Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support

Jishen Zhao[†] Sheng Li[‡] Doe Hyun Yoon[♯∗] Yuan Xie[†♮] Norman P. Jouppi[§∗]
[†]Pennsylvania State University [‡]Hewlett-Packard Labs [♯]IBM Research
[♮]AMD Research China Lab [§]Google
[†]{juz138, yuanxie}@cse.psu.edu [‡]sheng.li@hp.com [♯]dyoon@us.ibm.com [§]jouppi@acm.org

ABSTRACT

Persistent memory is an emerging technology which allows in-memory persistent data objects to be updated at much higher throughput than when using disks as persistent storage. Previous persistent memory designs use logging or copy-on-write mechanisms to update persistent data, which unfortunately reduces the system performance to roughly half that of a native system with no persistence support. One of the great challenges in this application class is therefore how to efficiently enable atomic, consistent, and durable updates to ensure data persistence that survives application and/or system failures. Our goal is to design a persistent memory system with performance very close to that of a native system. We propose Kiln, a persistent memory design that adopts a nonvolatile cache and a nonvolatile main memory to enable atomic in-place updates without logging or copy-on-write. Our evaluation shows that Kiln can achieve 2× performance improvement compared with NVRAM-based persistent memory with write-ahead logging. In addition, our design has numerous practical advantages: a simple and intuitive abstract interface, microarchitecture-level optimizations, fast recovery from failures, and eliminating redundant writes to nonvolatile storage media.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Primary Memory*

Keywords

Persistent Memory, Non-volatile Memory

1. INTRODUCTION

Applications that require high reliability, such as databases and file systems, need to periodically store critical data in nonvolatile devices so the data can survive system failures or program crashes. Commodity computing systems employ slow block-addressable storage media, such as spinning disks or flash, to store this critical data. Due to hardware (PCIe or SATA I/O delay) and software (legacy block-oriented file system interfaces) costs, applications suffer from significant throughput degradation.

Persistent memory is a new technology incorporating the properties of both main memory and storage. An application can directly access persistent data through a memory interface with loads and stores, without paging data blocks from/to a storage device or context switching while servicing page faults. Recent work [12,45] has demonstrated much higher program throughput (up to 32×) by utilizing byte-addressable nonvolatile memory technologies (NVRAM) such as spin-transfer torque RAM (STT-MRAM) or phase-change memory (PCM) to build persistent memory. These studies operate directly on nonvolatile data that is accessible through the processor-memory bus, eliminate the overhead of PCIe or SATA accesses and legacy block-oriented file-system interfaces, and update the persistent data structures at cache line granularity without the need for batching. Neither memory (SRAM, DRAM, and flash) nor storage media (hard drives and optical discs) in current commercial systems are both nonvolatile and byte-addressable. Hence, NVRAM-based persistent memory enables a new class of applications that can store pointer-rich, user-defined data structures directly in a nonvolatile memory and process a large amount of data at low latency and high bandwidth.

A caveat for persistent memory design is that system failures or program crashes may corrupt the state of data structures. For instance, a power outage may occur while an application is inserting a node in a doubly-linked list. If only one pointer is written out to nonvolatile devices (NVRAM) and the other is still in volatile devices (processor caches or DRAM), the doubly-linked list will be broken and not usable after the crash. Ideally, a persistent memory system (hardware, software, or a combination of both) must ensure safe data updates so that data integrity is maintained in the presence of system failures or program crashes. Borrowing the ACID (atomicity, consistency, isolation, and durability) [38] concept from the database community, persistent memory systems must update a set of programmer-defined nonvolatile locations in an atomic, consistent, and durable way to enforce crash consistency (i.e., **persistence**).

Unfortunately, supporting persistence in memory still incurs significant performance cost, even with the latest proposals. Existing persistent memory designs employ logging or copy-on-write (COW) to manage persistent data updates. Logging mechanisms track the changes to critical data by maintaining a set of journals, which store old data values

[∗]This work was done while the authors were with Hewlett-Packard Labs.

