a standalone logic task, such as inserting an entry into a queue [6], inserting/deleting a key-value pair into/from a hash table [7] and so on. Strands are independent of each other from perspective of persistency and can be written back to persistent memory concurrently. The only remaining persist ordering constraints specified by *persist barriers* lie in individual strands. Similar to a memory barrier, any persists after a persist barrier cannot proceed ahead of any persists before the persist barrier. Although strand persistency minimizes persist ordering constraints, however, due to its implementation complexity, there is still no proposed persistent memory system based on strand persistency.

In this work, we propose a novel strand based persistent memory system, called SPMS, which aims to maximize performance of persistent memory systems with crash consistency guarantees. SPMS can keep the persistent memory system in consistent states in the event of system failures and eliminate the recovery phase when the system restarts after crashes. Our evaluations show that SPMS outperforms the state-of-the-art persistent memory design by 6.6% and has slightly higher performance than the baseline system without crash consistency guarantees. What's more, the write traffic to persistent memory is also reduced by 30% compared to the baseline. The major contributions of this paper are as follows:

- We propose a novel persistent memory design based on strand persistency.
- We introduce a cacheline-based strand group tracking technique, which can track strands without sacrificing cache efficiency.
- By guaranteeing the atomicity of strands, we eliminate persist barriers within strands and maximize the performance of persistent memory systems.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

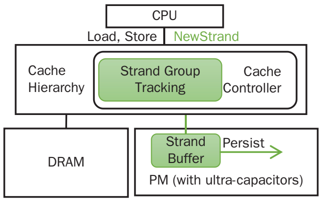


Fig. 1. Overview of SPMS

II. Background and Motivation

A. Persistent memory transactions and wraps

Persistent memory transactions, which are groups of persistent memory accesses are proposed in Kiln[7], a persistent memory design characterized by a non-volatile LLC. A persistent memory transaction is committed in the non-volatile LLC first and then flushed back to persistent memory. Kiln can track each persistent memory transaction and guarantee its atomicity. While the ordering between transactions is preserved by clean-on-commit, out-of-order writebacks of stores within a transaction from volatile caches to non-volatile LLC are allowed. Similar to persistent memory transactions, Doshi et al.[10] propose *wraps* which are failure atomic code regions with all-or-nothing semantics. Updates in a wrap are stored in persistent log first and then written back to home locations in persistent memory atomically. Since this design is similar to software-based write-ahead logging except that it is implemented in hardware, we refer to it as *hardware log*.

Both persistent memory transactions and wraps are atomic with respect of system failures without persist ordering constraints within transactions/wraps. Inspired by this, our design ensures the atomicity of strands to eliminate the persistent barriers within strands and corresponding cache flush and memory fence instructions.

B. Device Support for Crash Consistency

Enforcing atomicity needs the support of persistent memory devices. BPFS [5] proposes to extend DIMMs with extra capacitors to provide enough energy to finish in-flight writes in the event of power losses. Prior researches almost all assume persists are performed atomically at 8-byte granularity. Actually, if needed, the persistent memory can easily provide larger granularity of atomicity with bigger capacitors [5]. In this work, we propose to incorporate persistent memory modules with ultra-capacitors that can provide energy for flushing tens of megabytes data back to persistent memory, like the commodity NVDIMMs[11], which have dedicated power sources to allow DIMMs to dump volatile memory data into non-volatile flash in case of power failures. NVDIMMs have evolved from using backup battery to using ultra-capacitors as backup power sources.

III. Strand Persistency based Persistent Memory

SPMS consists of strand tracking components, a volatile strand buffer and ultra-capacitors equipped with persistent memory modules as shown in Figure 1. Each strand is assigned a strand ID and tracked by the strand tracking logic. Cachelines evicted from the LLC are buffered in the strand buffer. When all updates of a strand are evicted from caches and buffered in the strand buffer, the strand is committed. A committed strand can be flushed out of the strand buffer and update persistent memory in-place. In case of system failures the ultra-capacitors can provide sufficient energy to flush all committed strands to persistent memory. SPMS simplifies persistent memory writes by performing in-place updates and eliminates cache flushes and memory fences.

