Proteus: A Flexible and Fast Software Supported Hardware Logging approach for NVM

Seunghee Shin   Satish Kumar Tirukkovalluri   James Tuck   Yan Solihin

Dept. of Electrical and Computer Engineering
North Carolina State University
{sshin6,stirukk,jtuck,solihin}@ncsu.edu

ABSTRACT

Emerging non-volatile memory (NVM) technologies, such as phase-change memory, spin-transfer torque magnetic memory, memristor, and 3D Xpoint, are encouraging the development of new architectures that support the challenges of persistent programming. An important remaining challenge is dealing with the high logging overheads introduced by durable transactions.

In this paper, we propose a new logging approach, *Proteus* for durable transactions that achieves the favorable characteristics of both prior software and hardware approaches. Like software, it has no hardware constraint limiting the number of transactions or logs available to it, and like hardware, it has very low overhead. Our approach introduces two new instructions: log-load creates a log entry by loading the original data, and log-flush writes the log entry into the log. We add hardware support, primarily within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data. We also propose a novel optimization at the memory controller that is enabled by a persistent write pending queue in the memory controller. We drop log updates that have not yet written back to NVMM by the time a transaction is considered durable.

We implemented our design on a cycle accurate simulator, MarssX86, and compared it against state-of-the-art hardware logging, ATOM [19], and a software only approach. Our experiments show that *Proteus* improves performance by 1.44-1.47× depending on configuration, on average, compared to a system without hardware logging and 9-11% faster than ATOM. A significant advantage of our approach is dropping writes to the log when they are not needed. On average, ATOM makes 3.4× more writes to memory than our design.

CCS CONCEPTS

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

KEYWORDS

Non-Volatile Main Memory, Software Supported Hardware logging, Failure Safety

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies, such as phase-change memory, spin-transfer torque magnetic memory, memristor, and 3D Xpoint, are expected to be in the market soon [1, 16, 20, 22, 27, 40]. For example, Intel and Micron announced that their 3D Xpoint memory will be in the market in 2017 [16]. Due to their non-volatility and byte addressability, a subset of them which have low read latencies are being considered for use as main memory, either to augment or replace DRAM [11]. A *non-volatile main memory* (NVMM) can be directly accessed using load and store instructions. This gives an opportunity for programmers to persist important data in data structures in main memory, skipping the need or overheads of serializing it to the file system.

Persisting data structures in main memory must be implemented in a way that ensures data consistency, so that software can recover to a consistent state in the event of software or hardware failures. Data consistency is difficult to ensure in systems with volatile caches because the order in which values are *persisted* (e.g. written back to the NVMM) depends on the cache replacement policy, which often differs from program order. To deal with these ordering challenges, architects have proposed memory persistency models [2, 9, 18, 37] to give programmers a guarantee of the ordering in which stores are persisted to NVMM.

One model that is easier for programmers to reason about is that of *durable transactions* [2, 23, 43]. With a durable transaction, all stores in a transaction persist or none of them do.[1] This is a simple and useful abstraction for programmers.

In this paper, we explore a key challenge of using durable transactions: how to perform logging efficiently and flexibly. Durable transactions require logging, either through *redo* or *undo* logging. The log allows the transaction to be recovered if a failure occurs during the transaction. Each log entry can be created through software code inserted by the programmer, through a library [2], or directly in hardware without additional code [19]. Software approaches (SW) incur large performance overheads due to additional instructions but offer the greatest flexibility, including unlimited transaction size and

---

[1]Note a fundamental difference between durable transaction and transactional memory (TM): a durable transaction specifies when data is made durable in NVMM, whereas TM deals with when data is visible to other threads. Consequently, durable transactions apply even to sequential code.

FLEXIBILITY

		LOW	HIGH
PERFORMANCE	**HIGH**	Atom	Proteus (NEW)
	LOW		Mnemosyne Nv-Heap PMEM

Figure 1: Logging model taxonomy.

control over logging operations. The latter approach (HW) [19] has low performance overheads, but it is typically less flexible.

In this paper, we propose a new logging approach, referred to as *Proteus*, that achieves the favorable characteristics of both software and hardware approaches: software controls it's own log area, it can support unlimited transactions of arbitrary size, it can manage its own recovery, and it has low overhead. Our approach introduces two new instructions that a log-load instruction creates a log entry by loading the original data, and a log-flush instruction writes the log entry to NVMM. We presume no additional programming effort beyond specifying transaction boundaries, since the compiler can generate instructions appropriately for code inside transactions. Additional hardware support is introduced, largely within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data to ensure durable transaction semantics. *Proteus* avoids any limitation on the size or number of transactions through judicious design of the interface: software remains in control of allocating the log area, and hardware keeps the cost of updating the log low. The taxonomy in Table 1 illustrates each logging model's pros and cons.

We also consider integrating *Proteus* with a battery backed WPQ, allowing the WPQ to be considered part of the persistency domain. Once writes reach the WPQ they are considered durable. The presence of a battery-backed WPQ is consequential: it presents a new opportunity to avoid writes to the NVMM. A key observation that we exploit is that most log entries are created and discarded, because failures are rare. Thus, we apply an optimization where we distinguish whether data blocks in the WPQ are there for logging or not. This distinction allows us to treat them differently, where log entries are kept as long as possible in the WPQ and discarded when a transaction commits, whereas non-log write-backs are allowed to drain from the WPQ to the NVMM. Although this optimization offers an insignificant improvement in performance, more importantly it helps in extending the lifespan of NVMM by avoiding many unnecessary writes to the NVMM [3–5, 12, 28, 39, 47, 49].

We implemented *Proteus* on a cycle accurate simulator, MarssX86, and compared it against state-of-the-art hardware logging (ATOM [19]) and a software only approach. Our experiments show that *Proteus* improves performance by 1.44-1.47× depending on configuration, on average, compared to a system without hardware logging and 9-11% faster than ATOM. A significant advantage of our approach

is dropping writes to the log when they are not needed. On average, ATOM makes 3.4× more writes to memory. Even though stores are often not on the critical path, persistent writes are critical given the store-ordering constraints required for durable transactions.

The remainder of the paper is organized as follows. Section 2 provides background on memory persistency, new PMEM instructions, and previous logging implementations for transactions. Section 3 introduces our software supported hardware logging approach. Section 4 describes the design of *Proteus* in detail, Section 5 describes the evaluation methodology, and Section 6 evaluates our design and presents our key findings. Section 7 presents a sensitivity study of alternative architectural configurations and their impacts on performance. Section 8 discusses related work. Section 9 concludes.

2 BACKGROUND

2.1 Memory Persistency Models

A *persistency model* is a specification of the allowable orderings in which stores persist (i.e. are made durable in the NVMM) with respect to the order in which stores appear in the program order. Persistency models give programmers a means to reason about the order of persists, when persists become durable, and failure-safety. Previous research proposed several memory persistency models, starting from a very high level of abstraction based on durable transactions [23, 43], where all stores in a transaction persist entirely or none of them do. In this model, stores within a transaction are not ordered. Only the ordering of stores in a transaction with stores outside the transaction are enforced.

At a lower level of abstraction, *strict persistency*, *epoch persistency*, *buffered epoch persistency*, and *strand persistency*, have been proposed [24, 37]. They give programmers a guarantee of the ordering in which stores are persisted to the NVMM, regardless of the presence of durable transactions. Strict persistency [37] piggy-backs on the sequential consistency model by specifying that a store that is globally visible must also have persisted. Due to this constraint, before each store persists to NVMM, all previous stores must have persisted to NVMM. While it is easier to reason about failure safety under this model, it comes with significant performance costs of not allowing write reordering and write coalescing that naturally occur in write back caches. Epoch persistency relaxes the ordering constraint in strict persistency [9, 37]. It allows the programmer to put a persist barrier which defines an epoch in the program. Stores from an epoch (i.e. between two persist barriers) can persist in any order, but they must all persist before the 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. The buffered epoch persistency model [9, 18] relaxes the persist barrier by not forcing 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. 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.

