Hiding the Long Latency of Persist Barriers Using Speculative Execution

Seunghee Shin James Tuck Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University, NC, USA
{sshin6,jtuck,solihin}@ncsu.edu

ABSTRACT

Byte-addressable non-volatile memory technology is emerging as an alternative for DRAM for main memory. This new *Non-Volatile Main Memory* (NVMM) allows programmers to store important data in data structures in memory instead of serializing it to the file system, thereby providing a substantial performance boost. However, modern systems reorder memory operations and utilize volatile caches for better performance, making it difficult to ensure a consistent state in NVMM. Intel recently announced a new set of persistence instructions, clflushopt, clwb, and pcommit. These new instructions make it possible to implement fail-safe code on NVMM, but few workloads have been written or characterized using these new instructions.

In this work, we describe how these instructions work and how they can be used to implement write-ahead logging based transactions. We implement several common data structures and kernels and evaluate the performance overhead incurred over traditional non-persistent implementations. In particular, we find that persistence instructions occur in clusters along with expensive fence operations, they have long latency, and they add a significant execution time overhead, on average by 20.3% over code with logging but without fence instructions to order persists.

To deal with this overhead and alleviate the performance bottleneck, we propose to speculate past long latency persistency operations using checkpoint-based processing. Our speculative persistence architecture reduces the execution time overheads to only 3.6%.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → **Memory and dense storage**;

KEYWORDS

Non-Volatile Main Memory, Speculative Persistence, Failure Safety

ACM Reference format:
Seunghee Shin James Tuck Yan Solihin Dept. of Electrical and Computer Engineering North Carolina State University, NC, USA . 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution . In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 12 pages.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
http://dx.doi.org/10.1145/3079856.3080240

http://dx.doi.org/10.1145/3079856.3080240

1 INTRODUCTION

Non-volatile memory technologies are advancing and some of them are competing with DRAM for use as a future main memory. For instance, Intel and Micron announced that their 3D Xpoint memory will be in the market in 2017 [21]. These new *non-volatile main memory* (NVMM) technologies are byte-addressable and have reasonably fast access latencies [1, 5, 6, 21, 23, 27, 29, 38]. Because NVMM are accessed using regular loads and stores, programmers can store important data in data structures in memory instead of serializing it to the file system.

While avoiding the file system can potentially provide a large performance advantage, it introduces a major challenge of ensuring the consistency of data across unexpected hardware or software failures. In a conventional system, the order in which store values are written back to the main memory from the last level cache (LLC) does not follow the program order, so a failure produces an unpredictable outcome of which stores have their values permanently reflected (i.e. *durable*) in the NVMM. Thus, to achieve *failure safety*, two critical components are required: (1) a specification or model of *persists* (i.e. when stores will be durable in the NVMM), and (2) rewriting of software so that it can recover safely upon a failure based on the persistency model.

Various persistency models have been proposed [4, 12, 22, 30, 37], including the use of transactions [25, 45]. Our starting point is Intel PMEM [4], a persistency model supported by several new instructions, such as *clwb*, *clflushopt*, and *pcommit*, used in conjunction with existing x86 instructions such as *clflush* and *sfence*. *Clflushopt* and *clwb* force dirty data out of the cache hierarchy, while *pcommit* acts as a persist barrier by forcing a flush of the write buffers in the memory controller. These instructions may be re-ordered by the processor with respect to non-dependent loads and stores, hence fence instructions are needed to precisely control when data is made durable at the NVMM. While these instructions have been announced, their performance impact has not been studied or characterized in the context of NVM workloads, and few workloads have been written using these new instructions. Similarly, ARM has recently introduced a new instruction, DC CVAP, for persistence support in ARMv8.2 [3], that cleans and flushes a cache block to the persistence domain.

To gain a deeper understanding of the programming and performance challenges of persistent data structures on future processors and applications, we rewrite several data structures and kernels commonly found in databases and file systems to incorporate failure safety using the PMEM persistency model. Like prior work [25, 45],

we adopt transactional semantics using write-ahead logging as the basis for reasoning about failure safety. After building and using a microarchitectural simulator extended to support PMEM, we make several observations. First, write-ahead logging requires frequent use of *sfence-pcommit-sfence* instruction sequences to correctly order persists in these benchmarks. Second, the PMEM instructions occur in clusters with fence operations, have long latencies, and incur a significant performance penalty, causing execution time overheads of up to 122%, and 28% on average, on top of logging overheads. The performance penalty arises primarily due to the pipeline stalling for the completion of the *sfence-pcommit-sfence* instruction sequence.

Based on these observations, we propose *speculative persistence* (SP), architectural support to speculatively execute past long latency persist barriers to hide their latency and reduce their impact on performance. Speculative execution is triggered when a persist barrier stalls the pipeline. The analysis of our benchmarks shows that such barriers can take 100s to 1000s of cycles to complete. Rather than waiting, in SP, a checkpoint is taken at the persist barrier, the sfence is speculatively retired, and the processor proceeds speculatively retiring the sfence and following instructions. Meanwhile, the pcommit completes non-speculatively in the background. Stores are buffered and not allowed to propagate to memory until the pcommit finishes and speculation completes. Buffering of speculative state and conflict detection proceed in much the same way as prior speculation schemes.

In the past, speculative execution has been proposed for other purposes such as hiding L2 miss latency [8, 24, 34, 42], improving memory consistency model performance, and speculating past synchronization operations [19, 31, 39]. While bearing some similarities with prior speculation techniques, SP faces new challenges. First, other persist barriers are likely to occur in the shadow of the current speculative persist barrier. Because these instructions may need to flush data out of the cache, they cannot execute as part of the speculative region and must instead be buffered and played back at commit of the speculative region. Second, because some instructions must be delayed and played back later, this limits how far we can speculate. To overcome these challenges, we leverage multiple checkpoints to speculate across multiple persist barriers. The speculative regions commit sequentially as each pending persist barrier completes in sequence, thereby ensuring proper transactional semantics.

We evaluate our new architecture and compare it against the same system without speculation. Our experiments show that SP reduces the execution time overheads to only 3.6% on average, compared to code with logging and PMEM instructions but without fences to enforce ordering constraints.

The remainder of the paper is organized as follows. Section 2 provides background on memory persistency, and describes the new PMEM instructions. Section 3 describes the new workloads we created using write-ahead logging, Section 4 describes our new architecture and its implementation, Section 5 describes the evaluation methodology, and Section 6 evaluates our design and presents our key findings. Section 7 discusses related work to speculation. Section 8 concludes this work.

2 BACKGROUND

2.1 Memory Persistency Models

A modern processor design relies on multiple levels of caches and out-of-order instruction execution. In such a system, the order in which writes are made durable to the NVMM is unpredictable and does not follow program order, as it depends on the order of write backs from the last level cache (LLC) to the NVMM. Upon failure, values in the cache (which may correspond to older stores) are lost, while values in the NVMM (which may correspond to younger stores) are not. To achieve *failure safety*, programmers need precise specification of when stores will be durable in the NVMM, referred to as the *persistency model*, in order to reason about how they need to rewrite their software to achieve failure safety of their code.

