2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071088

Reconciling Selective Logging and Hardware Persistent Memory Transaction

Chencheng Ye†, Yuanchao Xu‡, Xipeng Shen‡, Yan Sha†, Xiaofei Liao†, Hai Jin†, Yan Solihin§
† National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
‡ North Carolina State University, Raleigh, North Carolina, USA
§ University of Central Florida, Florida, USA
{yecc,xfliao,hjin}@hust.edu.cn, {yxu47,xshen5}@ncsu.edu, soyan0408@gmail.com, Yan.Solihin@ucf.edu

***Abstract*—Log creation, maintenance, and its persist ordering are known to be performance bottlenecks for durable transactions on persistent memory. Existing hardware persistent memory transactions overlook an important opportunity for improving performance: some persistent data is algorithmically redundant such that it can be recovered from other data, removing the need for logging such data. The paper presents an ISA extension that enables selective logging for hardware persistent memory transactions for the first time. The ISA extension features two novel components: fine-grain logging and lazy persistency. Fine-grain logging allows hardware to log updates on data in the granularity of words without lengthening the critical path of data accesses. Lazy persistency allows updated data to remain in the cache after the transaction commits. Together, the new hardware persistent memory transaction outperforms the state-of-the-art hardware counterpart by 1.8× on average.**

I. Introduction

Durable transaction is fundamental for computations involving persistent data objects [1]–[5], such as objects on CXL-based byte-addressable non-volatile memory expansion [6], NVMDIMM-C [7], and byte-addressable SSD [8], [9]. Log creation, maintenance, and its persist ordering are known to be performance bottlenecks for durable transactions. The large time overhead has remained a critical issue for the use of persistent data objects in computations.

Figure 1 illustrates the use of durable transactions. To insert node B into a double-linked list on persistent memory, four writes are needed to update the "next" and "prev" fields of the three involved nodes. If the first store takes effect immediately on the persistent memory and then the system crashes after that, the linked list would be in an inconsistent state. *Durable transaction* is a construct to address the problem, which guarantees that upon a crash or power loss during a durable transaction, all the modifications to the persistent data objects in the transaction can be completely canceled, that is, those data objects can restore to the states they had at the start of the transaction. So for the double-linked list example, including the writes into a durable transaction can keep the linked list in a consistent state upon unexpected crash or power loss.

Supporting a durable transaction requires data logging. Both undo and redo logging can be used. The former keeps a copy of the old value of a to-be-modified data item, and the latter redirects memory writes to a shadow copy of the data of interest. Maintaining logs incurs a considerable time overhead. As prior studies show [10], making a copy of every data to be updated incurs about 3.3X write traffic and 3X overhead on average (log records contain data and metadata such as addresses and end marks.) The problem has prompted some recent efforts to reduce the logging overhead, which falls into two main classes. The first class is *selective logging* [11]–[18], which tries to avoid logging some data updates if the effects of those updates can be canceled based on other information. For the linked list example in Figure 1, if we do not log the writes after the first write, the linked list can actually still be put into a consistent state after a crash. The insight is that the bi-directional linkage in the data structure provides some redundant information enough for recovery. At a crash, execution of the code in Figure 1(d) is all needed to restore the consistency of the list (assuming undo log is used). Although prior work has shown that selective logging may reduce logging overhead by 1.13-1.82X [12], [18]–[20], it has been regarded as an application-specific approach, requiring programmers to manually design it and tailor its implementation based on the deep semantics of the application.

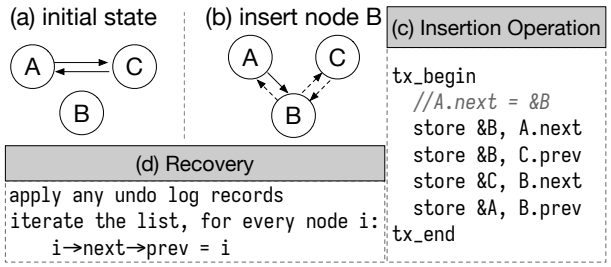

Fig. 1. Node insertion in a double-linked list

The second class is *hardware persistent memory transaction*. Unlike the scenarios assumed in the *selective logging*, this approach does not rely on software-based logging but uses hardware to more efficiently log data updates [21]–[25]. It reduces the logging time overhead through algorithm-independent hardware accelerations. Although this approach

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

is general, it loses the benefits of selective logging. All the aforementioned proposals in hardware persistent transactions automatically log every data update, causing selective logging to lose applicability, and the opportunities for avoiding unnecessary logging are wasted. The loss is substantial, up to 2.1× performance loss (as Section VI details).

In this paper, we propose the first solution that enables *selective logging in hardware persistent memory transaction*. Our approach reconciles the two approaches to get the best of both worlds.

The basic idea is to introduce a new instruction, *storeT*, which makes the hardware skip data logging on such store instructions. There are, however, a set of challenges for this idea to work: How to materialize the new store instruction? How to make the design generally applicable to transactions of various sizes? How to deal with a cache line containing both log-free and to-be-logged data while maximizing the benefits of selective logging? Should the hardware persist log-free data at the end of a transaction? As some log-free data are recoverable from other data, they are not strictly necessary to be persisted. How to take advantage of this property without harming the soundness of the execution?

This paper presents our design and how it addresses all the challenges. The proposed architecture consists of a full design of the ISA extension for two mainstream hardware logging techniques, undo and redo logging. It has several important features: (i) a log coalescing and packing mechanism to enable efficient logging of persistent memory accesses at word level; (ii) a lazy mechanism that defers the persisting of log-free data whenever possible without sacrificing the recoverability; (iii) other components, such as cache entry format and cache coherence, for incorporating some typical hardware transaction optimizations, including unbounded transactions, decoupled execution and logging, and so on. They help ensure the compatibility of the new architecture with existing hardware transaction primitives. The paper, in addition, examines the implications of the introduced selective logging mechanism to programming systems and suggests several opportunities for compilers.

Our evaluation shows that the proposed architecture is effective in enabling selective logging on hardware memory transactions. On six durable data structures benchmarks, it brings 1.8× speedups over prior hardware persistent memory transactions and reduces memory write traffic to persistent memory by 46%.

The introduced architecture support, for the first time, makes selective logging possible for hardware transactional memory. It addresses the fundamental roadblock for practical computations on persistent objects, and offers an easy way to flexibly combine selective logging and lazy persistency, opening up new opportunities for optimizing transactions for persistent memory (Section V).

Overall this work makes four main contributions:

- We propose the first architecture design to enable selective logging in hardware persistent memory transactions.
- We introduce a log coalescing and packing mechanism to enable efficient logging of persistent memory accesses at the word level.
- We propose a lazy mechanism that defers the persisting of log-free data whenever possible without sacrificing recoverability.
- We empirically evaluate our design, confirming the significant benefits of the selective logging capable hardware persistent memory transaction.

II. Overall Design

This section presents the overall design of selective logging capable hardware persistent memory transaction. Although the key design is not bound to a particular kind of hardware transaction, this section assumes undo logging transactions to simplify the discussion. Further, the following discussion assumes that a persistent/durable transaction is a transaction only for atomic durability, despite the fact that it can be easily extended for concurrency. Section V discusses the relationship with atomic durable/concurrency transactions.