Even lower in the abstraction level is a set of primitives that can be used to specify the ordering of store persistence. Intel PMEM [2] is an example of this approach. PMEM was designed to be compatible with x86 systems, and includes a few new instructions, such

as *clwb*, *clflushopt*, and *pcommit*. *clwb* and *clflushopt* flush a dirty block from caches to a *write pending queue* (WPQ) in the memory controller (MC), while *pcommit* flushes the dirty block from WPQ to the NVMM. By appending either clflushopt or clwb and pcommit after stores that need to persist, programmers can selectively choose which stores should persist and when they persist. 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. To force an ordering among two of these instructions, a store fence instruction (sfence) is required. sfence, which was originally introduced as a memory barrier to control the visibility of a store with respect to other threads, is extended in PMEM so that it waits for all pending PMEM instructions to complete before retiring. This prevents following stores and PMEM instructions from executing until the sfence completes.

Another important concept, related to the persistency model, is the *pesistency domain*. The persistency domain is an architectural description of which components in the system are persistent. Once a store reaches the persistency domain, it can be considered durable. Often, only the NVM itself is in the persistency domain. However, moving the persistency domain on-chip and closer to a core can significantly reduce the time to complete a persist operation. Intel has proposed such an optimization called Asynchronous DRAM Refresh (ADR). While the name refers to DRAM, a key feature of this specification is that data in the WPQ in the memory controller can be considered part of the persistency domain. This change makes *pcommit* unnecessary since pending operations in the WPQ are already in the persistency domain and do not need to be forced to NVM. Consequently, Intel has deprecated the *pcommit* instruction.

Due to its lower abstraction level, PMEM instructions can be used to implement some of the other persistency models. For example, if a sequence of (clflushopt, clwb) is inserted after each store, strict persistency is achieved. An example is illustrated in the following code. The first column shows strict persistency where st X, st Y, and st Z are strictly ordered. The second column shows epoch persistency where st X and st Y fall within one epoch, but st Z is in the next epoch.

Strict Persistency	Epoch Persistency
i1: st X, 1;	i8: st X, 1;
i2: clwb X;	i9: st Y, 1;
i3: sfence;	i10: clwb X;
i4: st Y, 1;	i11: clwb Y;
i5: clwb Y;	i12: sfence;
i6: sfence;	i13: st Z, 1;
i7: st Z, 1;	

In this paper, we assume durable transactions as our persistency model, where all stores in a transaction either persist together or not at all [23]. Similar to ADR, we include the memory controller in the persistency domain, which allows our scheme a significant opportunity to reduce the number of writes to NVMM.

2.2 Failure Safety

Applications manipulating important data such as database or file-system must ensure that the data is consistent under power failure (*failure safety*). To construct a durable transaction, where a group of stores are made durable atomically, two approaches are popular:

copy-on-write (COW) and *write-ahead logging* (WAL), With COW, a write triggers data to be copied to a new location where the write will occur. The original data is left intact. COW requires address remapping so that future reads can be redirected to the new location. This remapping is generally considered expensive. For block-based storage, the cost can be amortized by applying copying to a large granularity such as a page, and committing updates infrequently [9]. However, NVMM access is byte-based and stores occur much more frequently than file writes. Thus, COW may be prohibitively expensive to use in NVMM.

With WAL, a redo/undo log for all desired changes is persisted in NVMM prior to committing any modifications. If a failure occurs in the middle of transaction processing, the saved log is used to redo/undo the transaction [42]. Although each update requires two writes, it is not as expensive as an update in COW. WAL can be performed frequently in a small granularity such as the cache block size. The following steps show an example of how undo logging in software can be constructed to ensure failure safety:

Step 1	Perform undo-logging and persist undo log.
Step 2	Set *logFlag* and persist it, indicating transaction start.
Step 3	Update data structure and persist it.
Step 4	Unset *logFlag* and persist it, indicating transaction completion.

Figure 2: Steps needed to implement fail-safe undo logging in software.

logFlag is used to indicate the progress status of the transaction. If failure occurs before Step 2 completes, the transaction is retried. At re-execution, the log will be overwritten. If failure occurs after Step 2 completes but before Step 4 completes, the log is used to undo the transaction, prior to transaction re-execution. In this implementation, each step must persist before the next step is executed. If PMEM is used, at the end of each step, modifications in the cache must be flushed using *clwb* and *sfence*. These instructions are inserted to prevent reordering with the next step. In this paper, we adopt this implementation as our baseline.

3 HARDWARE LOGGING

In the previous section, we have discussed how logging is necessary to support durable transactions that are critical in achieving failure-safe applications. In this section, we introduce our new design, *Proteus*, which performs logging in hardware in a flexible way.

3.1 Software Supported Hardware Logging

As discussed earlier, with durable transaction support, programmers can achieve failure safety by grouping related writes into an atomic section. The atomic section requires WAL to ensure a log is created prior to making modifications to data in the NVMM. Log creation, maintenance, and truncation, can be performed in software (*software logging* or SL) or in hardware (*hardware logging* or HL). SL, traditionally a popular technique for persistent (block-based) storage devices, has recently been adapted for use in NVMM, e.g. Mnemosyne [43] and NV-Heap [8]. SL is flexible: it is compatible with a wide range of systems, it does not impose any restrictions on
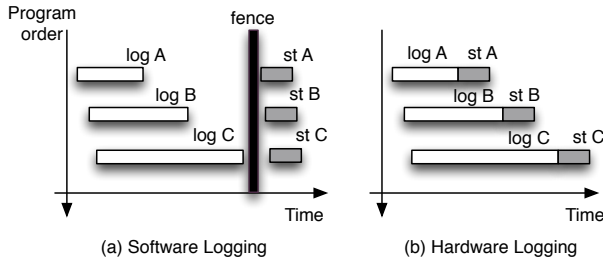
(a) Software Logging   (b) Hardware Logging

Figure 3: Comparison between software logging and hardware logging.

the transaction size or count, and can be changed without impacting the underlying architecture. However, the flexibility also comes with significant costs. First, there are additional instructions that must be executed for creating and maintaining logs, including stores and cache line write-back instructions. Second, a log must persist prior to data modifications, hence the log creation directly lengthens the critical path of transaction execution. Third, when implemented with PMEM, memory fences are needed to impose ordering between the steps shown in Figure 2. In our experiments, across a variety of benchmarks, on average SL makes the execution time 1.51x longer.

In response to the execution time and instruction overheads of SL, hardware logging (HL) has recently been proposed, e.g. ATOM [19]. With HL, software indicates the transaction start and end, while a hardware transaction manager creates and manages the log automatically. HL has several advantages over SL. First, no logging memory instructions are needed. Second, since the processor can distinguish logging memory updates from data updates, their ordering can be ensured in hardware at a finer granularity without reliance on memory fences. The effect of this on execution time is illustrated in Figure 3. Finally, the awareness of logging updates allows new optimizations. For example, in ATOM, the posted log optimization allows stores to complete before logging updates are made durable by locking the cache block in the MC, thereby preventing subsequent data updates from being persisted until the logging update becomes durable. Source log optimization relegates the creation of log entries to the memory controller instead of the cache controller. This allows log entries to be created with a lower latency. Both of these optimizations are important for reducing the overhead of logging.