Previous research proposed several memory persistency models, *strict persistency*, *epoch persistency*, *buffered epoch persistency*, *strand persistency*, and *transactional persistency*. Strict persistency [37] piggybacks the sequential consistency model, by specifying that a store that is globally visible must also have persisted in NVMM. Due to this constraint, before each store persists to NVMM, all previous stores must have persisted to NVMM. The easiest way to implement strict persistency is using a write through cache hierarchy for stores. Since any visible stores are persisted, failure safety reasoning is easier for programmers. However, it comes with significant performance costs of not allowing write reordering and write coalescing that naturally occur in a write back cache.

Epoch persistency relaxes some of the ordering constraints of strict persistency [12, 37]. Epoch persistency allows the programmer to put persist barriers that define epochs. Stores from one epoch (i.e. between two persist barriers) can persist in any order, but they must all persist at the next persist barrier. Write coalescing can occur for stores from the same epoch. At the persist barrier, the processor may stall waiting for all stores from the epoch to persist. In a related buffered epoch persistency model [12, 22], a persist barrier does not force prior stores to persist right away (hence the processor may not stall), as long as stores from one epoch persist prior to any stores from the next epoch.

Epoch persistency relaxes the ordering of persists within epochs but enforces order across epoch barriers. Strand persistency [37], on the other hand, relaxes the ordering constraints for persists separated by a *strand barrier*. No ordering is enforced on persists in different strands other than those implied by persist atomicity.

Transactional memory persistency relies on transactions as a unit of persistency [25], where all stores in a transaction either persist together or not at all. The study also proposed using an extra buffer and cache bypassing for persists in every core as performance optimizations. Copy-on-write can be another way to implement a transaction with checkpoints and Rei et al. adopt it in their proposed systems using NV memory for checkpoint [41].

2.2 Intel PMEM Persistency Model

Intel recently announced a new PMEM persistency model in pmem.io [4], based on several new instructions, such as *clwb*, *clflushopt*, and *pcommit*[1], used in conjunction with existing x86 instructions such as *clflush* and *sfence*.

[1] We are aware that Intel has deprecated pcommit [20]. In our work, we do not assume that the memory controller is part of the persistency domain, so pcommit is still required.

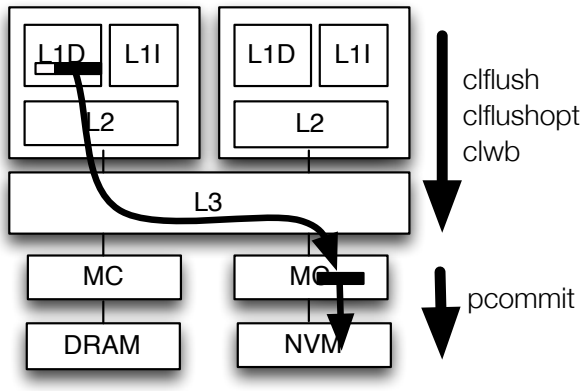

Figure 1: Intel persistence instrcutions

Compared to other persistency models, PMEM instructions provide a more flexible approach towards memory persistency. It allows programmers to construct other persistency models such as strict and epoch persistency (if desired), but it also allows programmers to select which stores need to persist and in which order they persist.

Figure 1 illustrates the PMEM instruction operations. The *clflushopt*[2] and *clwb* instructions clean dirty cache blocks from the cache hierarchy by writing them back, and in the case of clflushopt, the blocks are also evicted. After dirty data is evicted from the cache, it may be waiting in a buffer in a memory controller. The dirty data is not durable yet until it is removed from the buffer and written to the NVMM. The *pcommit* instruction is used to force durability of such pending write backs, by forcing the memory controllers to flush all pending writes. When all buffers have been flushed, each memory controller sends an acknowledgement back to the core that executed the pcommit, indicating the completion of pcommit.

By appending clflushopt or clwb, and pcommit after stores that need to persist, programmers can selectively choose which stores should persist, and when they persist. This flexibility allows programmers to persist only stores to recovery-critical data, instead of stores to all data. It also permits write coalescing. These features are critical to achieving low overhead failure safety.

However, using PMEM instructions (clflushopt, clwb, and pcommit) by themselves is not sufficient because there is no ordering guarantee with respect to other stores or other persistence instructions, except for some implicit dependences on the same address. Clflushopt is ordered with respect to other stores on the same cache block, however clwb is only ordered with respect to stores to the same address. Otherwise, to force an ordering among two of these instructions, a store fence instruction (sfence) is required. Sfence, which was originally an instruction used for memory consistency, takes up an additional role in that when placed after PMEM instructions, it will wait for all pending PMEM instructions to complete before retiring, and prevent following stores and PMEM instructions from executing until it completes.

[2] *clflush* has a similar functionality but much worse performance, so we do not use it in our study.

To illustrate an example of the usage of the PMEM instructions, suppose that we wish to persist a store to address X before modifying address Y. Then, we would write:

```
i1: st X, 1;
i2: clwb X;
i3: sfence;
i4: pcommit;
i5: sfence;
i6: st Y, 1;
```

Instruction i2 (clwb) forces the store X to be written back from the cache, instruction i4 (pcommit) persists the store to the NVMM. The first sfence (i3) stalls the pcommit until the write back is complete. Next, the second sfence (i5) makes the update to X durable, and forces store Y (i6) to wait until the data block containing X is truly durable. pcommit's completion is detected when the write buffers in the memory controller are flushed and the processor has received acknowledgement from all memory controllers. Thus, pcommit may take a long time to complete, and becomes the major performance bottleneck, which we strive to address in this paper.

In our *speculative persistence*, the processor takes a checkpoint and enters speculative region, and then speculatively retires the sfence instruction i5, and proceeds executing instruction i6 and beyond. The processor exits the speculative region when the pcommit completes successfully.

3 WORKLOADS

In the previous section, we have discussed PMEM instructions that make up a flexible building block of memory persistency that programmers can control. In this section, we discuss how we utilized the instructions to develop the workloads that we use in our evaluation. We focus on single-threaded data structures and algorithms commonly used in databases and file systems. To achieve failure safety, we use a write-ahead logging transaction approach. We wish to investigate failure safety performance issues for single-threaded programs, and leave multi-threaded programs for future work.

3.1 Failure Safety through Transactions Using Write-Ahead Logging

Failure safe updates to non-volatile storage have long been achieved through transactions [32]. We believe that future software must update NVMM using transactions as well. It may be implemented directly in software by programmers or provided through a compiler or library [45]. We implement transactions directly in software as part of the normal operation of data structures using a form of write-ahead logging [32].

The general strategy of write-ahead logging is to make an undo log of all desired changes to memory before making any modifications. That way if a failure occurs in the middle of the update, then the undo log can be used for recovery. The general sequence of steps to complete the transaction on NVM are as follows:

Step 1 Perform undo-logging. Make the undo-log updates durable.
Step 2 Logged_bit is set and made durable, indicating a transaction has begun.
Step 3 Commit updates to the memory and make them durable.
Step 4 Logged_bit is unset and made durable, indicating the transaction is complete.

To understand how this works, we must consider the state of memory after a failure. The logged_bit indicates whether a transaction is in progress or not. If the logged_bit is 0, no transaction was in progress when the failure occurred and the current data structure is reliable. Otherwise, the logged_bit is 1, and a transaction was in the middle of processing. In this case, the undo log will be used to recover the state of the data structure. Because we do not know at what step the failure occurred, we must pessimistically recover using the undo log regardless.