A key component of the design is a new instruction `storeT`. Unlike normal `store` instruction, the hardware persistent memory transaction creates no log record for the data updated by `storeT`. It is intended to be used by programmers or compilers in scenarios where in the recovery of a crash-interrupted transaction, the program tolerates or has the built-in code to fix inconsistencies caused by the updates on the log-free data. Log-free data can be persisted at the transaction commit just as normal data do, but can also enjoy deferred persisting as detailed later in this paper.

Figure 2 shows the semantics of `storeT`. The 1-bit flag *lazy* determines whether the hardware shall persist the updated data at the transaction commit or defer its persistence. Section III-C explains the use of the field. The 1-bit *log-free* flag determines whether or not the hardware disables the semantic of `storeT`, treating `storeT` as a `store`.

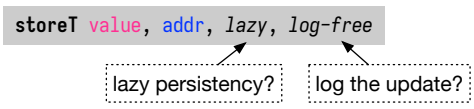

Fig. 2. Syntax of `storeT` instruction

Our proposed architecture to materialize `storeT` has two important features, fine-grained logging, and lazy persistency. We name the architecture SLPMT (*Selective-Logging Persistent Memory Transaction*). SLPMT adds minor modifications to a few components of existing CPU architecture, including adding additional fields for L1 and L2 cache lines, and extending the cache coherence protocol and the data path of memory instructions. The new fields record which part of a cache line is already logged and needs to be persisted at a transaction commit. The cache coherence protocol notifies the cache subsystem to persist the data, detect conflicts, or invalidate cache lines. We also add new on-core components, including a buffer for coalescing logs, a set of signatures for recording the

read- and write-set of committed transactions with some data persistency deferred, and a register for recording the essential information of transactions, as illustrated in Figure 3.

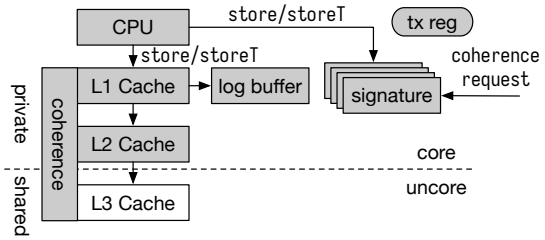

Fig. 3. Overview of the SLPMT architecture. Solid gray components are new or modified.

When the program updates a persistent datum with a `store` instruction, the hardware sets the *persist* bit of the L1 cache line to one to indicate that the cache line must be persisted at the transaction commit. It then checks the *log* bit of the L1 cache line to determine whether to create a log record for the cache line. If the log bit is unset, indicating the cache line is not yet logged, the hardware creates a log record for the datum and then puts the record into the log buffer. The use of the buffer is to harness the opportunities for coalescing the records to reduce the write traffic in persisting the log records. It flushes all log records when the transaction commits; it flushes a single log record when the associated cache line is evicted from the private cache into the shared cache.

When an updated cache line is evicted from L1 into L2, the hardware propagates the log bits and the persist bit to L2 cache, such that when the subsequent `store` to the same datum fetches the cache line into L1 cache, it can see that the log record associated with the datum has been created and hence avoids redundant logging.

The hardware executes a `storeT` instruction as follows. It sets the persist bit to the value of the `lazy` field of the instruction. Unlike `store`, it does not check or set the log bit, hence creating no log record for that cache line.

To commit a transaction, the hardware persists all the records in the log buffer and cache lines in the private cache whose persist bits are one. It defers the persisting of other cache lines till any of the data needed for recovering the cache lines are modified (detailed in Section III-C). To detect such data modification, the hardware maintains a per-transaction signature to track the modifications on the read- and write-set of a transaction, namely the working set. Before updating any data in the working set, the hardware must ensure that all the lazy persistency cache lines created by the associated transaction have reached persistent memory. Then the hardware reclaims the signature and proceeds with the data update.

III. Detailed Design

This section gives the details of the design. It starts with a basic design and then explains the introduced optimizations.

A. Baseline Design

We first present the baseline design without fine-grained logging or lazy persistency.

The baseline assumes undo logging. It allows unbounded transactions. It detects and manages conflicts. As the management of volatile and persistent data is decoupled, we focus on the design for persistent data only.

SLPMT augments each L1 and L2 cache line with a persist bit and a log bit. The `storeT` instruction sets the bits according to the operands. SLPMT performs operations on the cache line according to the bits. The persist bit indicates whether SLPMT persists the cache line on transaction commit, namely eager persistency. The log bit indicates whether SLPMT has created a log record for the cache line; thus, further updates on the cache line will not trigger log creation. For example, a `storeT` instruction with unset lazy and log-free operands sets the persist bit of the target cache line but not the log bit, while `store` sets both bits. Table I shows the full semantics of all the forms of `store` and `storeT` instructions.

TABLE I
SEMANTICS OF `store` AND `storeT`. SLPMT SETS THE PERSIST AND LOG BITS ACCORDING TO THE COMBINATION OF THE LAZY AND LOG-FREE OPERANDS.

Instruction			Effects	
	lazy	log-free	persist bit	log bit
store	-	-	1	1
storeT	0	0	1	1
storeT	0	1	1	0
storeT	1	1	0	0
storeT	1	0	0	1

When SLPMT sets the log bit, it creates an undo log record. The effect of `store` instruction becomes visible only after SLPMT inserts the record into the buffer. The log record consists of the data and the address of the cache line. SLPMT stores the record with the log buffer next to the L1 cache. The buffer propagates a record to persistent memory in two cases: (1) when CPU must evict the associated cache line from L2 to L3 cache, SLPMT flushes the log record; (2) when the buffer is full, SLPMT flushes all the eight records. In the latter case, the log buffer essentially coalesces the records, a feature inspired by a prior study [24].

An interesting combination is to set a lazy operand but keep the store logged, indicating that `storeT` must create a log record for the cache line, but it is unnecessary to persist it on transaction commit. It results in the opportunity for hardware to avoid persisting the log record of the lazy persistent cache line if the cache line remains in the cache when the transaction commits. The hardware still must persist the log record if the cache line is evicted from the cache during the transaction. On commit, the hardware simply discards associated log buffer records for lazily persistent cache lines.

Before an L2 cache line overflows to the L3 cache, if the persist bit is set, SLPMT notifies the log buffer to persist the log record. SLPMT clears the persist bit and proceeds with the

eviction if the log record is not in the buffer or has already reached the persistent memory, .

When a transaction commits, SLPMT drains the log buffer and then issues cache coherence requests to persist all cache lines in the private cache whose persist bits are set. It must maintain the correct order of persisting three kinds of data: log records, cache lines updated by `store`, and cache lines updated by `storeT` only. We call them *logged cache lines* and *log-free cache lines*. Figure 4 shows the order in which different kinds of cache lines must reach persistent memory for undo and redo logging.