In order to get the high performance of HL without losing the flexibility of SL, we propose a third approach: *software supported hardware logging* (SSHL). The key idea behind SSHL is to still rely on software to perform log creation and maintenance, but allow software to inform hardware which operations are performing logging. This removes the need for hard-wiring log creation/maintenance into hardware, thereby preserving SL flexibility. On the other hand, since hardware can now distinguish which operations are logging operations as opposed to regular loads/stores, it can optimize specifically for logging operations differently from regular loads/stores.

We exploit the combined effect of knowing which operations are logging and the fact that the memory controller is in the persistency domain to drop logging operations from the MC once they are no longer needed. When a transaction ends, we can be sure that all of

its data updates are durable, either in the NVMM or in the WPQ at the memory controller. Hence, the logging operations that ensure failure safety are no longer needed. Since we can differentiate them from regular stores or necessary persists, we can simply discard them. This not only helps to save power, but also avoids write amplification due to logging updates that may wear out the NVMM.

3.2 New Logging Instructions

Proteus requires software to inform hardware of logging operations. Each logging operation requires two memory addresses: the *log-from* address to hold the address of the original data and the *log-to* address to hold the corresponding address in the log. In order to indicate logging updates to hardware, *Proteus* needs instruction support. We consider two approaches in designing the logging instructions. In the first approach, we consider using a single instruction to do the entire logging operation. However, this requires the log-from and log-to addresses to be specified in a single instruction. This leads to the complexity of possibly handling two page faults in one instruction. Thus, we consider an alternative approach where we rely on two instructions to specify a logging operation. The first instruction reads data from the log-from address. The second flushes it to the log-to address making the log entry durable. The two instructions have the following format:

| log-load $LR1 M1 | Load 32B block from log-from address M1 to log register LR1 |
| log-flush $LR1 M2 | flush data from LR1 to the log entry in NVMM at log-to address M2 |

Here, LR is a register that holds a full log entry consisting of the log data, the log-from address, and some other metadata (see Section 4). The LR registers are added to support the logging operation. The log-load reads a 32-byte block from address M1 (the log-from address) into the log register LR1 along with the log-from address. The log-flush writes the value in LR1 to the log-to address M2. The log-load completes when data is received at the log register. The log-flush completes when the flush is received at the WPQ in the memory controller and after the memory controller acknowledges receipt.

When a durable transaction is defined by programmers using *tx-begin* and *tx-end*, a compiler expands each store in the transaction into three instructions, *log-load*, *log-flush*, and store. Figure 4 shows an example of such transformation. On the left column, the program expresses a durable transaction region with two stores to addresses A and B. On the right column, the compiler replaces each store with a sequence of log-load, log-flush, and store. LR1 and LR2 are two 40-byte log registers. The number of log registers is selected such that it does not cause a structural hazard in the pipeline. The number of log registers determines the number of stores that can be logged concurrently. LTA is a special register that records the log-to address. In the figure, the auto-increment addressing mode is used to indicate that after a log-flush the LTA is incremented automatically. At the end of a transaction, tx-end instruction is inserted by a compiler in order to inform the end of the transaction to the processor. The actions taken by the architecture when executing tx-end are discussed in Section 4.

In the figure, the write to A is represented by instructions i2, i3, and i4, while the write to B is represented by instructions i5, i6, and

Durable transaction	After code generation
| ```
tx-begin
  A = ...
  B = ...
tx-end
``` | ```
i1: tx-begin
i2: log-load LR1, A
i3: log-flush LR1, (LTA)+
i4: st A
i5: log-load LR2, B
i6: log-flush LR2, (LTA)+
i7: st B
i8: tx-end
``` |

Figure 4: Example code transformation for our SSHL approach.

i7. Instructions i2 (log-load) and i3 (log-flush) show a read-after-write (RAW) register dependence on LR1. However, instruction i3 (log-flush) and i4 (st) do not show register or memory dependences between them. This allows i4 to be committed from the pipeline and placed in the store buffer. However, an additional ordering must be added here, where i4 cannot be released to the cache until the prior log-flush completes. Enforcing the ordering requires an additional hardware structure, which will be discussed later.

4 DESIGN DETAILS

4.1 Log area allocation and log granularity

Prior to using the log, an application needs to create and initialize the log area. One design question is how the log area is accessed. To answer it, we must consider how recovery after a failure will be supported, and in particular, whether the log is accessible through virtual memory or not. Choosing a virtual address (VA) for the log area implies that the application may be responsible for restoring its state. On the other hand if a physical address (PA) is used, the operating system (OS) must be responsible for restoring the state of all applications. Using PA for logging requires that both data and log pages are fixed and never swapped out. On the other hand, using VA for logging allows data or log pages to be swapped in/out, but this implies that the page table mapping can be restored across crashes, possibly by making the page table persistent or by leveraging relocatable objects [2].

Implementation complexity is important to consider. PA logging incurs significant hardware complexity because the logging operation breaks the virtual address space abstraction of programs, hence the OS and hardware need to be modified accordingly. For example, address space protection and isolation now require separate OS/hardware mechanisms. A fault cannot be handled by a conventional page fault handler. The HL approach is bound to use PA logging in order to avoid the memory controller from having to interact with an application's page table. On the other hand, SL and SSHL can use VA logging, because an application is responsible for the log area allocation and management. Consequently, our *Proteus* uses VA logging. Specifically, each thread in the application can allocate one log area. The log area is treated as a circular buffer, so that the log can wrap around when space runs out. We assume that the programmer chooses a log area size that is sufficiently large to accommodate the durable transaction size. If the logs overflow the assigned size in a transaction, the processor raises an exception.

Figure 5 shows a high level diagram of our *Proteus* architecture. At the top left, it shows four new registers. **log-start** and **log-end** record the start and end address of the log area, respectively. The **cur-log** register tracks the current free log entry. Finally, **txID** records the current transaction ID being executed in the core.

The Logging Data Register (LDR) file contains a number of log entries. Each log entry contains a logging data value and metadata, which contains the log-from address for the log entry and the transaction ID. The logging data value has a size that corresponds to the logging granularity, i.e. the number of consecutive bytes that are logged together with a single logging operation. The choice of logging granularity affects performance: if too small, there is little opportunity for coalescing that leads to a high number of log-flushes and writes to NVMM. If too large, data and metadata may not fit in a single cache line, requiring two writes to perform one log-flush and more writes overall to NVMM. Thus, we choose the logging data size to be 32B, leaving the reminder for metadata. Both data and metadata fit into one cache line (64B).

4.2 Proteus Architecture

Proteus adds hardware structures shown in grey in Figure 5. Several 40-byte log registers (LR) are added to the register file (LDR), to keep the log data and log-from address while logging instructions are executing in the processor pipeline. An LR is allocated when a log-load instruction enters the out-of-order (OOO) pipeline, and deallocated when it is no longer needed for detecting register dependences, i.e. when dependent log-flush instructions have committed. Because LRs can be recycled quickly, we found that eight LRs are sufficient.

LogQ is a structure that keeps track of each logging operation. When a log-flush instruction enters the OOO pipeline, an entry is created in the LogQ. It contains the log-from address (location of the original data in NVMM), log-to address (location of the log entry in the log area), and log-data (data value to be flushed to NVMM). The number of entries in the LogQ determines the maximum number of concurrent log-flush operations. A log-flush operation avoids write-allocate in the cache by directly passing the request to the memory controller (MC), avoiding cache pollution. To avoid a cache coherence issue, the log area is marked uncacheable. When the log-flush is received at the MC, the MC sends an acknowledgment to the LogQ and the entry is deallocated from the LogQ. Since an entry is deallocated relatively quickly, we found that increasing LogQ size has diminishing returns and 16 entries in our evaluation is a fair size.