To ensure correctness of the above, the 4 steps must be strictly ordered. If Step 2 begins before Step 1 completed, the premature undo log may be erroneously applied to the data structure. Likewise, if commits began before the logged_bit were set, then an incomplete set of updates may go undetected. We can enforce the necessary ordering between these steps using persist barriers, the sfence-pcommit-sfence discussed in the previous section. Given that each step needs a persist barrier, this implies that at least 4 pcommits and 8 sfence operations are needed per transactional update to NVM. Also, compared to a volatile data structure, the added cost of undo-logging will also be a significant overhead.

3.1.1 Detailed Example: A Non-Volatile Linked List. Figure 2 shows an example of our transactional code for a linked list. We choose a linked list code for illustration due to its simplicity (e.g., compared to balanced trees). Before making any updates in the linked list, the modified nodes need to be logged. In the example, a new node 'temp' needs to be inserted after node 'nn'. Hence, we log data of node 'nn' and the address of 'nn'. After the logging is completed, a logged_bit is set. If the system crashes during the transaction, if the logged_bit is set, the transaction is undone by overwriting the original location using the logged data. If the bit is unset, the data structure is consistent as modifications to the linked list have not occurred. The example in the figure shows that one transaction requires four pcommits with several sfences and clwb. Since the data structure is stored in the NVMM, a system crash may result in the inconsistent data structure. In Figure 2, if the system crash happens after line 28 but before line 29, the data structure becomes inconsistent, resulting in the linked list shown in Figure 3. This illustrates the importance of the use of a transactional approach, which in our case is supported through write-ahead logging.

3.2 Workload Construction

Using the write-ahead logging approach discussed above, implemented using PMEM persistency instructions, we constructed a workload consisting of benchmarks with data structures listed in the Table 1. They are similar to ones used in previous studies [11, 49]. For each benchmark, we construct an *operation*, which is either a node insertion or deletion (except for String Swap). An operation performs searching of a random key in the data structure. If the key is found, the node with the key is deleted. If the key is not found, a new node with the key is inserted. String Swap simply selects two keys indicating two strings in a string array and swaps them. The last two columns of the table will be explained in Section 5.2.

```
01  log_index =0;
02  log_op = OP_INSERT;
03  memcpy(&LOG[log_index], nn, sizeof(struct node));
04  LOG_ptr[log_index] = nn;
05
06  SFENCE;
07  CLWB(&LOG[log_index]);
08  CLWB(&LOG_ptr[log_index]);
09  log_index++;
10  CLWB((void*)&log_index);
11  CLWB((void*)&log_op);
12  SFENCE;
13  PCOMMIT;
14  SFENCE;
15
16  logged = 1;
17
18  SFENCE;
19  CLWB ((void*)&logged);
20  SFENCE;
21  PCOMMIT;
22  SFENCE;
23
24  temp=(struct node *)malloc(sizeof(struct node));
25  temp->node_key=num;
26  memcpy(temp->data, val, DATASIZE);
27  nn->next=temp;
28  nn=temp;
29  nn->next=right;
30
31  SFENCE;
32  CLWB(nn);
33  CLWB(temp);
34  SFENCE;
35  PCOMMIT;
36  SFENCE;
37
38  logged = 0;
39
40  SFENCE;
41  CLWB((void*)&logged);
42  SFENCE;
43  PCOMMIT;
44  SFENCE;
```

Undo logging
Set logged_bit
Transaction
Unset logged_bit

Figure 2: Persistence example with linked list

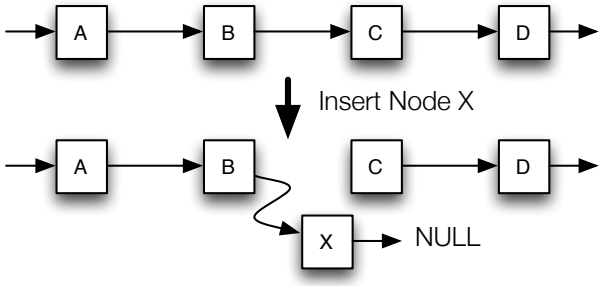

Figure 3: Inconsistent data structure example with linked list

Table 1: Benchmarks constructed for our study. For all benchmarks, we size each node to be 64 bytes and align them to cache blocks. Thus, to persist one node update, one clwb will be required.

Benchmark (Abbrev.)	Description	#InitOps	#SimOps
Graph (GH)	Insert or delete edges in a graph	2600000	100000
Hash-Map (HM)	Insert or delete entries in a hash map	1500000	100000
Linked-List (LL)	Insert or delete nodes in a linked list (Max:1024)	500	50000
String Swap (SS)	Swap strings in a string array	120000	500000
AVL-tree (AT)	Insert or delete nodes in an AVL tree	1000000	50000
B-tree (BT)	Insert or delete nodes in a B tree	1000000	50000
RB-tree (RT)	Insert or delete nodes in an RB tree	1500000	50000

We found that there are two types of data structures depending on the number of nodes involved in update operations. Hashmap, linked list, and graph belong to the first type of benchmarks with few nodes involved in an update operation. They have very small overheads from the undo logging code. In Hashmap, an operation uses a hash function to map a key to an index to a hash table entry. We use a chained collision policy, so that if the entry is already populated, the next consecutive entry is checked, and so on. Once a free entry is found, the operation logs the location and the size of the hash table before inserting the new record. If the key is found, the entry is logged before deleting the record from the entry. As discussed earlier, after logging, the logged_bit is set to 1 and once the insertion or deletion is complete, the bit is reset. In Hashmap, if no free entry is found for insertion, the table is resized. In this case, we create a new table twice the size of the original table and move all records from the original table to the new one. During record copying, each insertion is followed by clwb. pcommit persists the completion of the resizing. Other data structures, LinkedList and Graph are implemented in a similar manner, but without data structure resizing and copying. In the case of String Swap, we get two random indexes to swap in a string array. The length of each string in the entry is 256. Before swapping, an operation undo-logs two strings in an indexed entry. After the logging, eight clwbs are issued for logging entries and one clwb is for indexes. After the swap is completed, another eight clwbs are issued along with pcommit.

Self-balancing trees such as AVL tree, B tree, and RB tree, belong to the second type of benchmarks with a variable numbers of nodes involved in each update operation. Due to tree rebalancing, logging becomes more complex and has high overheads. Sometimes, all nodes in a root-to-leaf path are involved in an operation.

In a self-balancing tree, after an insertion or deletion of a node, the tree balance property is checked. If the tree is no longer balanced, rebalancing is triggered. Here, we face a design choice of how to transactionalize the tree balancing to achieve failure safety: incremental logging or full logging.