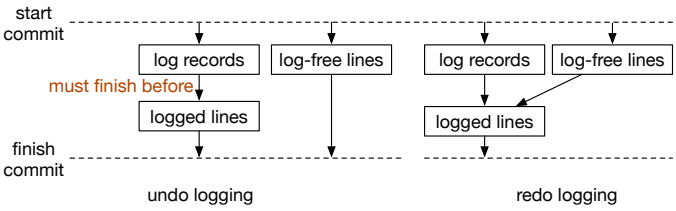

Fig. 4. The order in which a transaction persists the data and logs

For undo logging, the transaction must persist log records before persisting logged cache lines. It may persist log-free cache lines at any time.

For redo logging, to ensure the correct recovery, all log-free cache lines must reach persistent memory before logged cache lines. Consider the opposite. The log records and a logged cache line become persistent, while some log-free cache lines remain volatile. A system crash corrupts the log-free lines, jeopardizing the recovery of the transaction. It can neither revoke the updates on the logged cache lines with redo logs nor recover the log-free cache lines as the data it depends on, such as the logged cache lines, have already been updated. The remedy is to ensure that all log-free cache lines reach persistent memory before logged lines do. For explanation purposes, we focus the following discussion on undo logging transactions but note that the principle of selective logging applies to redo logging as well.

B. Fine-Grained Logging

To manage log in the granularity of words, SLPMT expands the log bit to a bitmap and associates each word with a log bit. Suppose each word is eight-byte long and each cache line is 64-byte long. A naive design would require eight log bits per cache line, incurring 36.5KB space overhead per core on representative CPU architectures[1]. We next explain how the design in SLPMT reduces the space cost substantially.

1) Cache Entry: To avoid such significant space overhead, SLPMT adopts different granularities for the log bits of different levels of cache. It augments every L1 cache line with eight log bits. Each log bit is associated with one eight-byte word. For L2 cache lines, SLPMT introduces a log bit for every 32-byte word, as shown in Figure 5. It maintains no log bits for L3 cache lines. This optimized design incurs only 1.5KB space overhead per core. SLPMT introduces transaction ID to implement lazy persistency, which will be presented later in Section III-C.

[1]Assuming each core provides 2.25MB L1, L2, and sliced L3 cache as in mainstream Intel and AMD processors.

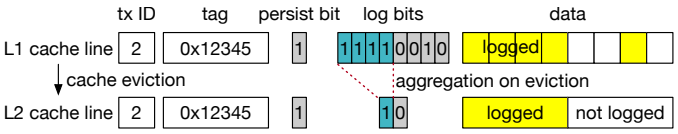

Fig. 5. Formats of L1 and L2 cache entries

Because the log bit granularity in L2 is larger than in L1, when an L1 cache line is evicted into the L2 cache, SLPMT sets one L2 log bit as the logical conjunction of the corresponding four log bits of the L1 cache line, as shown in Figure 5. When an L2 cache line is fetched into the L1 cache, SLPMT reverses the operation, replicating the log bit of the L2 cache line into the corresponding four log bits of the L1 cache line.

In some cases, the solution could cause duplicated logging for words and incur some overhead. Consider a program updating only one word in an L1 cache line. SLPMT creates a log and sets the log bit accordingly. Later, the cache line is evicted into the L2 cache; the conjunction of the log bits is zero; hence the log bit in L2 is unset. When the program updates the word again, SLPMT fetches the cache line into the L1 cache. It finds that the log bit is unset, and would create a log record for the word again (without overwriting prior logs). Fortunately, such cases are uncommon as programs often exhibit temporal or spatial locality, making the reuse of an evicted cache line uncommon.

An optimization to deduplicate logs is to speculatively create log records to encourage log bits aggregation. Consider a program that updates only the first three words of an L1 cache line. When it is evicted, SLPMT aggregates the first four log bits, but the conjunction is zero as the fourth word is not logged yet. Instead, SLPMT can create a log record for the fourth word despite the fact that it is clean. It would make the conjunction one. The optimization does not necessarily incur extra write traffic as SLPMT coalesces the log records of the four words through the adaptive log buffer. We will come back to this point in Section III-B2.

When an L2 cache line with a set persist bit is evicted into the L3 cache, SLPMT persists the associated log records and the cache line before the eviction. For a cache line fetched from the L3 cache into L2, the persist and log bits of the L2 cache line are all initialized to zero.

It is possible to use the same logging granularity for the L1 and L2 cache to simplify the design. But it would incur 3× extra on-chip space overhead. Given that the L2 cache size is growing fast (i.e., 1MB to 4MB for modern processors), maintaining the same logging granularities in L2 may cause up to 64KB of space overhead. The proposed mixed granularities reduce 75% of the space overhead.

2) Log Buffer: SLPMT adopts a four-tier log buffer to facilitate log coalescing. The idea is inspired by the design

of buddy memory allocation in Linux. The tiers are for the log records of a word, double words, quadruple words, and a cache line, respectively. Each record consists of an address and the logged data. Therefore, the log record for a single word, double words, and quadruple words are 16, 24, and 40 bytes, respectively, as shown in Figure 6.

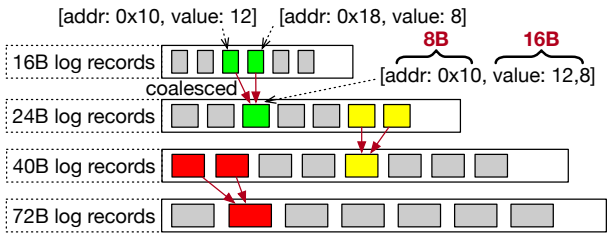

Fig. 6. Tiered log buffer

As SLPMT creates a log record for a word, it starts by inserting the record into the first tier. On insertion, the log buffer searches in the tier for a record of the data item next to the inserted one. If such a record exists, the log buffer coalesces the two records and inserts the coalesced record into the next tier. The log buffer repeats coalescing records on every insertion on every tier except for the tier of the full cache line.

The log buffer finds available slots in a tier by checking a register that records the allocation of each slot with a bit. When the tier is full, without any opportunity of coalescing records, the log buffer drains the tier by persisting all the records belonging to that tier.

The buffer performs record coalescing and persisting asynchronously with the execution of the program. Although it locks the tier during coalescing and persisting, locking the tiers other than the first tier does not block the execution of the program, as the hardware can still insert records into the buffer, even though the coalescing of the records may be delayed. When the log buffer locks the first tier, instructions other than `store` can still execute. When the first tier is locked, as the effect of `store` instruction must be visible only after the insertion operation finishes, the program stalls on accessing the updated data.

To align with the granularity in which the processor writes data on persistent memory, SLPMT sets the size of each tier according to the least common multiple of the record size and the cache line size. For example, the size of the tier of double words is $lcm(24, 64) = 192$ bytes, which is three cache lines long. Overall, the sizes of the tiers are two, three, five, and nine cache lines, respectively, such that each tier retains up to eight records. Prior work adopts a similar principle of setting the log buffer size [24].

As the commonest kind of operation on the log buffer is searching for a record with a given specific address on cache line eviction or record coalescing, SLPMT enables concurrent search by storing the addresses in the records with ternary content addressable memory [26], similar to the design of cache line tags.