A. Cacheline-based Strand Group Tracking

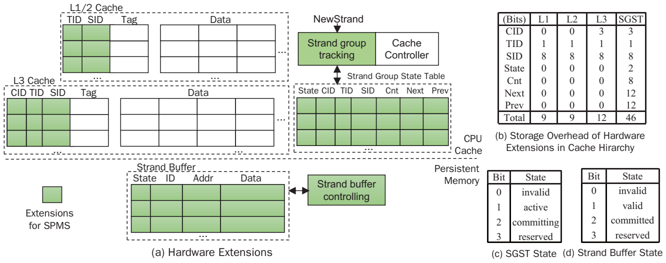
Prior work [7], [12], [10] has proposed various updates tracking schemes. Kiln [7] proposes to track persistent memory transaction writes with a FIFO. LOC [12] extends tags of cachelines to track transactions. Reference [10] implements hardware redo logs to record wrap updates. Similar to Kiln and LOC, we adopt a cacheline-based tracking mechanism to track updates in strands. Cacheline-based tracking is compatible with the native cache organization and is easily implemented. However, one challenge is strand conflicts, where one cacheline is updated by two or more strands. LOC proposes to maintain multiple versions of the cacheline when a conflict occurs, which decreases the effective capacity of the valuable cache. We propose strand group tracking scheme whose detailed implementation is introduced in IV-B. When two strands conflict, we combine them into a *strand group* treated as a strand. The cacheline-based strand tracking scheme is easily implemented and can preserve the efficiency of caches.

B. Strand Buffer

The strand buffer stores temporally cachelines evicted from the LLC. After a strand commits, the buffered cachelines belonging to this strand will be flushed back to persistent memory. For read requests to persistent memory, the strand buffer acts as a victim cache. Read requests hitting in the strand buffer are responded by the strand buffer directly without accessing persistent memory, because the new data located in the strand buffer. What's more, access latency of the strand buffer is much lower than the backend persistent memory.

C. In-place Updates

Different from SRAM and DRAM, most non-volatile memories have limited endurance, including PCM, STT-MRAM and ReRAM. In persistent memory designs, however, many consistency mechanisms induce a large amount of extra write traffic, which in turn aggravates the limited endurance problem, such as WAL and CoW. Kiln adopts STT-MRAM based LLC, which suffers from much more write traffic than main memory. What's more, cache flushes used to control persist ordering will increase persistent memory write traffic. However, SPMS can reduce persistent memory write traffic and extend persistent memory lifetime in two aspects: (1) different from WAL and CoW, updates are written back to their home locations in persistent memory directly, which reduces write traffic significantly; (2) the strand buffer can coalesce writebacks and then reduce write traffic further. What more, different from Kiln's STT-MRAM based LLC, the strand buffer is implemented by high-endurance and low-latency SRAM.



(a) Hardware Extensions

(Bits)	L1	L2	L3	SGST
CID	0	0	3	3
TID	1	1	1	1
SID	8	8	8	8
State	0	0	0	2
Cnt	0	0	0	8
Next	0	0	0	12
Prev	0	0	0	12
Total	9	9	12	46

(b) Storage Overhead of Hardware Extensions in Cache Hirarchy

Bit	State
0	invalid
1	active
2	committing
3	reserved

(c) SGST State

Bit	State
0	invalid
1	valid
2	committed
3	reserved

(d) Strand Buffer State

Fig. 2. SPMS Hardware Extensions and Storage Overhead

SPMS can keep the persistent memory in consistent states all the time even in case of system failures. Therefore, the traditional time-consuming recovery process can be skipped when the system reboots after system failures.

IV. Implementation and Overhead

A. Software Interface and ISA Extension

We assume programs are annotated by proper strand barriers. With a simple modification to the complier, a strand barrier can be translated to a *NewStrand*, a new instruction of the ISA. A NewStrand instruction indicates both the end of the former strand and the start of the later. The stores targeting the persistent memory address space within strands are recognized as strand updates. The study of how to separate a program into independent strands is beyond of the scope of this paper.

B. Strand group tracking components

In order to record each cacheline's strand information, we add extra fields to the tag of each cacheline. As shown in Figure 2(a), each tag in all cache levels is extended with the thread ID (TID) and the strand ID (SID). What's more, each tag in the LLC is extended with the CPU ID (CID) filed additionally. The storage overheads are illustrated in Figure 2(b). Only 9 bits (or 12 bits) are needed for each 64 B block in the L1/L2 cache (or the LLC).