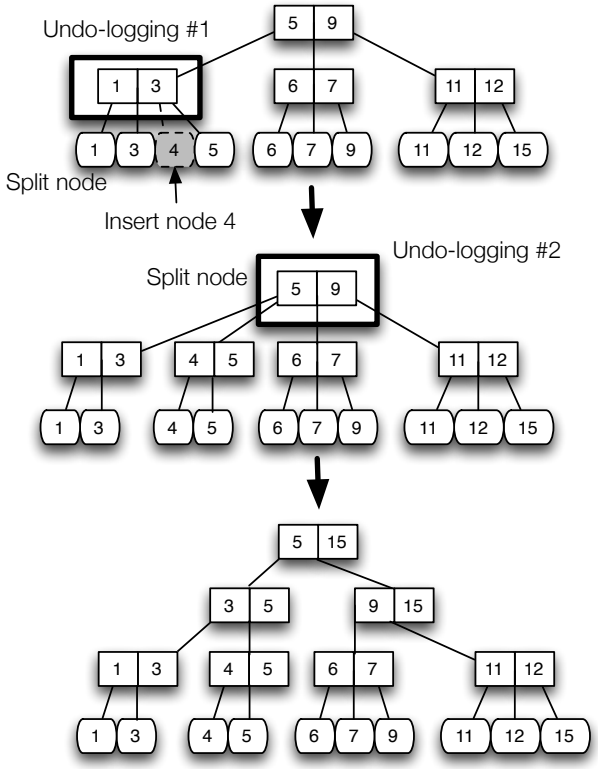

Figure 4: Rebalancing with incremental logging.

Incremental logging breaks rebalancing into multiple steps, where in each step we log as few nodes as needed to perform balancing for a particular affected node. In many cases, each rebalancing only involves one or two levels. In some cases, the rebalancing is escalated to higher levels of the trees. Hence, the nodes involved in each rebalancing step can be made a small logging unit. Figure 4 illustrates the *incremental logging* approach on a 2-3 B-tree. A 2-3 B-tree is a sorted balanced tree where each non-leaf node can have anywhere between two and three children nodes. Data is stored in the leaf nodes, while non-leaf nodes store keys to accelerate searching. In order to insert a new node, the tree is traversed recursively from the root to the appropriate leaf based on the new node's key.

The top diagram of the figure shows a leaf node 4 is about to be inserted into the tree, which will result in the node (1,3) with too many children, which will require node (1,3) to be split to keep the tree balanced. In the incremental logging approach, node (1,3) is logged prior to insertion. After leaf node 4 is inserted, rebalancing is triggered. Next, node (5,9) is logged and node (1,3) is split. This rebalancing is escalated until the tree is balanced. If the system crashes in the middle of rebalancing, the recovery uses the log to make the tree consistent, then continues to rebalance the tree. The advantage of the incremental logging approach is that only necessary nodes are logged because it performs undo-logging just before updating the

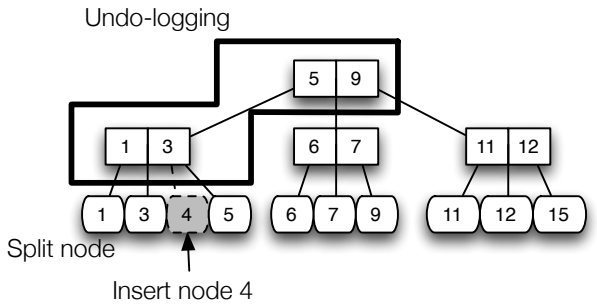

Figure 5: Rebalancing with full logging.

node. If the update doesn't trigger rebalancing, the operation can be performed quickly. However, the tree may be temporarily imbalanced when a failure occurs. Furthermore, pcommits and sfences are required for each step.

Given the programming complexity and the frequent persist barriers of incremental logging, we select the *full logging* approach for our workloads, where we conservatively log all nodes that may be required for rebalancing on a given operation. Figure 5 illustrates the entire root-to-near-leaf nodes are logged prior to inserting node 4. If rebalancing is triggered, no additional logging is required. The advantages of this approach are that it only requires a set of four pcommits regardless of whether rebalancing is triggered or not, and the tree is always balanced regardless of when a failure occurs. However, this approach requires logging of additional nodes that may or may not be modified, and this may be expensive if the tree is large.

4 SPECULATIVE PERSISTENCE

4.1 Architectural Requirements

As seen in the linked list example, operations on persistent data structures tend to require frequent and clustered sfence, pcommit, and clwb instructions. These instructions in conjunction with sfences incur long pipeline stalls. Based on this observation, we propose to reduce the pipeline stalls by executing speculatively. Rather than letting the sfence stalling the processor while waiting for a pcommit to finish, we propose to checkpoint the architectural state[3] and retire the sfence speculatively. This lets the processor continue executing and retiring instructions speculatively, while the pcommit completes in the background non-speculatively. Note, during speculative execution we must buffer speculative stores and prevent them from updating memory, as in prior speculation schemes [8, 9, 16, 18]. Since we speculate on the successful completion of the persist barrier and all stores prior to the barrier, we call our scheme *speculative persistency* (SP). SP allows the overlap of the long latency pcommit operation with the instructions that follow.

SP differs from typical speculations in a few unique ways, necessitating a novel design. First, we use transactions for fail safe persistence, and not for concurrency management. SP does not constraint the use of concurrency management, i.e. it is possible to use SP in conjunction with locks for concurrency management. Second, in SP, speculation failure occurs not due to conflicting accesses, but due to serious system failures (system crash, irrecoverable NVMM errors, etc.). Thus, the probability of speculation failure is very low and rollback can be expected to be extremely rare. This situation may change if we target multi-threaded workloads, depending on what transactions are used for, and the persistency and memory consistency models. However, we note that our study targets sequential programs, and a large class of applications today are still sequential, so it is important to accelerate them. Multi-threaded workload and issues are left for future work. One consequence of the low rollback rate is that rollback and recovery time is much less important than the speed of executing the speculative region. Another consequence is that speculation is less risky, hence we must attempt it as long as there are still opportunities to do so.

The final unique situation for SP is that PMEM instructions (pcommit, clwb, and clflushopt) themselves cannot be executed speculatively, because for them to be considered complete, they have to go to memory. Once they go to memory, they are no longer speculative and can no longer be rolled back. In contrast, regular store instructions can be buffered and considered complete. This is a major problem since, as illustrated in the linked list example, they occur close together and in clusters. Most speculation frameworks simply stop at such instructions. For SP to be effective, we must figure out a new design not to allow them to prevent speculation.

So far we have discussed the unique challenges SP faces, now we will discuss the approach we take in designing SP. First, we need to define the period of speculative execution, from the fence to the last speculative instruction, as a *speculative epoch*. The speculative epoch begins at an sfence and it ends when the sfence instruction would have otherwise retired from the processor. For example, it would retire when it receives acknowledgement from the memory controller that its buffers have been flushed. At the moment it would have retired, its checkpoint is discarded because it is no longer needed, and its pending memory operations are allowed to complete and update memory. Also, the core returns to a non-speculative mode of execution.

As discussed above, if PMEM instructions are encountered in the speculative epoch, they cannot be executed speculatively. To partially address this problem, we propose that persistent operations be delayed until the end of the speculative epoch, at which time they execute as quickly as possible. Depending on resource constraints and the number of such operations, some serialization may be inevitable.

This choice implies a re-ordering of the PMEM instructions with respect to loads and stores within the epoch. More precisely, given a point in the epoch where the persistent operation occurs, *p*, and the last instruction that is part of the epoch, *end*, then the operation is re-ordered with respect to all memory operations between *p* and *end*. This is architecturally allowed for PMEM instructions. Both pcommit and clwb have flexible ordering policies, allowing re-ordering with respect to all instructions other than mfence, sfence, xchg, or LOCK-prefixed instructions[4]. As long as instructions with strong ordering constraints are not present, the re-ordering is allowed. Hence,

[3]The checkpoint is taken by hardware and stored in registers, and it includes the register file, the PC, and any other register required to restore execution back to that point on a rollback.