When SLPMT drains the buffer, a problem is how to compact the records from different tiers into cache lines. The records have different sizes. Therefore, it is challenging to align them with cache lines. SLPMT organizes the log buffer as a pad [27], which is a cache operating in variable-sized objects and interfacing main memory in cache lines. In the case of a log buffer, the objects are the records.

Before draining the log buffer on the transaction commit, SLPMT scans all the records in the buffer to find out all records associated with lazy persistent cache lines (with persist bit unset). It simply discards those records.

C. Lazy Persistency

A fundamental requirement for lazy persistency is that SLPMT must persist the data at a correct point of time, such that they are guaranteed to be recoverable with other related data from a sudden crash. To that end, the timing of persistence must meet the following condition:

> Suppose at a time point P, the value of a lazy persistent datum A depends on a set of data S. The datum A must reach persistent memory before any update to S after P reaches persistent memory.

Consider the opposite. The system crashes when updates to S have reached the persistent memory while A remains volatile and gets corrupted. The post-crash recovery must rebuild A from the updated S, which is different from its version when A is created. Such a recovery can be hard or even impossible.

Tracking data dependency precisely, however, is costly. We hence propose an alternative to enable fast dependency checking. It is based on an important observation: As a lazy persistent datum is created within a transaction, all data it depends on must belong to the working set of the transaction, including the read- and write-set. Our proposed solution works as follows. Once a transaction with lazy persistent data commits, the solution records the working set of the transaction. When a datum in the working set is to be updated after the transaction, the solution blocks the update until it persists all the associated lazy persistent data. It is worth mentioning that as the solution detects the conflicts between the working set and both data updates in or out of transactions, it, unlike prior asynchronous persistency solution [28], does not rely on any assumptions about the semantics of the specific program, making it general. We next elaborate on the specifics of the solution.

1) ISA Modifications: Programmers enable lazy persistency for log-free data by setting the *lazy* operand of `storeT`, notifying SLPMT to set the persist and log bits of the updated cache line accordingly as shown by Table I. SLPMT detects lazy persistent cache lines by checking the associated persist and log bits.

Store operations on a data item within the lazy persistent cache line effectively cancel the lazy persistency. As the cache subsystem sends data to persistent memory in the granularity of the cache line, subsequent `store` or `storeT` (with unset

lazy operand) instruction sets the persist bit, causing SLPMT to persist the cache line at transaction commits.

2) Cache Entry: SLPMT associates each L1 and L2 cache line with a two-bit transaction ID, indicating which transaction updates the cache line. The transaction ID is local to a core. When a transaction begins, the hardware allocates an ID to the transaction. The *transaction register* maintains two values to record the first and the last free IDs. The two values act as the pointers to a circular buffer. All IDs between the first and the last free IDs are free.

When SLPMT must persist the lazy persistent cache lines updated by a transaction, it persists all data owned by those transactions prior to the must-persist one. When SLPMT runs out of free transaction ID, it reclaims the ID of the earliest transaction and then persists all lazy persistent data. The ID is the one next to the last free ID. Organizing the free transaction IDs as a circle avoids the case that the data updated by a transaction at the beginning of the program remains volatile after too many transactions.

Similar to committing a transaction, SLPMT persists the lazy persistent data with a cache coherence request. The request scans all the private cache to find out and persist all lazy persistent cache lines. The number of the coherence request issued is almost the same as other persistent memory transactions [21], [23], [24]. The reason is that, similar to other systems, the core issues a coherent event on each data or log persist operation, and the memory controller returns a message when the data or log record reaches the persistent domain.

3) Tracking Conflicts: To track the working set of a transaction, SLPMT introduces a signature for every transaction with an assigned ID. The signature records the addresses of data in the working set of the associated transaction. It detects conflicts on coherence requests as the design of unbounded hardware transactions in prior work [21], [23], [29], [30]. All the signatures share the same hash functions to save area and energy. On every coherence request triggered by `store` and `storeT`, the hardware checks the signatures for the address to write on. If there is a match, SLPMT persists in all lazy persistent cache lines updated by the transaction associated with the found signature. On both `store` and `load` instructions, SLPMT checks the transaction ID of the cache line to access. If the ID is different from the current transaction ID, indicating that the cache line is lazy persistent data created by an earlier transaction, it triggers the same data persistence as mentioned above. Both checks are cheap. Checks of the signature and of cache line persisting happen on store operations; they are off the critical path of program execution. Checks of the transaction ID happen just on cache line access.

4) Consideration for Persistency: Note that there is no need to maintain a particular order in which the lazy persistent data reaches persistent memory as they can always be recovered from persistent data. Therefore, complex mechanisms [28] for ordering the data persistency among threads is not necessary for SLPMT.

Our solution allows a program to enforce the persistency of the lazily persistent data of a specific transaction. The program just needs to run a few empty transactions as the hardware enforces persistence when it reuses a transaction ID. For example, if a program runs four empty transactions, all lazily persistent data are made durable.

D. Hardware Overhead

SLPMT incurs 6.1KB space overhead: 3.9KB on the new fields of L1 and L2 caches, 1.2KB on the log buffer, and 1KB on the signature. The implementation adopts four 2048-bit signatures. All the new storage spaces are volatile. Overall, the design incurs 0.2% die size of a Westmere core [31] as reported by CACTI [32]. The design extends the cache coherence protocol to realize lazy persistency without changing coherence states.

IV. Implications to Programming Systems

The focus of this work is to create a mechanism that can enable selective logging on hardware memory transactions. How to make the best use of this mechanism is up to programming systems, a systematic exploration of which is orthogonal to this study. Nonetheless, it is worth examining the implications of the new mechanism to programming systems support, which may provide some insights into adding support of SLPMT into compilers and programming modules.

A. Use in Customized Algorithms

There are several ways in which the SLPMT mechanism can be used. The first is to be directly used by programmers. Some intrinsics, APIs, language constructs, or annotations can be created, with which the programmer can indicate in a persistent transaction the assignments where `storeT` should be used. An alternative way is to make compilers automatically identify such assignments and generate `storeT` instructions for them. The two approaches can be used together, with the compilers serving as an optimizer.

Like other software transactions, incorrect annotations could cause the scheme to misfunction or suffer performance loss. If the programmer incorrectly marks a store as log-free, a crash in the middle of the transaction undermines the recoverability if the updated data reaches persistent memory. But such threats do not span across transaction commits. If the programmer incorrectly marks a store as lazy persistence, a crash in the transaction does not hurt recoverability. Instead, a crash after the transaction commits and before the data reach the persistent domain may cause the loss of the up-to-date value of the data.