The LogQ has another important function: imposing ordering between a log-flush instruction and a store to the same log-from address. Recall from our discussion in Section 3.2 that for correct failure recovery, the log entry must persist prior to the store persist. Therefore, a store to the same log-from address must remain in the StoreQ and not be released to the cache until the preceding log-flush operation is complete. Likewise, log-flushes have to check preceding stores in the storeQ before they are released to the cache. Thus, when a store retires and before it's committed to the cache, it checks its address against older entries in the LogQ. One corner case, worth mentioning, is that a full LogQ could prevent a log-flush from creating an entry. In this case, to ensure that no stores bypass the log-flush, we stall dispatch if a log-flush fails to find a free LogQ
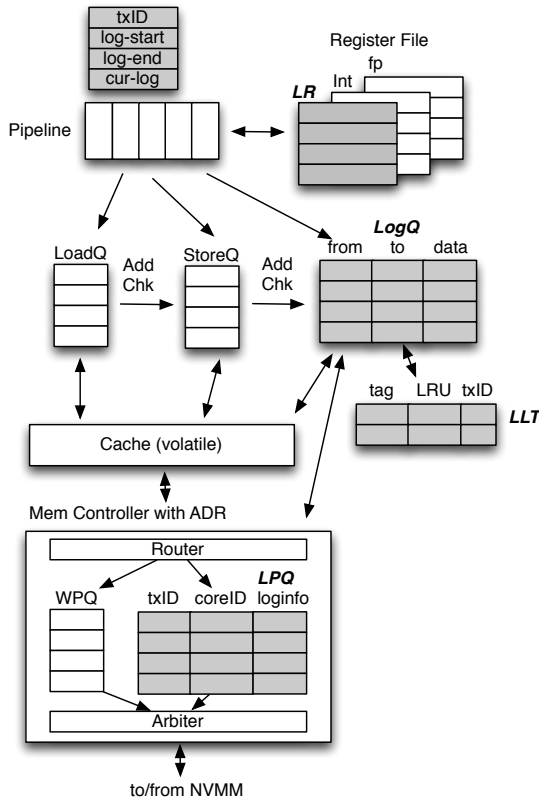
Figure 5: Proteus hardware design.

entry. This will ensure we can enforce the required persist ordering later between log-flush and a following store to the same address.

To achieve higher performance, the LogQ allows log entries to flush out-of-order. This must be done carefully to ensure correctness, otherwise it can jeopardize the correctness of recovery. For example, if two log entries in the same transaction have the same log-from address but different data, one must include updates from within the transaction and cannot be used for recovery. Only the first log entry in program order should be used to recover the state. One solution to prevent use of the wrong entry is to guarantee that the log-to address is assigned in program order for all log entries. In that way, recovery knows to use the earliest log entry and later ones are ignored. To guarantee this invariant, the log-flush determines its log-to address only after all previous log-flushes in program order have resolved their log-to addresses. In spite of the dependency among log-flushes to compute their log-to address, the LogQ can still hide the latency of logging by enabling the concurrent execution of the actual flushes to the MC. This turns out to be an important performance advantage over ATOM since it serializes log entry creation at store retirement.

Moving on, a structure called the Log Lookup Table (LLT) can be seen in the figure. Before explaining the structure, we observe that there is a significant log temporal locality within a transaction. That is, for our choice of log data size of 32 bytes, it is often the case that there are multiple stores to different bytes/words of the same 32-byte region. If we are not careful, each of these stores will create

a sequence of log-load and log-flush, leading to a high number of logging operations and writes to NVMM. As previously mentioned, any logging after the first one to a given log-from address is simply unnecessary overhead.

Eliminating unnecessary logging can be achieved through compiler analysis. However, the presence of aliased pointers make compiler analysis less effective. We prefer to solve it dynamically by adding the Log Lookup Table (LLT). The LLT keeps the last few log-from addresses in a transaction. If there is a new log-flush operation, its log-from address is checked against the LLT. On a match, the log-load and log-flush instructions complete immediately and are not given a log-to address. On an LLT miss, the log-load and log-flush proceed as usual, and the log-from address is added to the LLT replacing an LRU entry, if necessary, from the LLT. The LLT prevents repeated logging operations to the same log data, reducing the memory bandwidth devoted to logging. In addition, the LLT helps to reduce the size of the log area in NVMM, the LogQ, and the LPQ in the MC. For an LLT of 64 entries, the overhead is only 410 bytes.

When a transaction ends, triggered by the tx-end instruction, the LLT is cleared. This prevents the next transaction from finding stale data in the LLT and mistakenly believing it has already logged data. This is one of the primary purposes of tx-end, but there is also another purpose, as described in the next section.

4.3 NVMM Log Write Removal

Another hardware structure in *Proteus* is the Log Pending Queue (LPQ) in the MC. However, before going into that, we will first assume that the MC only has the Write Pending Queue (WPQ) and there is no LPQ.

With the introduction of ADR, the WPQ is now considered non-volatile. Thus, log entries can be considered durable when they are accepted at the WPQ (and before they are written to the NVMM), and acknowledgement of their completion can be sent to the processor at that time. This has the effect of allowing log-flush to complete sooner and stores released to the cache sooner.

We take an even greater advantage from ADR through an additional optimization. Although not in the critical path, writing log entries to the log area in NVMM is expensive due to the added power consumption and reduced write endurance of the NVMM. We note that log entries are no longer needed after a transaction ends (marked by the tx-end instruction) because all data updates are guaranteed to be durable, either in the WPQ or in the NVMM. In the common case, logs are written once and never read again. Hence, log entries that have not yet been written to NVMM after the transaction ends can be *flash cleared* and never written. This leads to the insight that we should keep log entries in the WPQ until a transaction ends, if possible, to avoid ever writing them to NVMM. This not only helps to save power consumption, but also avoids premature wear-out in the NVMM by significantly reducing the number of writes.

To achieve this, we must prioritize writing back regular writes from the WPQ to the NVMM. This requires a *priority bit* to be added (or expanded if it is already there) so that log flushes are de-prioritized and rarely released to the NVMM.

Inevitably, some log entries may have been released to the NVMM before the end of the transaction anyway. These log entries need to

be invalidated by reading and marking these log entries invalid in the NVMM. As a result, the potential savings are not as great as we might expect. To overcome this issue, ATOM introduces hardware in the MC to track all active log entries and clear them once a transaction is completed [19]. Because of this hardware design, ATOM's performance benefits are limited to its available resources. Once the resources run out, ATOM has to search the log area and invalidates them manually one by one. We address this issue in *Proteus* using a simple design. First, we add a transaction ID to the meta data of each logging operation so that all log entries belonging to a transaction can be identified quickly. Second, we allocate a separate log area for each thread. Intuitively, since there is only one active transaction at any particular time within a thread, only the log entries belonging to the most recent transaction are the valid logs.

In the above technique, the log area still needs to mark the end of a transaction (tx-end) to indicate whether the most recent log in the log area is valid or not. Instead of using one more log entry to mark the end of a transaction, *Proteus* utilizes the meta data of the last log entry for marking the end of the transaction. The last log entry in a transaction still needs to be flushed to NVMM. However, this mark is only necessary before the next transaction from the same thread starts in the log area. So we add a minor optimization that the last log entry is held in the WPQ (except when WPQ is full) and is discarded once a log entry from the next transaction reaches the WPQ.