[4]pcommit is also ordered with respect to serializing instructions, cpuid for example.

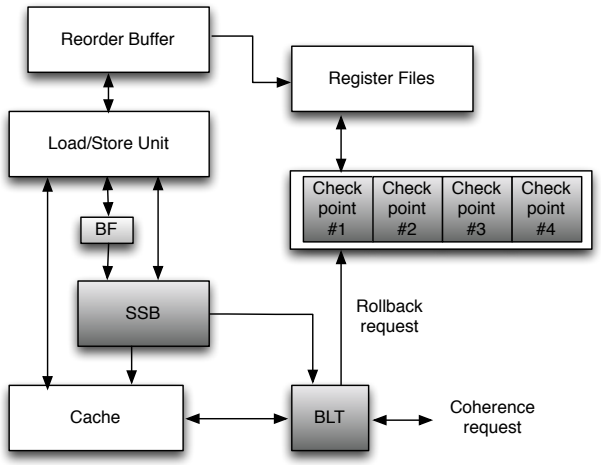

Figure 6: Speculative persistence design.

we require that epochs only contain PMEM instructions that may be legally delayed until the end of the epoch.

This requirement has an important implication. Clwb, clflushopt, and pcommit are not ordered with respect to loads or stores, so these operations can be moved to the end of a speculative epoch arbitrarily. However, they may not be moved past fences, LOCK-prefixed instructions, or XCHG. Hence, these instructions necessarily form boundaries for speculation, and we must end an epoch at these instructions. Even if we delay PMEM instructions, we cannot extend a speculation past an sfence if it already includes other PMEM instructions.

Given our workload characterization, we know that sfence-pcommit-sfence sequences are likely in the code following the first pcommit. If we are forced to stop speculating at the first sfence, speculation will not proceed far. To overcome this, we propose to use multiple speculative epochs.

Whenever we reach an instruction that cannot be re-ordered with respect to a prior PMEM instruction, we create a new checkpoint and begin a new speculative epoch. Such an epoch will be referred to as a *child* epoch. For each epoch, the architecture must track its speculative state and PMEM instructions separately.

Futhermore, speculative epochs must commit in sequence from the oldest to the youngest child to ensure that all ordering constraints between speculative epochs are enforced. This additionally implies that all of an epoch's PMEM instructions must commit and complete before any operations from the next speculative epoch can commit.

4.2 Implementation

Figure 6 provides a block diagram of our architecture. To a conventional superscalar processor, we add support for speculation. When a speculative epoch begins, a checkpoint is captured by copying the values of all architectural registers into one of the checkpoints, and the sfence which triggers the checkpoint is removed from the ROB. Speculative stores and PMEM instructions are removed from the ROB at retirement and placed in a *Speculative Store Buffer* (SSB) until the epoch commits.

Meanwhile, load instructions that follow must check the SSB to satisfy memory dependences with previously retired speculative stores. To ensure loads are handled efficiently, we add a *Bloom Filter* (BF) to summarize the contents of the SSB and avoid SSB lookups when possible. The *Block Lookup Table* (BLT) shields the speculative state from becoming visible to other cores.

4.2.1 Speculative Epochs. In general, we can divide the execution of an epoch into three points: the first instruction where it begins, the last instruction of the epoch, and the point in time at which it commits its speculative state.

In our architecture, speculation begins when an sfence following a pcommit is waiting for acknowledgements from the memory controller(s). This would constitute the first epoch in a group of epochs. The first epoch may end either when the pcommit's acknowledgements are received or in the creation of a *child* epoch. If the pending pcommit completes before the creation of a child epoch, then the epoch ends and commits at the same time. The checkpoint is discarded and all state in the SSB is allowed to proceed with updating the cache. Meanwhile, the processor resumes non-speculative execution.

However, it is more likely that the first epoch ends by creating a child epoch. In such a case, a child epoch begins on an instruction with a strong ordering requirement, such as a fence instruction, XCHG, or other LOCK-prefixed instructions, since these cannot be re-ordered. The hardware creates the child epoch by taking a new checkpoint using the current speculative architectural state. The child epoch then begins executing.

After the creation of a child epoch, the previous epoch is done executing, but it may not yet be ready to commit. If it is the first epoch, then it must still wait for the pending pcommit to complete. If it is not the first epoch, then it must wait for its predecessor to complete. Once an epoch's predecessor fully commits its speculative state, the child can become non-speculative and commit its state as well. This process occurs repeatedly until all child epochs commit their state and the processor resumes non-speculative execution.

In the event that a child epoch is needed but no checkpoints are available, the processor must stall and wait for a checkpoint to become free.

4.2.2 Speculation using the SSB. To support epoch execution, the SSB will hold speculatively retired stores and PMEM instructions for each epoch. The SSB is a queue and maintains the order of stores and the order of PMEM instructions within an epoch and across epochs. Each entry of the SSB contains an opcode, address, the data if a store, and a checkpoint number to identify which speculative epoch it belongs to. This information enables the SSB to track which epoch the instruction belongs to and when it should commit.

When an epoch commits, the instructions in the SSB update the cache or memory in sequence as quickly as possible depending on the availability of ports to the cache. Keep in mind that speculative epochs may only contain instructions which can be legally re-ordered to the end of the epoch, so re-executing all stores and PMEM instructions at the end is sufficient to safely commit the speculative epoch.

Some instructions and instruction sequences need special support. Consider the case of the sfence-pcommit-sfence instruction sequence. In a naive implementation, each fence creates a child epoch and

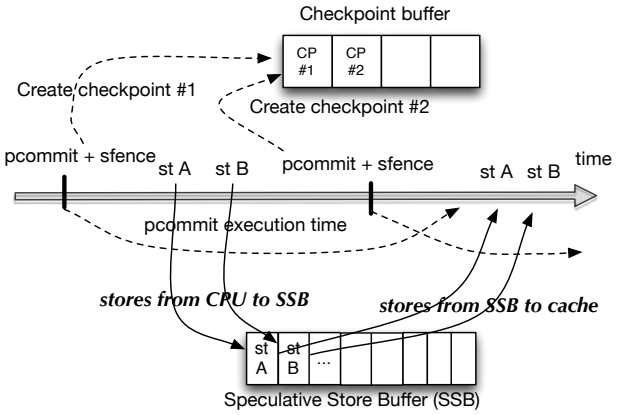

Figure 7: Speculative persistence

its own checkpoint, but it would be wasteful to devote an entire checkpoint to a single pcommit instruction. Checkpoints are a limited resource, so we optimize this case for efficiency. Instead of forming two checkpoints, we form a single checkpoint only for the last sfence, and we place a special opcode in the SSB indicating that an sfence-pcommit-sfence is required before committing the next epoch.

Figure 7 illustrates the SSB's operation over time. When the first pcommit+sfence is encountered, a first epoch starts and a checkpoint is taken. The next two stores (st A and st B) are placed in the SSB and retire. The second pcommit+sfence results in a second checkpoint. When the first pcommit is completed, st A and st B are released to the cache. For added efficiency, we separate the SSB from the store queue and we assume that it is only accessed by loads during speculative execution. However, as long as any stores remain in the SSB, loads must check this buffer for dependences before (or in parallel to) accessing the L1D cache. However, in our analysis, a 256-entry SSB requires 5-cycle to access, longer than a typical L1D cache. To avoid the SSB becoming a performance bottleneck, we adopt a bloom filter as in CPR [14]. In our case, the bloom filter is 512 bytes in size. It is set as a store is inserted into the SSB. The bloom filter is reset completely upon exiting speculative execution. Thus, the bloom filter can only produce false positives but not false negatives, and the periodic resets keep the false positive rates low.