B. Opportunities for Compilers

There are lots of opportunities for compilers to help with the use of `storeT`. This section presents two common patterns of persistent memory programming to illustrate some of the opportunities. In our discussion, we will assume undo logging transactions. We assume the program is represented in the *static single-assignment* (SSA) form, such that each variable

can be assigned only once. The discussion assumes that the instruction that creates a variable is also the first instruction that updates the data at the memory location of the variable. The discussion is based on the results of a compiler memory dependency analysis [33] that identifies clobbered load and store instructions.

a) Pattern 1, for log-free store: A persistent memory variable can be log-free if it is created by a function before or within a transaction, such that on post-crash recovery, re-executing that function can reproduce the variable. The side-effect of the function must be ignorable or can be canceled without the knowledge of the variable. Figure 7 shows a common example where a program allocates a new node when it inserts a record into a list or a tree. It assumes the recovery uses a garbage collector or a persistent inspector from PMDK [34], [35] to reclaim the leaked variable `x`.

```
//insert(node* pos, value_type &v)
tx_begin()
  ...
  node* x = malloc(sizeof(node))
  ...
  x→prev = pos //x is log-free
  x→value = v //log-free
tx_end()
```

Fig. 7. Pattern 1 for compiler-assisted selective logging; simplified from GCC STL's list implementation [36]

Similarly, if a variable will not be used further, there is no need to persist it at the transaction commit. For example, if a transaction intends to free a memory region, such as removing a node from a tree and deallocating its memory space, any update in that transaction on the memory region needs no persistence.

The compiler identifies the pattern based on the use of certain functions, such as *malloc* and *free*, and finds all the store instructions corresponding to the associated memory regions. It then replaces the `store` instructions with `storeT` for the log-free purpose.

The recovery is independent of the crashed transaction. For log-free data, the recovery runs garbage collection to reclaim memory regions allocated in the interrupted transactions. The lazily persistent data needs no recovery as they are meant to discard when the transaction aborts.

b) Pattern 2, for lazy persistence: Besides the unneeded persistent variables, a persistent variable can be lazily persistent if the recovery can rebuild both its address and its value from either other persistent data or log records.

The compiler identifies candidate lazily persistent variables by identifying flow-out variables of the transaction. It is a similar but reverse process of the flow-in variable analysis done in previous compilers for software transactions used in idempotent region analysis [19], [20], [37], [38]. For each of the flow-out variables, the compiler tracks along the def-use chains of the variable to determine whether the variable's value at each of the store instructions in the transaction depends on only data either recoverable or already persisted before that instruction and marks the variable as recoverable. Then, based on the analysis results, for each store instruction in the transaction, the compiler checks whether both of its operands are recoverable. If so, it replaces the `store` instruction with the lazy-persistency `storeT` (not for the log-free purpose, though).

The compiler generates a recovery for each transaction. Similar to the re-execution transaction [19] that uses compilers to find all the dependent variables in a transaction and generate the code for re-executing the transaction from those variables, our compiler generates the re-execution process from the variables that the candidate stores depend on. To find the dependent variables to initiate the lazily persistent rebuilding, the compiler may need to record the addresses of the variables or infer which log records are required for the recovery. We use a log-free store to record those addresses within a log area other than the undo log area.

It is worth noting that the lazily persistent variables can be free from logging when a future transaction updates them because the variables can be rebuilt to recover their values prior to the transaction. This optimization, however, requires the transaction to determine which variables are lazily persistent through compilers or extra hardware support, which we leave for the future to explore.

V. Discussion

A. Other New Opportunities

Besides the benefits already mentioned, the introduced hardware support, SLPMT, offers an easy way to flexibly combine selective logging and lazy persistence, offering new opportunities for optimizing persistent transactions.

An example is the optimization of in-place update transactions. Persistent memory offers fast sequential write but slow random write. Conventional in-place update such as undo logging transactions [24], [39], [40] suffers from slow data persistence on the critical path of transaction commit. SLPMT enables optimizations to eliminate the random data writes with the following strategy. For every transactional store operation, the transaction updates the data with lazily persistent but logged `storeT`. The transaction also creates a record of the new value of the data with eager but log-free `storeT` instructions. It appends the record to the end of a sequential array. At transaction commit, the hardware persists the array but leaves the updated data in the cache.

If the software crashes during transaction execution, the lazily updated data either remain in the volatile cache or have associated undo log records when they overflow. The recovery can exploit the undo log records to revoke all updates. If the software crashes after the transaction commit, the lazily persistent data that remain in the cache may lose. The recovery uses the sequential records as a redo log to reapply the updates. Unlike conventional redo logging [41], [42], this solution requires no address indirection.

B. Transaction Abort

A program may abort an SLPMT transaction in the absence of crashes for concurrency control. Similar to transactional memory [29], [30], the SLPMT transaction revokes updates on volatile data by issuing a coherence request that invalidates associated cache lines updated within the transaction.

Different from transactional memory, SLPMT revokes the updates on persistent data with the following steps: (1) clear the log buffer and the signatures; (2) trigger an interrupt to let a kernel-space system call to apply the undo log to the persistent data; (3) once the call finishes, the user-specified recovery revokes the update on log-free data. There is no need to drain the log buffer, as the cache lines associated with the log records must still reside in the private cache. When the cache lines are evicted into L3, the hardware must persist the log record before the eviction.

C. Context Switch

SLPMT allows thread context switch through a design based on prior work [21], [23], [29], [43]. It borrows those designs to detect and resolve the conflicts when a thread is switched out. Regarding crash consistency, before a context switch, the OS kernel drains the log buffer. The hardware tracks the dependencies for lazy persistent data even when the context switches. There is no operation on the signatures and the values for transaction ID allocation as they are not specific to a context. Note that, different from other designs, the signatures are not used for detecting conflicts as they record the working sets of committed transactions.

D. Relationships between SLPMT and Atomic Transactions

Our discussion has assumed that the transaction of interest is a durable transaction. There are some prior works [21], [23] that propose persistent atomic transactions where a transaction is a unit of both atomicity and persistency. The support from SLPMT on persistency is compatible with the atomicity assurance by the traditional hardware transactional memory. In other words, the selective logging-capable persistency by SLPMT still works for persistent atomic transactions with no extra changes needed. Specifically, to realize atomicity, we can equip the SLPMT with cache coherence protocol designs in transactional memory. The coherence protocol is orthogonal to the modification needed by durability or selective logging.

E. Battery-Backed Cache

For processors with battery-backed cache, the need for logging is removed if the working set of a transaction fits into the cache. However, log is still needed to ensure the atomicity if any data is evicted into memory, which is possible as a durable transaction can be large [21], [23]. In those cases, SLPMT still applies.

VI. Evaluation

We evaluate the effectiveness of the proposed techniques, with a focus on the performance benefits brought by SLPMT and each of its key techniques, and provide some sensitivity analysis.

A. Workloads

Table II lists the workloads. They consist of four transactional application kernels from STAMP benchmark [44], plus an exemplary persistent memory key-value store application from PMDK framework. Whereas specifically designed data structures can potentially benefit more from the proposed hardware primitives, we port existing code according to the rules proposed in Section IV to present the usability of the hardware primitive and the effectiveness of the compiler.