MICRO-46, December 7-11, 2013, Davis, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2638-4/13/12$15.00.

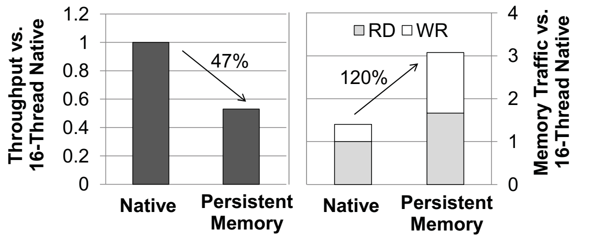


Figure 1: Comparison between a native system with no persistence support (Native) and log-based persistent memory (Persistent Memory). Speedups of transaction throughput (higher is better) and memory traffic (lower is better), including reads and writes, are averaged across benchmarks.

(undo logging) or new updates (redo logging). COW stores new updates in a temporary data copy, while the real data is unchanged. However, these mechanisms increase the demand of storage space and reduce system performance by increasing memory traffic with extra data transfers. Furthermore, previous persistent memory designs use instructions such as flush (`clflush`) and memory fence (`mfence`) to ensure consistency by flushing the dirty lines in caches at the barrier of each persistent memory update. As a result, we observe a large performance gap between a system with a persistent memory and a "native system" (i.e., with no persistence support). Persistent memory implementations using off-chip NVRAM and logging incur a 120% increase in memory traffic (60% in reads and 180% in writes) and only achieve 53% of the throughput of a native system (Figure 1). Therefore, **our goal** is to design a persistent memory with performance close to that of the native system.

We propose Kiln[1], a persistent memory design that employs a nonvolatile last level cache and a nonvolatile memory to construct a persistent memory hierarchy. Our design allows a persistent memory system to directly update the real in-memory data structures, rather than performing logging or COW. We refer to these direct updates to the real in-memory data structures as **in-place** updates. We also develop a set of light-weight software and hardware extensions to facilitate atomicity and consistency support. With in-place updates, Kiln can achieve 91% of native system performance, which is about a 2× improvement over log-based persistent memory designs using NVRAM. In particular, we make the following contributions:

- We propose a persistent memory design that closes the performance gap between systems with and without persistence support. Our persistent memory allows in-place updates to real in-memory data structures, without performing logging or COW.
- We provide an optimized flush operation, which enforces the order of persistent memory updates without flushing the entire cache hierarchy or executing flush and memory fence instructions.
- We develop a simple and intuitive software interface and a set of light-weight ISA and architecture extensions to provide atomicity and consistency support for our persistent memory.

[1]"Kiln" was once used by ancient Mesopotamians to bake the clay tablets with temporary scripts and turn them into permanent records. We name our persistent memory design Kiln, because it is analogous to the persistent memory that turns volatile data into permanent records.

2. BACKGROUND AND RELATED WORK

Protecting data against system failures and crashes forces a trade-off between performance and reliability. In this section, we study the requirements of developing persistent memory and investigate the persistence mechanisms of previous work.

2.1 Properties of Persistent Memory

Persistence has been well investigated in databases and file systems. We borrow the concept of atomicity, consistency, isolation, and durability (ACID) [38] from the database community to study the properties of persistent memory. These four properties can be separately maintained in different manners in a system. For example, transactional memories (TMs) [22] maintain A, C, and I, separated from D, while a recent study on failure-atomic msync [36] focuses on A and D.