While SP does not intend to address concurrency management and we purposely leave it out of the scope of this work, we need to ensure that SP does not create new persistency problems for a traditional parallel program that were not present without speculation. The scenario that can cause problems is when a core is in a speculative epoch and a coherence intervention from another core occurs. It would be incorrect to reveal speculative state to the other core, and it would be equally incorrect to allow speculation to proceed with incoherent data. Neither option is allowed. Conflicts between external coherence requests and local speculative state must be detected to trigger a rollback. To detect a conflict, as in SC++ [17], we added a block lookup table (BLT) that holds a list of all the cache block addresses accessed by both speculative loads and stores. Coherence operations are checked against the BLT; any BLT match is treated as an atomicity violation and triggers an abort and rollback. The rollback is to the oldest uncommitted checkpoint. To keep the design simple, currently our BLT design does not distinguish the addresses from different speculative epochs. If a conflict is detected, we simply roll back to the oldest checkpoint and flush the SSB and all speculative state.

5 METHODOLOGY

5.1 Simulation configuration

We implemented clflushopt, clwb, and pcommit instructions in a processor simulator built on MarssX86 [36]. MarssX86 is an open source cycle-accurate full system simulator for an x86-64 architecture. It supports detailed out-of-order CPU, cache, and memory controller models. Table 2 shows the detailed parameters and architecture configurations of the processor and memory system in our simulation. Our machine model includes an out-of-order issue core with three cache levels, backed by an NVMM with 50ns/150ns read/write latency. The NVMM latencies are similar to those assumed by prior work [26, 28, 33, 46, 48].

Table 2: The baseline system configuration.

Processor	OOO, 2.1GHz, 4-wide issue/retire ROB: 128, fetchQ/issueQ/LSQ: 48/48/48
L1I and L1D	32KB, 8-way, 64B block, 2 cycles
L2	256KB, 8-way, 64B block, 11 cycles
L3	2MB, 16-way, 64B block, 20 cycles
SSB	variable size and latency (Table 3)
Checkpoint Buffer	4 entries
NVMM	50ns read, 150ns write

In MarssX86, we implement the ordering constraints for clwb as described in the Intel manual [13], where clwb is ordered only by store-fencing operations (e.g. sfence and mfence) and older stores to the same address. Also, pcommit is only ordered by store-fencing operations. Like regular stores, clwb accesses the cache after it is retired in the CPU pipeline. The clwb becomes globally visible when the dirty cache block is written back to the buffer in the memory controller. Pcommit is similar except it becomes globally visible after all dirty blocks are persisted to NVMM.

To implement speculative persistence, we implement a FIFO buffer as a speculative store buffer in between the pipeline and cache. Under the speculative execution mode, all stores go to the speculative buffer and loads access this buffer before accessing the cache. Because the memory fence is retired when all previous memory operations are globally visible, the precise moment when clwb and pcommit become globally visible is important. In our implementation, once clwb is issued to the memory controller and the acknowledgement from the memory controller is replied back to CPU, it becomes globally visible. Likewise, once pcommit is completed and its acknowledgement is replied back to CPU, it becomes globally visible. Even though a speculative region may have completed, retired stores from the pipeline are accumulated in the SSB until all stores in the buffer are committed to the cache to keep the store order.

Table 3: SSB configurations and parameters.

Num entries	32	64	128	256	512	1024
Latency (cycles)	2	3	4	5	7	10

Once speculative persistence is triggered, loads must access the SSB and check for any store-to-load dependences before the L1D is accessed. Since SSB consists of CAM, RAM, and peripherals, its access latency cannot be ignored. In our experiments, we assume that CAM and RAM is accessed sequentially, but other peripherals are ignored for simplicity. Table 3 shows the access latency for each SSB size. The access latency grows significantly as the size increases.

5.2 Benchmarks

Currently, no full applications ported to PMEM are available to us. Thus, we used multiple types of trees in addition to lists, as these are common data structures in various applications. Our benchmark methodology is borrowed from recent studies [11, 22, 30, 37, 49]. The benchmarks that form the workload are shown in Table 1. The table shows the number of initial operations that are executed first to populate the data structure, which are executed in fast-forward mode in the simulator. For LinkedList, since the search time increases proportionally with the number of nodes, we limit its maximum nodes to 1024, so that the search time does not dominate. For simplicity, we assume that a deleted node is not immediately garbage collected, so that it can be reclaimed if a transaction fails.

For self-balancing tree benchmarks (AVL tree, B tree, and RB tree), we use full logging, which uses four pcommits per operation. We always assume the worst and log all nodes which may be involved if rebalancing is triggered.

6 EVALUATION

6.1 Overall Performance

In order to quantify the various sources of overheads, Figure 8 shows execution time overheads for successive additions to each benchmark: adding undo logging code (Log), adding PMEM persistence instructions including clwb, clflushopt, and pcommit (Log+P), and adding sfence (Log+P+Sf). While only Log+P+Sf is a correct and fail safe version, Log and Log+P help us debug the performance-robbing factors. The final bars show our SP with 256-entry speculative store buffer (SP256). The baseline is the original benchmarks without any logging or persistence. The final set of bars show the geometric mean of overheads, calculated by geometrically averaging the slowdown ratios and subtracting one from it.

The figure shows that adding logging code already causes significant performance overheads, 25% on average, but higher in tree benchmarks (28% for AVL tree (AT), 95% for B tree (BT), and 83% for RB tree (RT)), due to many more nodes being logged. Non-tree benchmarks only suffer less than 5% overheads from the logging code. Adding PMEM instructions increase the overheads only slightly, to 33% on average, so the PMEM instructions by themselves do not contribute much overhead. However, when the store fences are added, the average overhead shoots up to 60%. Interestingly, non-tree benchmarks, which only log few nodes per operation, such as graph (GH), HashMap (HM), and String Swap (SS), suffer

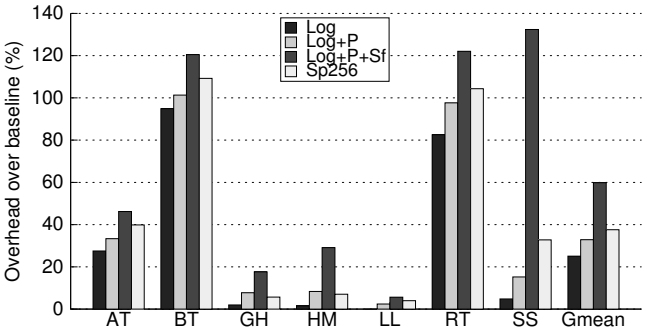

Figure 8: The successive additional overheads from logging (Log), PMEM instructions (Log + P), sfence instructions (Log+P+Sf), versus our scheme (SP256), normalized to code without logging and persistency.