So far we have described our flash clearing and priority bit optimizations. However, we notice that keeping log entries only in the WPQ has a limitation that logging operations and regular writes compete for entries in the WPQ. Increasing the number of WPQ entries can alleviate this, but it adversely increases read latency. An incoming read must be checked against WPQ entries for a match. A larger WPQ increases the checking time and directly increases the critical path of read requests.

To avoid extending the critical path of reads, we add the LPQ, and log flushes go only to the LPQ freeing the WPQ only for regular write-backs. An incoming read does not check against the LPQ because logs are not used again by the processor except during failure recovery. This also eliminates the additional priority bit in the WPQ. Prioritizing write requests in the WPQ over logging requests in the LPQ can be achieved at the arbiter instead. The LPQ contains log entries, where each entry contains the transaction ID, core ID, and various information about the log. When a transaction ends, all LPQ entries matching the particular transaction ID are cleared.

4.4 Context Switch

We also consider the case of supporting a context switch with our new hardware. The new registers added to the core need to be saved, namely the txID, log-start, log-end, cur-end, and LR registers. Also, we need to clear the LLT, just like on a tx-end, to ensure that entries in the LLT are not mistakenly used by another thread. Note, even if the same thread is rescheduled, it may simply result in additional log entries for the same data. However, this can be handled easily during recovery by recovering from the earlier entries in the log.

The other case we need to handle is ensuring that data in the MC is flushed to NVMM. Since we do not know how long the thread may be switched out, we send a message to the MC informing it to write all LPQ entries for the txID to NVMM. This conservatively assures correctness. Since context switches are rare, the performance penalty is minor.

For processors do not have an explicit context switch instruction that can perform these actions, we add the *log-save* instruction to carry out the actions we described.

5 METHODOLOGY

5.1 Simulation configuration

For our experiments, we implemented Intel PMEM instructions, *clwb*, *clflushopt*, and *pcommit* in a processor simulator built on MarssX86 [36], which is an open source cycle-accurate full system simulator for an x86-64 architecture. We also add the appropriate ordering constraints for memory fences (*sfence* and *mfence*) with respect to PMEM instructions. Of the PMEM instructions, we only use *clwb* because *pcommit* has been deprecated while *clflushopt* is not needed (it invalidates a block after flushing it). Our simulation model supports a detailed out-of-order multicore CPU, coherent caches, interconnection, and memory controller models. To simulate a detailed memory system, we also integrated DRAMsim2 into MarssX86. In our implementation, the ordering constraint of *clwb* and *pcommit* is implemented as described in Intel's manual [14], where *clwb* is ordered with respect to older stores to the same address and store-fencing operations (e.g. sfence and mfence). After *clwb* is retired from the CPU pipeline, it accesses the cache in the same way as regular stores. If the *clwb* hits a dirty block in the cache hierarchy, the dirty cache block is flushed to the WPQ in the MC. Once the cache block is placed in the WPQ, the *clwb* becomes globally visible.

In addition to implementing *Proteus*, we also implement the state-of-the-art in hardware logging, ATOM [19], for comparison. In ATOM, a log entry is automatically generated right before a store gets retired. Logging delays the store's retirement and the store is held in the storeQ until the logging operation is completed. In order to compare with the best-performing version of ATOM, we implemented and integrated both posted log optimization and source log optimization into ATOM. Thus, logging is considered complete once the log entry arrives at the MC which locks the cache line and sends an acknowledgment back to the cache controller. This optimization reduces the latency of logging, which is on the critical path of the store operation. Furthermore, on a cache miss with a logging operation, a log entry is created in the MC before the data is sent to the cache.

The simulation parameters and architecture configurations for the processor and memory systems are listed in Table 1. Our machine model includes a quad-core processor with each core supporting out-of-order issue and execution with three levels of cache backed by DRAM/NVMM. The configuration parameters are similar to Intel's Skylake architecture [15] with minor differences. The DRAMsim2 parameters are also listed in the table. To simulate NVMM latencies for the baseline memory which assumes DDR3-1600, we increased tRCD to 29 cycles for read and 109 cycles for write (50ns for read and 150ns for write), in line with numbers assumed in prior work [4, 25, 26, 31, 42, 44, 46].

Table 1: The baseline system configuration.

Processor	OOO, 3.4GHz, 4 cores, 5-wide issue/retire. ROB: 224, fetchQ/issueQ/LoadQ/StoreQ: 48/64/72/56
L1I and L1D	32KB, 8-way, 64B block, 4 cycles, private per core
L2	256KB, 8-way, 64B block, 12 cycles, private per core
L3	8MB, 16-way, 64B block, 42 cycles, shared by all cores
Interconnect Bandwidth	96B/cycle for CPU-L1, 64B/cycle for L1-L2 32B/cycle for L2-L3, 16B /cycle for L3-MC
DRAM	DDR3-1600 (800MHz), 8GB 1 channel, 16 Banks per rank, 2KB row-buffer
t_{CAS}-t_{RCD}-t_{RP}-t_{RAS}-t_{RC}-t_{WR}-t_{WTR}-t_{RTP}-t_{RRD}-t_{FAW}	
11-11-11-28-39-12-6-6-5-24	
NVM	tRCD 29 for Read, 109 for Write
Proteus	LR: 8 registers, LogQ: 16 entries, LLT: 64 entries (8way), LPQ: 256 entries

Table 2: Benchmarks constructed for our study. Except for SS which has 256 bytes for each string, we size each node to be 64 bytes and align them to cache blocks in all benchmarks. Thus, to persist one node update, one *clwb* will be required. InitOps and SimOps are expressed in terms of the number of operations that are executed per thread.

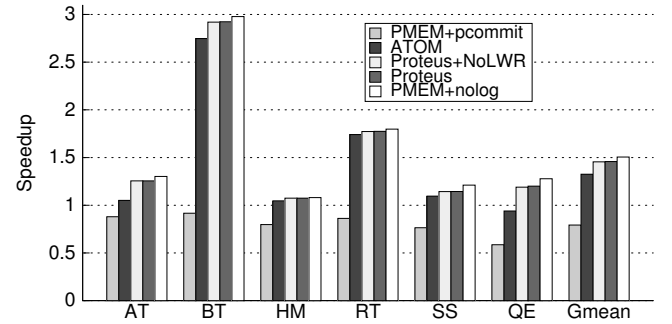| Benchmark (Abbrev.)	Description	#InitOps	#SimOps
Queue (QE)	Enqueue/dequeue in 8 queues	20000	50000
HashMap (HM)	Insert or delete entries in 16 hash maps	100000	20000
String Swap (SS)	Swap strings in a string array (262144 items)	20000	50000
AVL tree (AT)	Insert or delete nodes in 16 AVL trees	100000	10000
B tree (BT)	Insert or delete nodes in 16 B trees	100000	10000
RB tree (RT)	Insert or delete nodes in 16 RB trees	100000	10000

5.2 Workloads

Using the PMEM instructions, we constructed a workload consisting of benchmarks with data structures listed in the Table 2. The benchmarks are borrowed from or are similar to those used in previous studies [8, 18, 19, 30, 37, 42, 48]. For each benchmark, we construct an *operation* that is either a node insertion or deletion (except for String Swap). The operation is wrapped inside a durable transaction. Each benchmark receives an operation type and a key for each operation from an input file which contains the list of operations generated randomly. We used multiple data structures per benchmark to avoid excessive lock contention among multiple threads. In this work, we assumed that multicore shared memory accesses among transactions are solved by thread synchronization using locks, which guarantees mutual exclusion between concurrent transactions. We believe that this is a separate problem and not the focus of our work. And hence, each operation must obtain a lock for a data structure before the operation is performed and no other threads can interrupt the executing thread in the middle of the update. The table shows the number of initial operations per thread that are executed first to populate the data structure, which are then fast-forwarded in the simulator. To eliminate non-determinism from our experiments, the pthread barrier is used after initial operations so that all threads run together as they are simulated.