In particular, a persistent memory system needs to ensure atomicity, consistency, and durability. First of all, a persistent memory system contains nonvolatile devices so each data update is retained during power loss, crashes, or errors. This is referred to as the **durability** property. Second, because the granularity of programmer-defined data updates can be larger than the interface width of the persistent memory, a single update is typically serviced as multiple requests. Therefore, sudden power losses or crashes can leave an update partially completed, corrupting the persistent data structures. To address this issue, each single update must be "all or nothing", i.e., either successfully completes or fails completely with the data in persistent memory intact. This property is **atomicity**. Third, **consistency** requires each update to convert persistent data from one consistent state to another. Taking an example where an application inserts a node to a linked list stored in persistent memory, a system (including software programs and hardware) needs to ensure that the initial values of the node are written into the persistent memory before updating the pointers in the list. Otherwise, the persistent data structure can lose consistency with dangling pointers in a sudden crash, leading to a permanent corruption not recoverable by restarting the application or the system. Typically, programmers are responsible for defining consistent data updates, because only the programmers know what it means for application data to be in harmony with itself. Of course, programmers can leverage runtime API to do this. While executing the software programs, hardware and system software need to preserve the demanded consistency.

The fourth property, **isolation**, ensures that concurrent data updates are invisible to each other. Today, a programmer writing portable code atop a POSIX-compliant OS and hardware has two separate families of mechanisms for solving two isolation problems. One family of mechanisms is used to ensure orderly race-free access to data in multithreaded or multiprocess concurrent programs. This set of mechanisms includes mutexes, semaphores, TMs, and lock-free/wait-free data structures and algorithms. The other family of mechanisms is used to update data in durable media. This set of mechanisms includes system calls such as `write()`, `fsync()`, and `mmap()`/`msync()`. Commodity systems use separate and orthogonal mechanisms for handling isolation in the face of concurrency and durable updates. Our persistent memory design permits the same kind of orthogonal separation of concerns. Various concurrency control mechanisms can be integrated with our design.

Specifically, our persistent memory design maintains A and D, preserves C that is defined by programmers, and relies on concurrency control mechanisms to support isolation.

2.2 Maintaining Atomicity by Multiversioning

Multiversioning is a common method to ensure atomicity. With multiversioning, multiple copies of data exist. When performing updates to one copy of data, another copy is left intact. If one copy of data is corrupted by a partial update, another copy is still valid and available for recovery.

Most previous work on persistence, e.g., persistent object systems [4,9,12,19,28,42,47], the Java persistence API [2,33], RVM [39], Rio file cache [11], Stasis [40], Mnemosyne [45], eNVy [48], and UBJ [29], employ one of two techniques to maintain multiversioning: write-ahead logging (or journaling) [12, 20, 31, 39, 43, 45] or COW [11, 13, 23, 44, 48] (Figure 2 (a) and (b)). Several previous studies investigated the use of battery-backed RAMs as persistent storage [8,14,18]. Although battery-backed RAMs are byte-addressable, these designs inefficiently access the RAMs through a driver like disks and adopt database management systems (DBMS) or file systems to implement logging or COW to manage the persistent memory. NV-heaps [12] and Mnemosyne [45] adopt durable software transactional memory (STM) to support persistence for in-memory data objects. Both designs enforce atomic transactional updates by maintaining a redo log.

Both logging and COW mechanisms impose significant performance overhead by explicitly executing logging or data copying instructions. While the software overhead is tolerable with traditional disk-based persistent memories where the I/O delay dominates the performance overhead, the fraction of software overhead increases dramatically when the persistent memory can be accessed at a much faster speed [10]. Furthermore, duplicated data (logs or data copies) traverse the cache hierarchy to the memory, contaminating caches with non-reusable cache lines. Therefore, the key reason that the native system runs fast is that it performs in-place updates to the real in-memory data, without explicitly duplicating the data like logging or COW does. However, in-place updates are hard to implement in most previous NVRAM-based persistent memory designs [12,13,44,45], which maintain persistence in a single-level memory. In such systems, at least one more copy of data needs to be stored in addition to the real data, to maintain multiversioning.

An exception of ensuring atomicity without multiversioning is when an update can be completed instantaneously, typically with very small granularity of memory stores. Examples of such cases are updating a single variable [45] or a memory store of the granularity the same as the bus width [13, 35]. Unfortunately, these studies do not provide any mechanisms that can be applied to in-place updates of larger granularities.