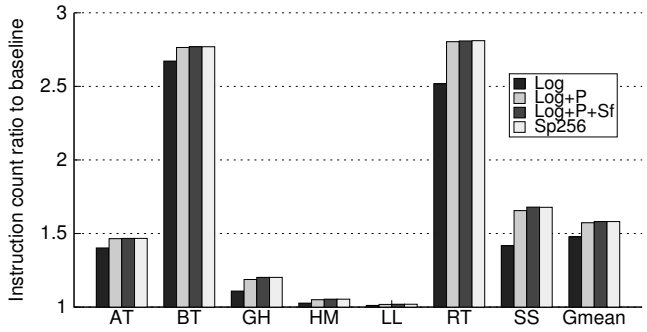

Figure 9: The ratio of instruction count to baseline instruction count.

very significant slowdowns from the addition of sfences. This points to the sfences as the primary bottleneck for these benchmarks. Also, considering that logging is necessary for ensuring failure safety, sfences are also the primary bottleneck for tree banchmarks.

Finally, our scheme, SP, successfully brings down the overheads to 38%, which is only slightly higher than Log+P, indicating that most of the overhead due to the pipeline stalls introduced by sfences have been removed.

Figure 9 shows the number of committed instructions in each benchmark divided by the original number of committed instructions. It shows that the logging code is the primary contributor in instruction count increase. PMEM instructions only add slightly to the committed instruction count, and the sfence count is negligible. This confirms that the slowdown from sfences cannot be due to the increase in the instruction count. Figure 10 shows the number of fetch queue stall cycles divided by the original number of execution cycles. It further corroborates that the overhead of sfences come from pipeline stalls: the fetch queue stall cycles of Log+P+Sf are much higher than those of Log+P. The figure also shows that these additional fetch queue stall cycles are nearly eliminated with our scheme SP256, bringing them only slightly above Log+P.

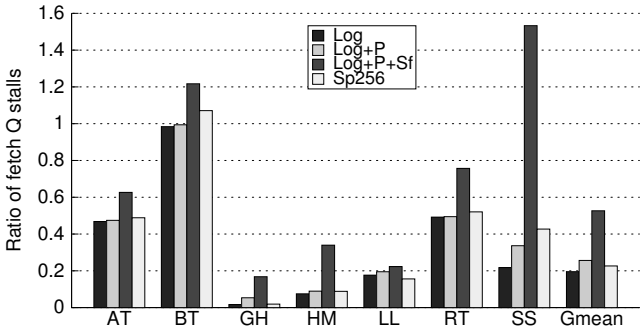

Figure 10: The ratio of fetch queue stall cycles to baseline exeuction cycles.

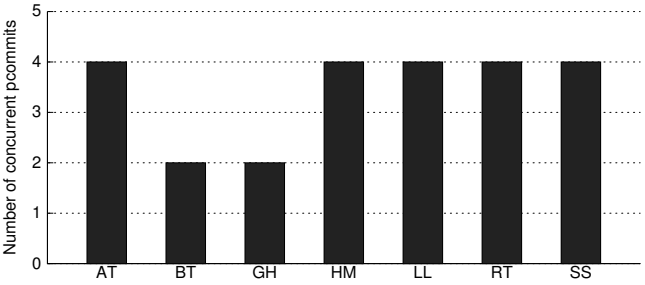

Figure 11: The maximum number of in-flight pcommits.

6.2 Speculative Store Buffer Size

Let us now consider the checkpoint buffer and speculative store buffer design. In designing the checkpoint buffer, we need to figure out how many checkpoints we need to be able to create and keep. To measure that, Figure 11 shows the number of maximum concurrent pcommits, which represents the maximum number of pcommits that are encountered while there is at least one older pcommit that has not completed. This data is collected using the Log+P version without the presence of sfences. The figure shows that the maximum number of concurrent pcommits for most benchmarks is four, meaning that a checkpoint buffer with four entries is sufficient.

Next, we use Log+P versions of the benchmarks to count how many store instructions are executed in the pipeline while a pcommit is outstanding. Figure 12 shows the total number of such stores divided by the total number of pcommits. This store count includes clflush and clwb. The figure shows that the number of store instructions is less than 20, except for SS. Taken together, we can infer that the SSB needs to have as many entries as the number of concurrent pcommits multiplied by the number of stores that are executed for each outstanding pcommit. From the two previous figures, we can infer $4 \times 20 = 80$ entries. Note, however, in the Log+P version the pipeline executes faster, so 80 entries for the SSB is likely a floor for a good design.

Figure 13 shows the execution time overheads of SP with various numbers of entries. The figure shows that, on average, 256 entries shows the best performance, but we also note 128 entries produces nearly as good performance as 256. The overheads increase as the SSB increases beyond 256 entries due to the high access latency. The

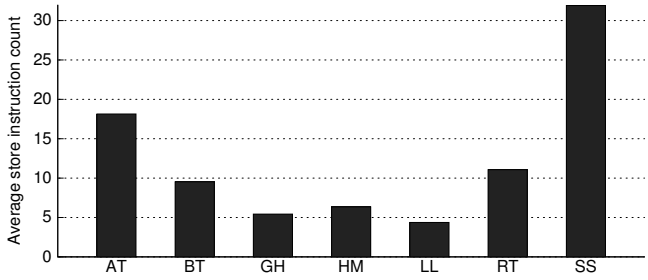

Figure 12: The average number of speculative stores while a pcommit is outstanding.

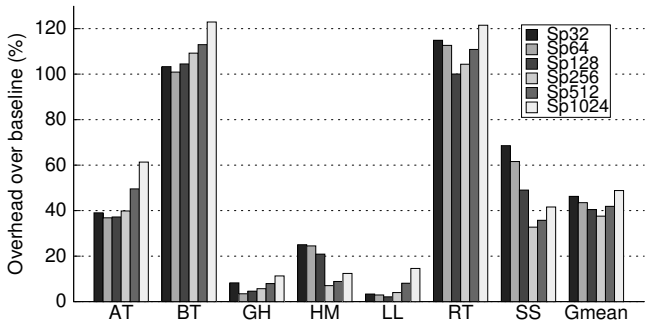

Figure 13: The overhead of different configurations for speculative store buffer size over baseline.

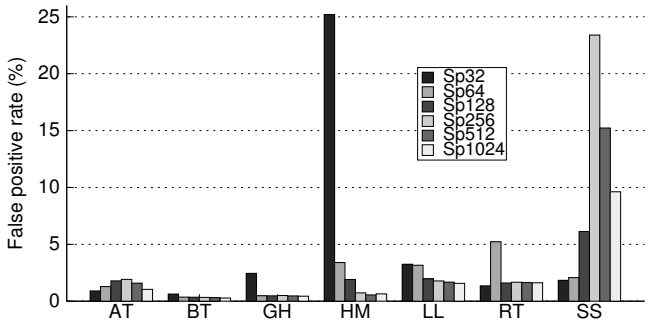

Figure 14: The rates of bloom filter's false positive.

overheads increase as the SSB decreases from 128 due to structural hazards of not being able to continue speculative execution.

6.3 Bloom filter

Due to the overhead of SSB access latency, we use a bloom filter to let a load quickly check for a match of address with stores in the SSB. The bloom filter can produce false positives but not false negatives. Figure 14 shows the false positive rates of a 512-byte filter. The figure shows relatively low false positive rates except for SS. We also found that the false positive rates are not caused by the size of the bloom filters, so enlarging it does not help much. Instead, the false positives occur when stores have completed and left the SSB while the bloom filter has not been reset yet.