On each benchmark, we created a manual undo-logging version and a version for ATOM and *Proteus*. We found that rebalancing operations in self-balancing tree benchmarks (AVL tree, B tree, and RB tree) are challenging for the creation of undo logs because it is difficult to know which nodes will be modified at the start of the transaction. Therefore, our manual undo-logging assumes the worst and logs all nodes that could be modified by the operation. Furthermore, for simplicity, we assume that memory allocations and deallocations are performed in a failure-safe way so that our undo-logging need not cover them.

6 EVALUATION

Figure 6: Speedup comparison on NVMM, with software logging with PMEM as baseline.

In order to assess the performance benefits of *Proteus*, we implemented and compared the following schemes: software logging represented by an Intel PMEM based implementation of WAL, both with *pcommit* (PMEM+pcommit) and without it (as the base case), hardware logging represented by ATOM [19] including all of its optimizations (ATOM), and software-supported hardware logging represented by our scheme *Proteus* (Proteus) and without log write removal (Proteus+NoLWR). In order to see how close they perform to an ideal case, we also implemented PMEM but with logging removed (PMEM+nolog). The latter does not provide failure safety and is devoid of any logging overheads, and thus it is an ideal case.

The result of their speedup over the base case of PMEM without pcommit for all benchmarks and for the geometric mean of all benchmarks are shown in Figure 6. First, let us observe the PMEM+pcommit bars. They are significantly below 1.0 in all benchmarks, with a geometric mean of 0.79. This shows that moving

the MC and WPQ into the persistency domain is very helpful for performance. Next, consider the last bars (PMEM+nolog) that are significantly higher than 1.0, with a geometric mean of 1.51. This shows that the addition of logging code and its execution causes very significant performance overheads, whereas its removal speeds up execution by 51% on average. However, on benchmarks with complex data structures such as BT and RT, the speedups when logging is removed are very high: in the case of BT, logging code removal results in a 2.98× speedup. This is because, for complex data structures, it is difficult to determine which components of the data structure will need to be undo logged. For example, tree balancing operations may affect only a few nodes in the best case or the entire tree in the worst case. Thus, logging code needs to assume conservatively that a high number of nodes will be affected by a transaction.
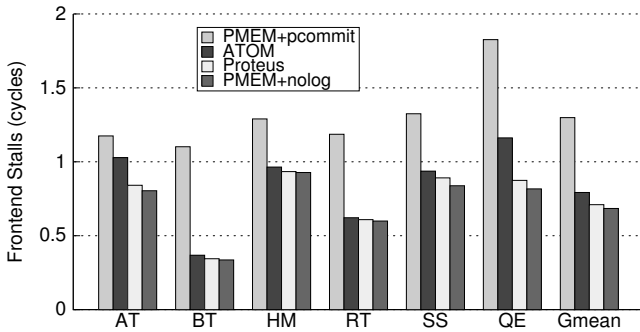

Figure 7: The normalized stall cycles of the pipeline front-end.

Now let us examine ATOM and *Proteus*. ATOM performs quite well, achieving a 1.33× speedup on average while *Proteus* achieves a geometric average of 1.46× speedup. In other words, *Proteus* is faster than ATOM by $\frac{1.46}{1.33} - 1 = 10\%$. Furthermore, *Proteus*'s speedup is only 3.3% lower than the ideal case of no logging. In ATOM, a log entry is automatically generated right before a store gets retired and the retirement is delayed until the log entry becomes durable. Due to this constraint, the rate at which store operations are completed is reduced. For ATOM's design, this is necessary to solve the dependency between log entries and stores. However, *Proteus* does not have this limitation because the LogQ manages the dependency. This allows *Proteus* to support concurrent logging as long as log-flushes do not have dependences with preceding stores. In addition, it allows stores to complete earlier. We found that concurrent logging provides an important advantage in overall performance.

To analyze the performance difference between ATOM and *Proteus*, we investigated the stall cycles at the front-end of the pipeline before instruction dispatch. The front-end could be stalled by a lack of free resources in the ROB, physical registers, or LSQ. Figure 7 shows the stall cycles normalized to the stall cycles of PMEM+nolog in the front-end. ATOM has 12% more stalls than *Proteus* and 16% more stalls than the ideal case. On the other hand, the number of stall cycles in *Proteus* is fairly close to the ideal case, only 4% more stalls. These results show that ATOM creates more pressure on the pipeline and eventually stalls it, but *Proteus* is free from this limitation.

Figure 8 compares the number of NVMM writes for each benchmark, normalized to the number of NVMM writes of PMEM+nolog. On average, ATOM has three times more writes to NVMM (3.4×), compared to PMEM without logging. In benchmark (QE), it more than quadruples the writes to NVMM and in the worst case (AT), it has six times more writes to NVMM. The increase in number of writes is due to logging (creation and truncation). This is significant because it cuts the write endurance of NVMM by more than three quarters. In contrast, *Proteus* only increases the number of writes slightly. In the worst case (AT), the increase in writes is still relatively low, at 6%. The reason for *Proteus*'s advantage is that most log updates are held at the LPQ and flash cleared when a transaction ends, thanks to the fact that the MC is part of the persistency domain. Thus, most log flushes do not even go to the NVMM.
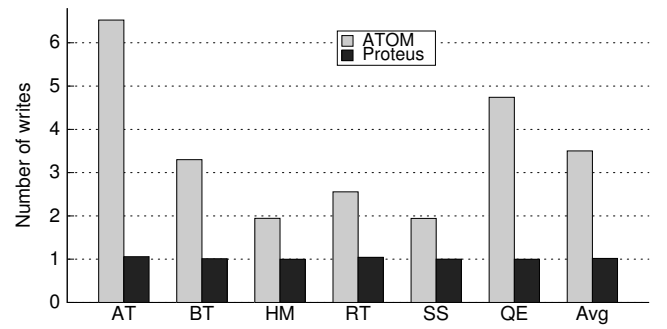

Figure 8: The number of NVMM writes, normalized to PMEM with no logging.

7 SENSITIVITY STUDY

In order to quantify the impact of memory latency on the performance of the logging schemes we studied, we ran our experiments with slower NVMM and faster DRAM. Moreover, later in this section, we expand our study into the impact of the new components of *Proteus* on performance with varied hardware structures and their tradeoffs.
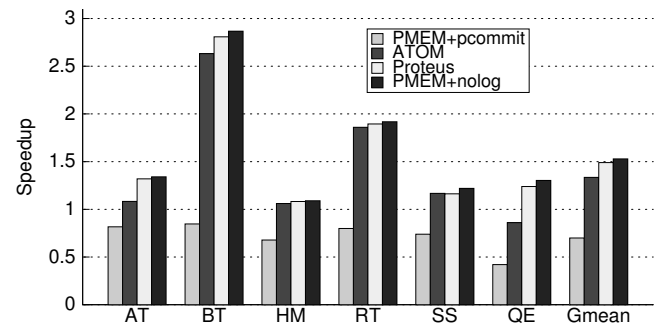
7.1 Performance on slow NVM

Figure 9: Speedup comparison on slow NVMM (300ns for write), with SW logging as baseline(PMEM).