TABLE II
Workloads

Benchmark	Description
kernels of STAMP benchmark [44]	
hashtable	chained hash table that resizes when there are three records in each bucket on average
rbtree	red-black self-balancing tree; each node contains a pointer to the parent and an integer recording the color
heap	max heap using an array to store all the nodes
avl	AVL self-balancing tree; no parent point in the node; the limited
exemplary application [45] from PMDK framework	
kv	key-value store engine that can be configured with various indexing data structures; the evaluation uses btree, ctree, and rtree

Similar to a prior work [19], we evaluate each benchmark with ycsb-load [46] workload. Each benchmark involves 1,000 insertion operations. Each operation consists of a persistent memory transaction, an 8-byte key, and a 256-byte value. We study the sensitivity to the value size.

Log-free and lazy persistency annotations are added manually into the transactions or automatically with the compiler by following the principles described in the earlier sections IV. Unless noted otherwise, we evaluate the kernel benchmark with manually inserted annotations and the application benchmark with compiler-inserted annotations due to the large code size of the application. We extend clang for the compiler support and implement the compiler optimizations with LLVM framework [47], [48]. The compiler detects the data def-use chain dependency with MemorySSA [33].

B. Hardware Simulation

We implement SLPMT on an X86-like processor. The design is not bound to a specific architecture. We model performance with the Gem5-21.2.1 system simulator. Table III shows the configuration. For parameters not shown in the table, we use the default value provided by Gem5.

We evaluate the correctness and the performance of the modified MESI cache coherence protocol with Ruby, a detailed memory subsystem simulator integrated into Gem5.

The evaluation models a persistent memory backed by Intel's ADR where data become persistent when it reaches the *write pending queue* (WPQ) in the memory controller. On a failure, the hardware drains the write pending queue to persistent memory. The size of the write pending queue is

TABLE III
SYSTEM CONFIGURATION

Component	Parameter
CPU	64-bit X86 out-of-order processor, 2GHz
Cache coherence protocol	MESI
L1 data cache	8-way 32KB, 4 cycles
L2 cache	4-way 256KB, 12 cycles
L3 cache	16-way 2MB, 40 cycles
DRAM	DDR4@2400Mhz, tRCD/tCL/tRP=14/14/14ns tRAS/tWR=32/15ns
PM	512 bytes write pending queue, 4ns latency; 150ns read latency; 500ns write latency
new components	
Log buffer	1,216 bytes in total
Signature	4 signatures, each is 256 bytes

512 bytes [49]. The evaluation also studies the performance implication on devices with longer write latency, such as byte-addressable SSD that aims for memory expansion [8], [9].

C. Evaluated Schemes

We compare four schemes, including two configurations of the proposed hardware and two designs of state-of-the-art in-place hardware undo durable transaction:

- FG: the baseline design with only fine-grain logging feature; both log-free and lazy persistence are disabled;
- SLPMT: the full design with all features;
- ATOM [24]: a design logging at cache line granularity; it uses a log buffer to coalesce up to eight cache lines at a time; it alters the persistence domain to remove the ordering constraints of logging and store operation in the persistence domain;
- EDE [40]: a design allows logging at any granularity; we coalesce the log records as much as possible; the design removes the global barrier between logging and store operations by sorting the associated operations with modified on-core issue queue and write buffer.

We equip FG with log-free or lazy persistence to study the breakdown of the benefit. `FG+LG` and `FG+LZ` refer to the fine-grain logging hardware with log-free or lazy persistence, respectively.

D. Kernel Benchmark

The complete design (SLPMT) achieves 1.57×, 1.65×, and 1.78× speedup on average over the baseline design, ATOM, and EDE, respectively, as shown in Figure 8 (left). The speedup is mostly brought about by the 35% persistent memory write traffic reduction over the baseline, as shown in Figure 8 (right).

1) Speedup: Although lazy persistency can also eliminate logging when a logged but lazily persisted variable remains in the cache before the transaction commits, our results show that selective logging brings about more write traffic reduction than lazy persistency does. The log record about the variable can

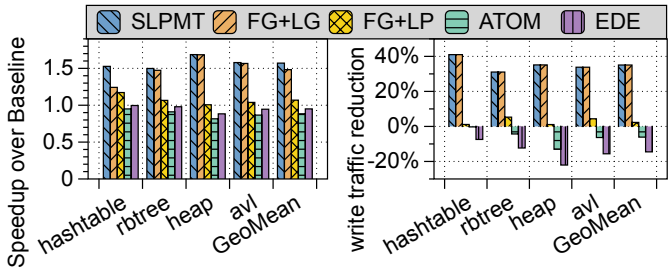

Fig. 8. Speedup over baseline (left); and persistent memory write traffics reduction over baseline (right). Higher is better.

be discarded if it has not reached the persistent domain after the transaction commits. The hardware can virtually avoid the write traffic incurred by the variable persistence because the variable can eventually get persisted, even though the persistence is not synchronous but enforced on cache overflow or transaction conflict.

Log-free and lazy persistence complement each other to bring about more gains in performance. On the *hashtable* benchmark, lazy persistence and log-free features accelerate the baseline by 17% and 24%, respectively. Together, they accelerate the benchmark by 52%. The benchmark particularly benefits from lazy persistence when it moves data on rehashing. As long as the software moves data without modifying the original data, it can lazily persist the moved data. This pattern is particularly common in incremental generational *garbage collectors* (GC), multi-version data structures, and data structure resizing. For example, an incremental generational GC copies scattered objects from one memory location to another to decrease memory fragmentation. When adding persistence to the GC, the GC can protect each object movement operation with a durable transaction that lazily persists the new object [3], as long as the old object is not deleted or overwritten inside the transaction.

SLPMT benefits from fine-grain logging besides log-free and lazy persistence. The baseline solution outperforms ATOM and EDE by 1.05× and 1.13×, respectively, showing the effectiveness of logging data at word granularity. Although EDE supports fine-grain logging, it loses opportunities for hardware log coalescing via a log buffer. Consequently, both EDE and ATOM generate more write traffic than the baseline solution does, as shown in Figure 8 (right).

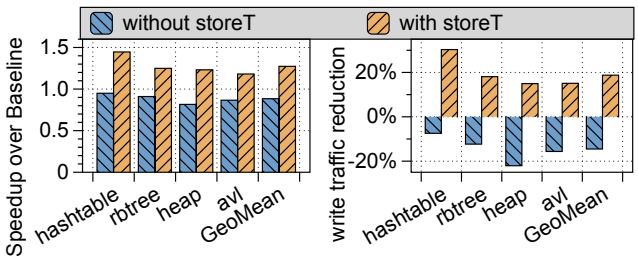

Fig. 9. Speedup (left) and persistent memory write traffics reduction over the FG baseline (right). Higher is better.

SLPMT is not bound to fine-grain logging. It still gains 1.27× speedup when logging at the cache line granularity, as shown in Figure 9. It is log-free and lazy persistent that bring about the speedup, because the hardware incurs 15% more write traffic without the features, as shown in Figure 9 (right).

2) Sensitivity Analysis to Value Size: SLPMT accelerates the baseline by 1.22× on average, even when the value is as small as 16 bytes, as shown in Figure 10. On all benchmarks, SLPMT gains on larger values as more variables can be log-free when the benchmark inserts a new value into the data structure.