Strand Group State Table (SGST) tracks the status of individual strands. Each SGST entry consists of seven fields: State, CID, TID, SID, Cnt, Next and Prev. State fields record the strand state shown in Figure 2(c). A NewStrand instruction changes the current strand from active state to committing state and starts a new strand with the transition of a SGST entry from invalid state to active state. CID, TID and SID record the core ID, thread ID and strand ID of the corresponding strand respectively. Cnt field records the number of cacheline updated by the strand and residing in caches. When a new cacheline is updated by a strand, the Cnt of the strand increases by 1. When one cacheline is evicted from the LLC, the Cnt of the corresponding SGST entry decreases by 1. The strand information including CID, TID and SID will be transferred to persistent memory together with the cacheline. When the Cnt field of a committing strand becomes 0, the strand is *committed*. The IDs of committed strands are transferred to persistent memory with a memory request and then the corresponding SGST entries are recycled and marked as invalid.

As we introduced, related strands that update the same cacheline are treated as a strand group. For instance, when strand B updates one cache block which is already updated by strand A, we store the index of strand B entry in the Next field of strand A. In this way all related strands belonging to a strand group are organized into a list. In order to improve operation efficiency, we add a Prev field and then one strand group can be represented and manipulated as a doubly linked list. A strand group is committed only when all strands belonging to this group are committed.

The storage overhead of SGST is shown in Figure 2(b). For a system supporting 4096 strands, the total size of SGST is only 184 KB, smaller than a typical L2 cache, 256 KB. This cacheline-based strand group tracking mechanism can be integrated with most classic cache replacement policies, such as LRU and pseudo-LRU.

C. Strand Buffer

Cachelines evicted from the LLC are buffered in the strand buffer, a volatile buffer located on the persistent memory module. Each entry in the strand buffer has four fields: State, ID, Addr and Data. Data fields store evicted cachelines, while Addr fileds store physical addresses. The ID field is the combination of CID, TID and SID fields in SGST. State fileds record state transitions of the corresponding cacheline. Initially, the state of each entry is invalid. When an evicted cacheline occupies one strand buffer entry, its state changes from invalid to valid. After the strand buffer receives the committed signal of a strand, all entries belonging to this strand are changed from valid to committed. Those committed entries can be written back to their home locations in persistent memory when there is no outstanding writeback or read request to serve. In case of system failures, the strand buffer will flush committed entries back to persistent memory within the energy window provided by the ultra-capacitors. Uncommitted entries and all data in caches will be discarded. In this way, the atomicity of strands can be guaranteed.

The strand buffer can be implemented as a full-associative cache based on SRAM. To avoid performance degradation, the operations to the strand buffer are performed off the critical path of memory accesses. The strand buffer flushes committed entries to persistent memory when there is no LLC miss and writeback to serve. Since there is no chip area limitation, the strand buffer can be significantly larger than the LLC. However, larger strand buffer needs larger capacitor to provide sufficient energy to finish outstanding committed strands in case of power failures. In the evaluation, we set the strand buffer capacity 8 times the capacity of the LLC.

When there's no free SGST entry, we stall memory requests and evict cachelines belonging to a committing strand to spare free entries for new strands. As for strand buffer overflow, we fall back on logging mechanism by writing the newly evicted cachelines into a log area in persistent memory.

V. Evaluation

A. Experimental Methodology

Our experiments are conducted using the full-system simulator GEM5 [13]. We adopt the timing simple CPU model and the classic memory system. We extend the cache hierarchy

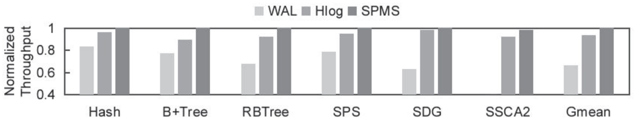


Fig. 3. Performance comparison of different persistent memory systems

and simple memory model to implement the strand group tracking components and the strand buffer described in IV. The parameters of the evaluated system are listed in Table I.

TABLE I
PARAMETERS OF THE EVALUATED SYSTEM

Processor	2.5 GHz, Timing Simple CPU
L1 I/D	Private 32KB, 8-way, 64B block , 2 cycles
L2 Cache	Private 256KB, 8-way, 64B block, 10 cycles
L3 Cache	2MB, 16-way, 64B block, 25 cycles
Strand buffer	16MB, 64B block, 30 ns access latency
PM	4GB, 120 ns access latency

Table II lists the benchmarks to evaluate our design. Most benchmarks perform insert/delete or swap operations to common data structures, including a hash table, search tree, graph, and array. A strand is a insert/delete of the corresponding data elements (e.g., key-value pairs, tree nodes, etc).