Current NVM read and write latencies, for various technologies, have not yet reached our previous assumption of fast NVM with 50ns read and 150ns write latencies. In order to see the impact of our scheme on slower NVM devices, we modeled a higher write latency at 300ns while keeping the read latency at 50ns. Not surprisingly, the overall performance of all test cases decreased 10-23% with slow NVMM compared to faster NVMM, indicating that slow write latency affects the performance. Figure 9 shows the speedup of each benchmark on a slow NVMM, with the baseline of PMEM as before. The geometric mean of speedups are 1.33 for ATOM, 1.49 for *Proteus*, and 1.53 in the ideal case. Compared to Figure 6, the speedup of the ideal case is slightly improved since it has fewer writes than the baseline. On the other hand, *Proteus* is also less affected by write latency and still maintains superior performance close to the ideal case. However, the speedup of ATOM stays the same over the baseline, indicating that ATOM is more influenced by write latency than *Proteus*. On average, *Proteus* experienced only a 10% performance decrease compared to the faster NVMM, which is superior than the 12% decrease of ATOM's. Hence, *Proteus*'s advantage becomes more substantial with longer NVMM write latencies.
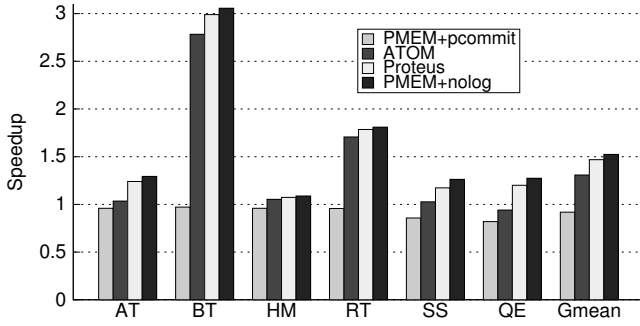
7.2 Performance on DRAM

Figure 10: Speedup comparison on DRAM, with software logging with PMEM as baseline.

Figure 10 shows the speedup comparison on DRAM. The rationale behind running with DRAM is to study the performance of logging when battery backed DRAM solutions, like NVDIMM, are used. On DRAM, we found that *Proteus* still performs quite well. We observe average speedups of 1.31 for ATOM, 1.47 for *Proteus*, and 1.52 for the ideal case. Compared to Figure 6, overall performance is improved from 11% for ATOM up to 30% for PMEM+pcommit. Because of its modest performance improvement, ATOM's speedup over the baseline decreased slightly. Considering the fact that logging is completed at the MC with posted log optimization, the speedup degradation in ATOM over the baseline is the result of the tradeoff between faster accesses to DRAM and relatively fixed logging latency in ATOM. On DRAM, the overhead of logging plays a slightly bigger role in the overall performance of ATOM. On the other hand, *Proteus* maintains its superior performance with a 13% performance improvement over NVMM. We found the reason is that it hides the logging latency more effectively using concurrent logging. Although it is not described in the figure, the speedup difference obtained

when changing from the LogQ size of 8 to 16 increases notably, from 0.02 with NVMM to 0.11 with DRAM. The bigger LogQ helps *Proteus* hide the logging latency even with fast memories.

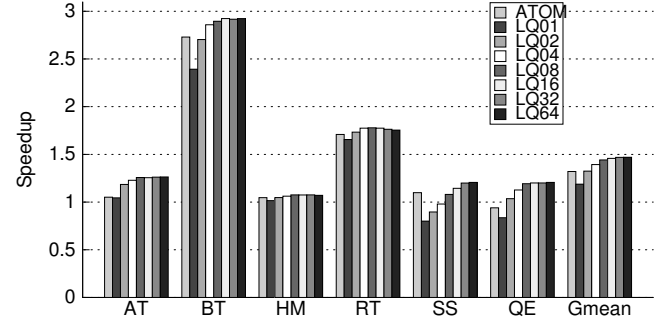## 7.3 Analysis of logging components

Figure 11: Speedup comparison with varying LogQ sizes. The baseline is software logging with PMEM.

Figure 11 shows the impact on speedup by varying the LogQ size for each benchmark. From the figure, average speedup shows an increasing trend as LogQ size is varied from 1 to 64, but we also can see the diminishing returns as the size increases. These results suggest that a LogQ size of 8 gives a 1.44x speedup. Speedups start to saturate with a LogQ size of 8, with very little improvement (1-2%) as the size is doubled. A LogQ size of 64 gives more than 1.47x speedup which is only 2.4% below the ideal case. We chose a LogQ size of 16 as our optimal configuration instead of 8 considering its superior performance on DRAM and its dominant performance over ATOM across all benchmarks, particularly since it performs worse than ATOM in benchmark (SS) with only 8 entries in the LogQ.
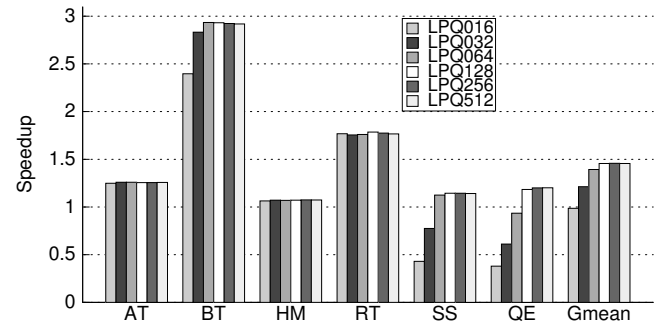
Figure 12: Speedup comparison with varying LPQ and LogQ sizes. The baseline is software logging with PMEM.

Figure 12 shows the combined interaction of varying LogQ along with LPQ. From the previous study we pick the optimal LogQ size of 16 and use it to study different LPQ sizes. We found that benchmarks require a certain LPQ size depending on their transaction size. As long as a large enough LPQ is provided, overall performance is unaffected. Otherwise, the performance drops rapidly for smaller sizes. We select the size of 256 entries as our configuration.

Table 3: Speedups for large transactions.

Transaction Size	1024	2048	4096	8192
Proteus	1.20	1.24	1.23	1.24
PMEM+nolog(ideal)	1.23	1.25	1.25	1.27

We implemented a microbenchmark with variable-sized, large transactions based on the linked list benchmark. The number of elements updated per node is taken as a variable to stress *Proteus*. Each transaction completes once all elements on a node are updated, and we chose the number of elements for our experiments as 1024, 2048, 4096, and 8192. We found that the benchmarks generate 20×, 39×, 78×, and 156× more log entries per transaction compared to the existing benchmarks. Although a large number of those log entries are filtered by LLT, we still found that 7×, 13×, 26×, and 52× more log entries per transaction are flushed to the MC. Table 3 compares the speedups of *Proteus* and the ideal case (PMEM+nolog) with varying transaction sizes. The data suggests that the performance of *Proteus* is still very close to the ideal case. The result also suggests that hardware structures (LogQ, LLT, and LPQ) used in *Proteus* are able to sustain large transactions.

Table 4: LLT miss rate (%) for different benchmarks.

Benchmark	AT	BT	HM	RT	SS	QE
miss rate	37.2	36.1	39.2	51.6	24.5	22.5

Table 4 shows the LLT miss rate for the benchmarks collected with an LLT size of 64 entries. The data shows that the benchmarks chosen did exhibit varied miss rates from 22.5% to 51.6%. Higher LLT miss rates indicate more log entries per transaction. The data shows that the LLT efficiently absorbs half to three quarters of logging traffic, helping to reduce the memory bandwidth devoted to logging. Furthermore, the decreased log entries help to reduce the necessary size of the LogQ, the LPQ, and the log area in NVMM.

8 RELATED WORK

The emergence of new non-volatile memory technologies have paved the way for designing systems with fast non-volatile main memory, which has opened many new research directions. Modern systems are optimized for volatile memory, such as DRAM, which makes it difficult to maintain data consistency in NVMM after power failure. In order to maintain data consistency in NVMM, researchers have borrowed traditional storage techniques such as copy-on-write or write-ahead logging and applied them in the context of NVMM. Condit et al. [9] proposed a new file system for NVM and used shadow paging, one form of the copy-on-write technique, to provide an atomic file-system update. This paper also proposed epoch persistency to ensure ordering persists. Ren at al. [41] proposed a hardware supported failure-safe system using copy-on-write.