2.3 Preserving Consistency by Ordering

Controlling write ordering is a primary mechanism to preserve consistency in application programs. Ordering means that the order that updates become permanent must match the order in which they are issued. A mismatch can happen when processor caches and memory controllers reorder memory requests to optimize performance. A persistent memory employs ordering control mechanisms to prevent mismatch. Most previous persistent memory designs ensure the ordering by write-through caching [45] or bypassing the processor caches entirely, flush, memory fence [35, 44, 45], and msync operations, each imposing high performance costs. With write-through caching, each memory store needs to wait until reaching the main memory. Flush and memory fence mechanisms can cause a burst of memory traffic and block subsequent memory stores. Furthermore, most previous designs [35, 44, 45] employ instructions such as `clflush`, which flushes dirty cache lines to ensure ordering, with a latency that can be up to several milliseconds. Besides the long latency, flushing an entire cache can also evict the working sets of other applications from the cache. BPFS [13] adopted an epoch barrier mechanism to minimize the flush traffic, however at the cost of reduced durability strength that leads to potential data loss.

3. DESIGN OVERVIEW

Kiln adopts a new persistent memory architecture consisting of a nonvolatile cache (NV cache) and a nonvolatile memory (NV memory), naturally forming a multiversioned persistent memory hierarchy (Figure 2 (c)). The newly updated versions are dirty NV cache lines. The old versions are clean data stored in the NV memory, which will be automatically updated when the dirty NV cache lines are evicted. With this multiversioned persistent memory hierarchy, Kiln simplifies persistent memory update operations by allowing memory stores to be performed in-place to the persistent data structures in the NV cache, without logging or COW. Therefore, Kiln's memory store operations are similar to those of the native system. As a result, Kiln's performance is also very close to that of the native system, yielding a significant performance improvement over previous NVRAM-based persistent memory designs. Table 1 qualitatively compares Kiln with the native system and related persistent memory designs in terms of memory update mechanisms and support of atomicity and ordering.

3.1 Assumptions and Definitions

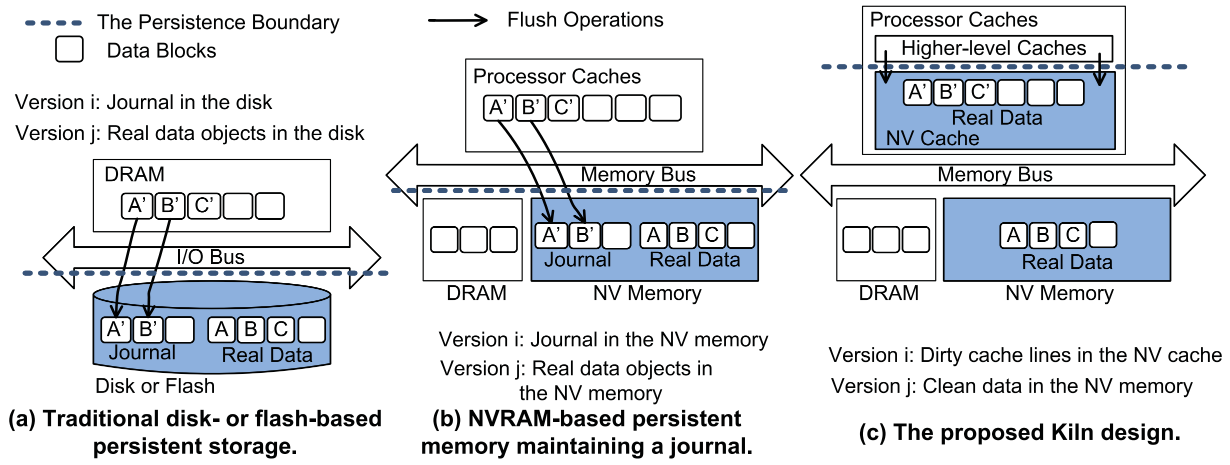
Mapping data to a hybrid memory address space: We assume that DRAM and NVRAM are both deployed on the processor-memory bus and mapped to a single physical address space. Kiln stores the user-defined critical data in NVRAM. The DRAM is used to store data that is not required to be persistent and can be overwritten frequently. Examples of such data are stacks and data transfer buffers. Runtime systems such as the ones developed by prior studies [12,45] can be employed to expose the NVRAM address space to persistent data objects.