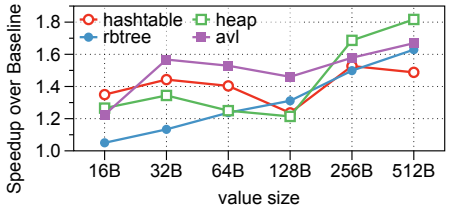

Fig. 10. Speedup sensitivity on value size

When the value is large, storing and logging new value incurs the majority of the write traffic. SLPMT eliminates unnecessary logging. Therefore, its reduced write traffic changes linearly with the value size, as shown in Figure 11. However, when the value size is small, updates on the pointers and counters dominate the write traffic; hence the write traffic reduction is mostly constant when the value size grows from 16 bytes to 32 bytes.

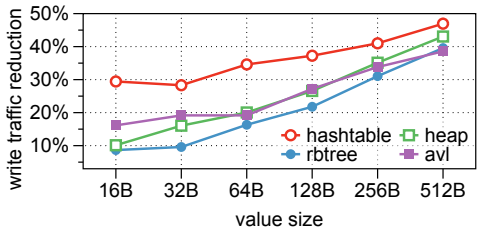

Fig. 11. Write traffic reduction sensitivity on value size. Higher is better.

3) Sensitivity Analysis to Write Latency: As CXL enables various implementations of byte-addressable persistent storage (e.g., memory backed by flash memory), persistent memory can have 600ns to 2300ns latencies [8], [9]. By reducing write traffic, SLPMT helps improve memory endurance as well as speedups, as shown in Figure 12.

For the benchmarks other than *hashtable*, the performance gain is largely stable as it is dominated by the write traffic reduction, which remains the same. *Hashtable* benefits from lazy persistence, which puts the data persistency off the critical path of transaction commits. It is hence more sensitive to the write latency.

4) Effectiveness of Compiler: The compiler-based application of storeT achieves similar speedups as the manual application does, as shown in Figure 13 (left). In particular,

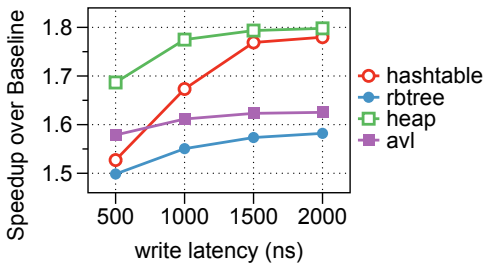

Fig. 12. Speedup sensitivity on write latency

the compiler finds out most of the log-free variables associated with new data structure entry creation. It also identifies a few lazily persistent pointer variables, such as the parent pointer of the *rbtree*. The compiler fails to infer deeper semantics of the software and hence misses the variables recording the colors or counters of the nodes. That, however, does not affect the performance as the children pointers needing early persistence may stay with the color variable in the same cache line, effectively canceling the lazy persistence. Across all the kernel benchmarks, the compiler identifies 16 out of 26 manually annotated variables.

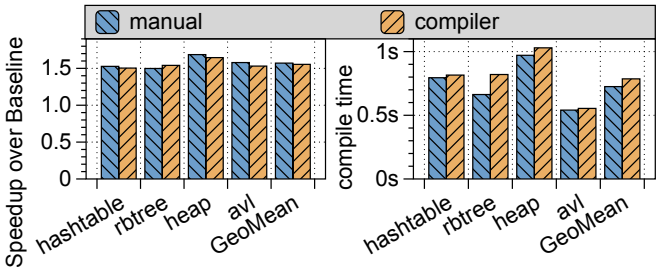

Fig. 13. Speedup over baseline (left); and compile time without or with compiler optimization over baseline (right). Higher is better.

The compiler analysis adds marginal time to the compilation time, as shown in Figure 13 (right). Whereas the compiler devotes 23% more time to compiling *btree*, the absolute difference remains less than 0.15 seconds.

E. PMKV

The speedup of SLPMT over the baseline varies on the backend data structure, as shown in Figure 14 (right). SLPMT achieves 1.35× to 1.87× speedup over EDE and 1.4× to 2× speedup over ATOM. It reduces the write traffic of the baseline by 32.6% to 47.6%. The write traffic reduction, however, is not reflected in the speedup as SLPMT reduces most write traffic on *kv-rtree* but achieves the highest speedup on *kv-ctree*. *Kv-rtree* may create more than one node in one insertion operation. It thus gives more opportunities for selective logging. The data structure, however, devotes a substantial computation time, leaving the benefits from selective logging less significant.

Rtree consists of key movement operation. The movement can be lazily persistent, as discussed in Section VI-D1. How-

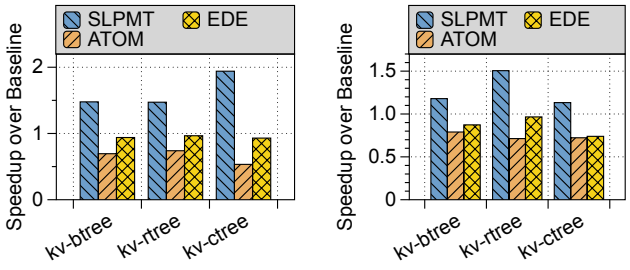

Fig. 14. Speedup over baseline. The value sizes are 256B (left) and 16B (right). Higher is better.

ever, because the evaluation uses the 8-byte key, the benefit is marginal.

When the value size is 16 bytes, SLPMT achieves less speedup than it achieves on large values. It still outperforms EDE and ATOM by 1.35× and 1.58× on average, respectively. However, most of the speedups are brought about by fine-grain logging. On top of fine-grain logging, the log-free and lazy persistence add 26.2% speedup.

SLPMT achieves a similar performance gain on *kv-rtree* despite the changes in value size as the key-value pair creation contributes less to the write traffic than other benchmarks. Therefore, the reduction in write traffic of *kv-rtree* remains largely the same compared with the case of large values.

VII. Related Work

To the best of our knowledge, this work is the first proposal that enabled selective logging on hardware persistent memory transactions.

Existing proposals of hardware persistent memory all record every updated data with logs. ATOM [24] and Proteus [22] decouple the data persisting and log persisting in accelerating transaction committing. Proteus [22] allows programmers to use two new instructions to manually create and persist logs, but still assumes that the two instructions are inserted before every store instruction. It cannot enforce lazy persistency nor ensure the order in which log and data get persisted, as shown in Figure 4. ReDU [25] utilizes log coalescing and packing for redo log in the granularity of words, reducing the write traffic to persistent memory. DHTM [21] and UHTM [23] relax the size limits of transactions, allowing them to store data with last-level cache or persistent memory.

Both selective logging and lazy persistency are ideas that have been explored in a body of *algorithm-specific* research [11], [12], [50]–[52]. This current work received inspiration from them but has a different objective in making the ideas generally applicable by integrating with hardware persistent memory transactions via architecture support. We briefly summarize some examples of those previous works as follows.