Using a copy-on-write approach for data consistency on NVMM is not easy to implement because remapping in hardware is expensive. Moreover, a large update granularity, which often amortizes the remapping overhead, is less suitable for use in byte-addressable NVMM. Hence, instead of using copy-on-write, the majority of papers on NVMM have adopted WAL which allows transactions at smaller granularity and does not require remapping. For example, Mnemosyne [43] proposed a library to access NVMM directly and used redo-logging to provide an atomic durable transaction. On the other hand, NV-Heaps [8] used software undo-logging for its atomic transaction implementation and improves its performance by logging at a large granularity instead of logging before every write. Rewind [7] proposed a library to manage NVMM directly from the user application and it also utilized a software undo-log approach for its durable transaction.

In general, logging operations are expensive because they force an ordering between logging operations and the following stores. Therefore, recent works discussed about the cost of logging in the transaction and endeavored to reduce it. For example, Atlas [6] uses software undo-logging in its durable transaction and proposes programming models to optimize them. Pelley et al. [38] proposes NVRAM group commit which groups the transactions executed into a batch and commits them at once, amortizing its commit overhead. Similarly, Kolli et al. [23] proposed DCT, which defers a transaction commit on a multi-threaded application and commits them together while resolving possible conflicts among them. Recently, Liu et al. [29] proposed a software logging approach, DudeTM, which efficiently eliminates the ordering constraint using asynchronous redo-logging. However, it creates more memory writes and requires additional memory space from both DRAM and NVMM compared to previous approaches.

Hardware logging approaches for mitigating logging overheads were proposed in multiple papers. Lu et al. [30] proposed LOC which relaxes the ordering constraint in transactions by using asynchronous redo-logging with background hardware supports. Similar to DudeTM, it requires additional memory accesses and space. Furthermore, it requires redirecting reads to logs or blocking reads during the logging operation until it has been committed to memory, which is generally considered an expensive operation. It also needs additional hardware support for updating memory in the background. Doshi et al. [10] proposed synchronous hardware redo-logging for atomic durability. However, their solution is not optimized for the case with multiple updates to the same address in one transaction. In this case, they simply create multiple log entries for the same data, which consumes unnecessary memory bandwidth and also degrades performance. Recently, Joshi et al. [19] proposed ATOM which uses synchronous undo logging and includes hardware to help create only one log entry per update per transaction. They also showed better performance compared to a previous redo-logging scheme [10]. In our work, we show that we attain better performance than ATOM, thanks to concurrent logging and log write removal.

A persistent cache hierarchy has also been proposed and used in multiple works [17, 32–34, 45, 48]. These works bring the data caches into the persistency domain. Hence, stores are considered durable once they leave the store buffer. Obviously, these systems do not have flushing overheads because data does not need to be forced out of the cache, but they still need to manage logging schemes. The ordering constraint between log entries and subsequent stores can be handled in the processor.

A few works [13, 21, 35] have used NVMM with write-ahead logging for a database system usually placed on disks. NV-Logging [13]

shows that replacing a whole disk with NVM is an expensive solution per dollar. Instead, they show it is cost effective to place only a logging subsystem on NVMM enabling concurrent logging for multiple transactions. Similarly, Oh et al. [35] proposes PPL which deploys a similar technique but on a smaller scale with SQLite on mobile systems. NV-WAL [21] optimizes PPL considering byte-addressability, write reordering, and user-level heap management on NVMM.

9 CONCLUSION

We have described a new logging approach, *Proteus* for durable transactions that achieves the favorable characteristics of both prior software and hardware approaches. Like software, it has no hardware constraint limiting the number of transactions or logs available to it, and, like hardware, it has very low overhead. Our approach introduces two new instructions: log-load creates a log entry by loading the original data, and log-flush writes the log entry into the log. We add hardware support, primarily within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data. We also propose a novel optimization at the memory controller that is enabled by a persistent write pending queue in the memory controller. We drop log updates that have not yet written back to NVMM by the time a transaction is considered durable.

We compared our design against ATOM [19] and a software only approach. Our experiments show that *Proteus* improves performance by 1.44-1.47× depending on configuration, on average, compared to a system without hardware logging and 9-11% faster than ATOM. We also show that our design performs closely to the ideal case. A significant advantage of our approach is dropping writes to the log when they are not needed. On average, ATOM makes 3.4× more writes to memory than our design.

ACKNOWLEDGEMENT

REFERENCES

[1] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

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

[3] 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

[4] 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

[5] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 107–119. https://doi.org/10.1145/3079856.3080230

[6] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the*

[7] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead System for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

[8] 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

[9] 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

[10] K. Doshi, E. Giles, and P. Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[11] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[12] Yiming Huai, Frank Albert, Paul Nguyen, Mahendra Pakala, and Thierry Valet. 2004. Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions. *Applied Physics Letters* 84, 16 (2004), 3118–3120. https://doi.org/10.1063/1.1707228

[13] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[14] Intel. 2016. *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[15] Intel. 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[16] 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

[17] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[18] 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

[19] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[20] 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 *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. 480–617. https://doi.org/10.1109/ISSCC.2007.373503

[21] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 385–398. https://doi.org/10.1145/2872362.2872392

[22] E. KÃijltÃijrsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 256–267. https://doi.org/10.1109/ISPASS.2013.6557176

[23] 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

[24] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783761

[25] M. H. Kryder and C. S. Kim. 2009. After Hard Drives - What Comes Next? *IEEE Transactions on Magnetics* 45, 10 (Oct 2009), 3406–3413. https://doi.org/10.1109/TMAG.2009.2024163

[26] 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

[27] 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

[28] Zhongqi Li, Ruijin Zhou, and Tao Li. 2013. Exploring High-performance and Energy Proportional Interface for Phase Change Memory Systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 210–221. https://doi.org/10.1109/HPCA.2013.6522320

[29] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[30] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *32nd International Conference on Computer Design*.

[31] 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

[32] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2150976.2151018

[33] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles Morrey. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. In *International Conference on Extending Database Technology*.

[34] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles Morrey. 2015. Zero-Overhead NVM Crash Resilience. In *Non-Volatile Memories Workshop*.

[35] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite Optimization with Phase Change Memory for Mobile Applications. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1454–1465. https://doi.org/10.14778/2824032.2824044

[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. http://dl.acm.org/citation.cfm?id=2665671.2665712

[38] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2 (Oct. 2013), 121–132. https://doi.org/10.14778/2732228.2732231

[39] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1669112.1669117

[40] Raghunath Rajachandrasekar, Sreeram Potluri, Akshay Venkatesh, Khaled Hamidouche, Md. Wasi-ur Rahman, and Dhabaleswar K. (DK) Panda. 2014. MICCheck: 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

[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] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/3079856.3080240

[43] 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

[44] C. Wang, Q. Wei, J. Yang, C. Chen, and M. Xue. 2015. How to be consistent with persistent memory? An evaluation approach. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 186–194. https://doi.org/10.1109/NAS.2015.7255223

[45] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[46] 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

[47] J. Joshua Yang and R. Stanley Williams. 2013. Memristive Devices in Computing System: Promises and Challenges. *J. Emerg. Technol. Comput. Syst.* 9, 2, Article 11 (May 2013), 20 pages. https://doi.org/10.1145/2463585.2463587

[48] 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

[49] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1555754.1555759