TABLE II
SIX DIFFERENT BENCHMARKS

Hash [7]	Insert/delete entries in a hash table
B+Tree [12]	Insert/delete nodes in a B+ tree
RBTree [7]	Insert/delete nodes in a red-back tree
SPS [7]	Random swap entries in an array
SDG [14]	Insert/delete edges in a graph
SSCA2 [15]	A directed graph analysis benchmark

B. SPMS performance

We compare SPMS with two other persistent memory systems with crash consistency guarantees: write-ahead logging (WAL) and hardware log (Hlog) [10]. WAL can be implemented as redo or undo log. Since redo log is usually more efficient than undo log, we choose to implement redo log. Different from software-based logging, through non-temporal instructions (e.g., MOVNT in x86 architecture), Hlog transfers log records of updates to a specific backend SCM controller which stores log records into persistent memory before performing in-place updates.

Figure 3 shows the throughputs of the three designs. The throughputs are normalized to the baseline system, which does not ensure crash consistency. Compared to WAL, both SPMS and Hlog have much higher throughputs. The main reason is that different from WAL, SPMS and Hlog avoid a large amount of extra persistent memory write traffic. We also find that the throughputs of SPMS are higher than Hlog's in all six benchmarks and SPMS outperforms Hlog by 6.6% on average. What's more, in most benchmarks, SPMS's throughput is higher than the baseline slightly. This is contributed to the very little overhead of SPMS. What'more, read hits in the strand buffer can improve the throughput of SPMS.

C. PM Write Traffic

Figure 4 shows the SPMS write traffic normalized to the baseline. SPMS reduces persistent memory write traffic by 30% on average. SPMS dose not incur any cache flushes for crash consistency guarantees. What's more, the strand buffer provides more opportunity to coalesce updates.

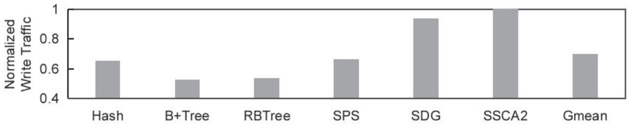


Fig. 4. Write traffic of SPMS (normalized to the baseline)

VI. SUMMARY

In this paper, we propose a novel persistent memory system based on strand persistency which minimizes persist ordering constraints with crash consistency guarantees. SPMS tracks each strands and guarantees its atomicity, eliminating persist ordering constraints within strands, allowing persistent memory to perform in-place updates and eliminating the time-consuming recover phase when systems restart after failures. SPMS outperforms the baseline system slightly and reduces the persistent memory write traffic by 30%. Our proposed persistent memory system introduces some minor modifications to the memory hierarchy. Although hardware modifications are discouraging propositions, we believe that these modifications are essential for extensive uses of persistent memory.

VII. ACKNOWLEDGMENTS

This work is supported by National High-tech R&D Program of China (No.2015AA015305), National Natural Science Foundation of China (No.61433019, U1435217, 61572045, 61332003, 61402503).

REFERENCES

[1] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau *et al.*, "Optimistic crash consistency," in *SOSP*, 2013, pp. 228–243.

[2] J. Ren, J. Zhao, S. Khan *et al.*, "ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems," in *MICRO*, 2015, pp. 672–685.

[3] J. Coburn, A. M. Caulfield, A. Akel *et al.*, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," in *ASPLOS*, 2011.

[4] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *ASPLOS*, 2011.

[5] J. Condit, E. B. Nightingale, C. Frost *et al.*, "Better I/O through byte-addressable persistent memory," in *SOSP*, 2009.

[6] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*, 2014, pp. 265–276.

[7] J. Zhao, S. Li, D. H. Yoon *et al.*, "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support Jishen," in *MICRO*, 2013, pp. 421–432.

[8] A. Kolli, J. Rosen, S. Diestelhorst *et al.*, "Delegated Persist Ordering," in *MICRO*, 2016.

[9] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," February 2016.

[10] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *HPCA*, 2016, pp. 77–89.

[11] "Micron Technology, Inc. NVDIMM DRAM Modules." [Online]. Available: https://www.micron.com/products/dram-modules/nvdimm#/

[12] Y. Lu, J. Shu, L. Sun *et al.*, "Loose-Ordering Consistency for persistent memory," in *ICCD*, 2014, pp. 216–223.

[13] N. Binkert, S. Sardashti, R. Sen *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, aug 2011.

[14] J. Siek, L.-Q. Lee, and A. Lumsdaine, "Boost Graph Library: Adjacency List - 1.59.0." [Online]. Available: http://www.boost.org/doc/libs/

[15] D. a. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *HiPC*, 2005, pp. 465–476.