7 RELATED WORK

Speculative execution. Speculative execution is a useful and versatile tool in computer architecture design and has been widely studied in a variety of contexts. Researchers have applied speculative execution to sequential consistency to compensate for its performance loss compared to relaxed consistency models [7, 9, 14–18, 35, 40, 47]. This form of speculation allows aggressive reordering of loads and stores for higher performance. Also, speculation has been proposed for other purposes such as hiding L2 miss latency [8, 24, 34, 42], and speculating past synchronization operations [19, 31, 39]. It has also been used to ease the burden of parallelization by supporting speculative parallelization [10, 43, 44] and transactional memory [2, 18, 19]. Hardware Transactional Memory is implemented in Intel's current processors using their TSX extension. Support for multiple checkpoints has been described before in various contexts, like [8, 18].

We are the first to apply speculation to memory persistency and identify its primary architectural requirements based on our workloads. Specifically, PMEM instructions that occur in the shadow of a prior pcommit instruction must be delayed and executed at the end of a speculative epoch. This in turn requires supporting multiple checkpoints and speculative epochs to speculate past multiple persist barriers.

Speculative persistence. Prior research also used the term *speculative persistence* [30], but our work assumes a different recovery model. The prior work allows stores separated by a persist barrier to be persisted in any order but revealed to software in transaction order. This breaks recoverability of our benchmarks because the logging is performed in software in our case. It also requires a more sophisticated/complex hardware for logging, bookkeeping, and recovery compared to what we propose.

8 CONCLUSION

In this paper, we have discussed Intel's PMEM persistency instruction support for non-volatile main memory (NVMM). We rewrote several data structures and kernels commonly found in databases and file systems to incorporate failure safety. We showed how transactional semantics using write-ahead logging can be used as the basis for reasoning about failure safety in the context of the PMEM model. Then, we discovered performance bottlenecks resulting from the use of PMEM instructions, namely frequent pipeline stalls resulting from the *sfence-pcommit-sfence* instruction sequence.

Based on these observations, we proposed *speculative persistence* (SP), architectural support to speculatively execute past long latency persist barriers to hide their latency and reduce their impact on performance. Speculative execution is triggered when a persist barrier stalls the pipeline. Rather than waiting, in SP, a checkpoint is taken at the persist barrier, the sfence is speculatively retired, and the processor proceeds speculatively retiring the sfence and following instructions. Meanwhile, the pcommit completes non-speculatively in the background. We discussed unique challenges that arise in designing SP and architecture support that addresses them. We evaluate our new architecture and compared it against the same system without speculation. Our experiments show that SP achieves execution time overheads of only 3.6% on average over code with logging but without fence instructions to order persists. This is a significant reduction compared to the 20.3% average overheads without speculation.

ACKNOWLEDGEMENT

Shin was supported in part by NCSU. Solihin's work was supported by (while he was serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. In *IEEE, Vol. 98, Issue: 12*. 2237 – 2251. https://doi.org/10.1109/JPROC.2010.2070830

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*. IEEE Computer Society, Washington, DC, USA, 316–327. https://doi.org/10.1109/HPCA.2005.41

[3] ARM. 2016. ARMv8-A architecture evolution. (January 2016). https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[4] NVM Library Team at Intel. 2016. Persistent Memory Programming. (August 2016). http://pmem.io.

[5] Amro Awad, Sergey Blagodurov, and Yan Solihin. 2016. Write-Aware Management of NVM-based Memory Extensions. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2925426.2926284

[6] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 263–276. https://doi.org/10.1145/2872362.2872377

[7] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1555754.1555785

[8] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. 2006. CAVA: Using Checkpoint-assisted Value Prediction to Hide L2 Misses. *ACM Trans. Archit. Code Optim.* 3, 2 (June 2006), 182–208. https://doi.org/10.1145/1138035.1138038

[9] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/1250662.1250697

[10] Marcelo Cintra, José F. Martínez, and Josep Torrellas. 2000. Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/339647.363382

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[13] Intel Corp. 2016. *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. Intel.

[14] Amit Gandhi, Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, and Konrad Lai. 2005. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 446–457. https://doi.org/10.1109/ISCA.2005.46

[15] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *In Proceedings of the 1991 International Conference on Parallel Processing*. 355–364.

[16] Chris Gniady and Babak Falsafi. 2002. Speculative Sequential Consistency with Little Custom Storage. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 179–188. http://dl.acm.org/citation.cfm?id=

645989.674317

[17] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. 1999. Is SC + ILP = RC?. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*. IEEE Computer Society, Washington, DC, USA, 162–171. https://doi.org/10.1145/300979.300993

[18] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. 2004. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/1024393.1024395

[19] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[20] Intel. 2016. Deprecate PCommit Instruction. (September 2016). https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[21] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology. (Jul. 2015). https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

[22] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

[23] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2007. 2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read. In *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. https://doi.org/10.1109/ISSCC.2007.373503

[24] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. 2005. Checkpointed Early Load Retirement. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*.

[25] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[26] Mark H. Kryder and Chang Soo Kim. 2009. After Hard Drives – What Comes Next? *IEEE Transactions on Magnetics, Vol. 45, Issue: 10*, 3406–3413. https://doi.org/10.1109/TMAG.2009.2024163

[27] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-effcient main memory alternative. In *IEEE International Symposium on Performance Analysis of Systems and Software*. https://doi.org/10.1109/ISPASS.2013.6557176

[28] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[29] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010), 143–143. https://doi.org/10.1109/MM.2010.24

[30] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *Computer Design, 2014 32nd IEEE International Conference on (ICCD'14)*. https://doi.org/DOI:10.1109/ICCD.2014.6974684

[31] José F. Martínez and Josep Torrellas. 2002. Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 18–29. https://doi.org/10.1145/605397.605400

[32] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770

[33] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*. ACM, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/2524211.2524216

[34] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 129–. http://dl.acm.org/citation.cfm?id=822080.822823

[35] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. 1996. An Evaluation of Memory Consistency Models for Shared-memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/237090.237142

[36] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference*.

[37] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. https://doi.org/10.1145/2678373.2665712

[38] Raghunath Rajachandrasekar, Sreeram Potluri, Akshay Venkatesh, Khaled Hamidouche, Md. Wasi-ur Rahman, and Dhabaleswar K. (DK) Panda. 2014. MIC-Check: A Distributed Check Pointing Framework for the Intel Many Integrated Cores Architecture. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 121–124. https://doi.org/10.1145/2600212.2600713

[39] Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. IEEE Computer Society, Washington, DC, USA, 294–305. http://dl.acm.org/citation.cfm?id=563998.564036

[40] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. 1997. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/258492.258512

[41] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[42] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. 2004. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 107–119. https://doi.org/10.1145/1024393.1024407

[43] J. Steffan and T Mowry. 1998. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA '98)*. IEEE Computer Society, Washington, DC, USA, 2–. http://dl.acm.org/citation.cfm?id=822079.822712

[44] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A Scalable Approach to Thread-level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/339647.339650

[45] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[46] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, and Mingdi Xue. 2015. How to Be Consistent with Persistent Memory? An Evaluation Approach. In *IEEE International Conference on Networking, Architecture and Storage (NAS'15)*. https://doi.org/DOI:10.1109/NAS.2015.7255223

[47] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for Store-wait-free Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 266–277. https://doi.org/10.1145/1250662.1250696

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 167–181. http://dl.acm.org/citation.cfm?id=2750482.2750495

[49] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744