For selective logging, a prior work [12] divides the matrix into tiles, in which the granularity of the proposed solutions detects and fixes data corruptions with tiles stored on persistent memory surviving the failures. Some other work [53]–[56] designs failure-tolerant graph algorithms for distributed work scheduling or data routing. For lazy persistency [12], [14], [57], [58], Alshboul et al. [12] proposes a lazy persistency matrix multiplication algorithm that lets data naturally overflow from cache to persistent memory. It associates each chunk of updated data with a persistent checksum. On a crash, the algorithm detects whether the data reaches persistent memory by checking the checksum. It fixes the inconsistency by recomputing the data with related data. David et al. [14] proposes a link-and-persist technique for a set of fundamental algorithms such as skip list and hash table. The technique allows an updated pointer to remain volatile until it is accessed in the future.

A prior work on hardware support for fault tolerance [18] presents mixed data structures that consist of volatile pointers and persistent pointers. It rebuilds the volatile pointers from those persistent pointers. Such designs incur substantial complexity for running the algorithms on the current build of operating systems as it needs to align volatile and persistent pointers to cache lines and incurs obvious space and time overhead. TSOPER [28] proposes hardware support that decouples the persistence and the visibility of the updates on data, which essentially defers the persistency. However, the design needs non-trivial modifications, including large persistent buffers tracking the dependency between atomic groups. SLPMT in this work does not rely on any modification to the persistency domain.

VIII. Conclusion

This paper presents the first known architecture support for consolidating persistent memory transactions with selective logging capability. The new technique makes selective logging applicable to general applications and, at the same time, improves the speed of programs on by 1.8× on average. The benefits significantly reduce the bottlenecks for durable transactions working with persistent objects, bringing persistent memory closer to practical adoptions for general programs. The new architecture support, meanwhile, gives an easy way to flexibly combine selective logging and lazy persistency, opening up new opportunities for the optimizations of persistent transactions.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback that has helped improve this paper. This work was supported by the National Key Research and Development Program of China under grant No.2022YFB4500303, National Natural Science Foundation of China under grants No.62202184 and No.61825202, the National Science Foundation (NSF) under Grants CNS-2107068, 1900724, and 2106629. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] C. Ye, Y. Xu, X. Shen, H. Jin, X. Liao, and Y. Solihin, "Preserving addressability upon GC-triggered data movements on non-volatile memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 2, pp. 1–26, 2022.

[2] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 2020, pp. 680–692.

[3] ——, "Ffccd: fence-free crash-consistent concurrent defragmentation for persistent memory," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 274–288.

[4] Y. Xu, Y. Solihin, and X. Shen, "MERR: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 987–1000.

[5] Y. Xu, C. Ye, X. Shen, and Y. Solihin, "Temporal exposure reduction protection for persistent memory," in *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2022, pp. 908–924.

[6] M. Jung, "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*, 2022, pp. 45–51.

[7] C. Lee, W. Shin, D. J. Kim, Y. Yu, S.-J. Kim, T. Ko, D. Seo, J. Park, K. Lee, S. Choi, N. Kim, V. G, A. George, V. V, D. Lee, K. Choi, C. Song, D. Kim, I. Choi, I. Jung, Y. H. Song, and J. Han, "NVDIMM-C: A byte-addressable non-volatile memory module for compatibility with standard DDR memory interfaces," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2020, pp. 502–514.

[8] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2B-SSD: the case for dual, byte-and block-addressable solid-state drives," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 425–438.

[9] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 971–985.

[10] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 336–349.

[11] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[12] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 439–451.

[13] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin, "Efficient checkpointing with recompute scheme for non-volatile main memory," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 2, pp. 1–27, 2019.

[14] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *Proceedings of the USENIX Annual Technical Conference*, 2018, pp. 373–386.

[15] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 462–477.

[16] S. Haria, M. D. Hill, and M. M. Swift, "MOD: Minimally ordered durable datastructures for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 775–788.

[17] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "ROART: Range-query optimized persistent ART," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, 2021, pp. 1–16.

[18] P. Mahapatra, M. D. Hill, and M. M. Swift, "Don't persist all: Efficient persistent data structures," *arXiv preprint arXiv:1905.13011*, 2019.

[19] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-NVM: log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 346–359.

[20] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 258–270.

[21] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: Durable hardware transactional memory," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 452–465.

[22] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[23] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020, pp. 525–538.

[24] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic durability in non-volatile memory through hardware logging," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2017, pp. 361–372.

[25] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 520–532.

[26] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, 2002.

[27] P.-A. Tsai, Y. L. Gan, and D. Sanchez, "Rethinking the memory hierarchy for modern languages," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 203–216.

[28] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras, "TSOPER: Efficient coherence-based strict persistency," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2021, pp. 125–138.

[29] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 261–272.

[30] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 227–238, 2006.

[31] "Westmere microarchitectures," https://en.wikichip.org/wiki/intel/microarchitectures/westmere_(client).

[32] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[33] "LLVM memorySSA analysis," https://llvm.org/docs/MemorySSA.html.

[34] "Find your leaked persistent memory objects using the persistent memory development kit," https://www.intel.com/content/www/us/en/developer/articles/code-sample.

[35] B. Choi, R. Burns, and P. Huang, "Understanding and dealing with hard faults in persistent memory systems," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 441–457.

[36] "GCC's implementation of STL list," https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_list.h.

[37] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[38] M. Hicks, "Clank: Architectural support for intermittent computation," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 228–240, 2017.

[39] S. Scargall, *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[40] T. Shull, I. Vougioukas, N. Nikoleris, W. Elsasser, and J. Torrellas, "Execution dependence extension (EDE): ISA support for eliminating fences," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021, pp. 456–469.

[41] M. Cai, C. C. Coats, and J. Huang, "HOOP: efficient hardware-assisted out-of-place update for non-volatile memory," in *Proceedings*

of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, 2020, pp. 584–596.
[42] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.
[43] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, "Hardware-based address-centric acceleration of key-value store," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2021, pp. 736–748.
[44] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.
[45] "Persistent memory key-value store, as of aug 1, 2022," https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/map.
[46] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
[47] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.
[48] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, "Supporting legacy libraries on non-volatile memory: a user-transparent approach," in *Proceedings of ACM/IEEE 48th Annual International Symposium on Computer Architecture*. IEEE, 2021, pp. 443–455.
[49] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020, pp. 496–508.
[50] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 895–906.
[51] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, 2011, pp. 73–84.
[52] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Proceedings of the International Symposium on Distributed Computing*, 2016, pp. 313–327.
[53] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 719–730.
[54] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Transactions on Software Engineering*, no. 6, pp. 585–594, 1980.
[55] M.-S. Chen and K. G. Shin, "Depth-first search approach for fault-tolerant routing in hypercube multicomputers," *IEEE Transactions on Parallel & Distributed Systems*, vol. 1, no. 02, pp. 152–159, 1990.
[56] M. Parter and D. Peleg, "Sparse fault-tolerant BFS structures," *ACM Transactions on Algorithms*, vol. 13, no. 1, pp. 1–24, 2016.
[57] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank, "Efficient lock-free durable sets," in *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019, pp. 1–26.
[58] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei, "Delay-free concurrency on faulty persistent memory," in *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 253–264.