Program hints on persistent memory transactions: Kiln adopts program hints to decide when and what data blocks need to be persistent. Recent NVRAM-based persistent memory designs [12, 45] obtain this information by allowing users to define durable STM transactions. Similarly, Kiln exposes to programmers an interface of "persistent memory transactions", which are groups of instructions performing persistent memory updates. Kiln reads users' input to define the beginning and end of each transaction.

States of a persistent memory transaction: Each persistent memory transaction will go through three states: *in-flight*, *committing*, and *committed*. After the first instruction of a persistent memory transaction starts execution,

Table 1: Comparison of Kiln with previous work. (⋆ means In-place updates are only performed for memory stores to a single variable or at the granularity of the bus width. ⋄ means ordering is maintained among the writes to the disk or flash by flush or checkpointing.)

Designs	Mechanisms					Persistence Support	
	In-place	Logging	COW	clflush/msync/fsync	mfence/barrier	Atomicity	Ordering
BPFS [13]	⋆	**No**	Yes	**No**	Yes	√	√
Mnemosyne [45]	⋆	Yes	Yes	Yes	Yes	√	√
NV-heaps [12]	No	Yes	**No**	**No**	Yes	√	√
CDDS [44]	No	**No**	Yes	Yes	Yes	√	√
UBJ [29]	**Yes**	Yes	Yes	⋄	⋄	√	√
eNVy [48]	No	**No**	Yes	⋄	⋄	√	√
Native System	**Yes**	**No**	**No**	**No**	**No**	×	×
Kiln	**Yes**	**No**	**No**	**No**	**No**	√	√



The Persistence Boundary
Data Blocks
Flush Operations
Version i: Journal in the disk
Version j: Real data objects in the disk
DRAM
A' B' C'
I/O Bus
A' B'
A B C
Journal
Real Data
Disk or Flash
(a) Traditional disk- or flash-based persistent storage.
Processor Caches
A' B' C'
Memory Bus
A' B'
A B C
Journal
Real Data
DRAM
NV Memory
Version i: Journal in the NV memory
Version j: Real data objects in the NV memory
(b) NVRAM-based persistent memory maintaining a journal.
Processor Caches
Higher-level Caches
A' B' C'
Real Data
NV Cache
Memory Bus
A B C
Real Data
DRAM
NV Memory
Version i: Dirty cache lines in the NV cache
Version j: Clean data in the NV memory
(c) The proposed Kiln design.

Figure 2: Overview of Kiln persistent memory design and previous work.

the transaction becomes an *in-flight* transaction. When the last instruction of the transaction completes execution, the transaction is in *committing* state. In this state, Kiln will perform the clean-on-commit operation (Section 3.3) and update the state of persistent data structures (Section 3.4). When these operations are completed, the transaction is *committed*. All data updated by this transaction is now persistent.

3.2 In-place Updates without Logging or COW

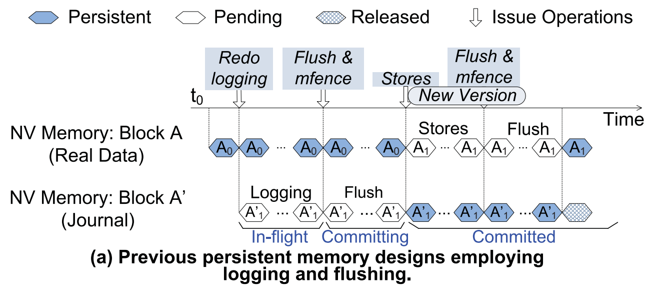
Previous persistent memory designs maintain multiversioning by software, using application or OS libraries. From the perspective of software, a memory system is a flat address space, consisting of a sequence of pages. Therefore, previous persistent memory designs need to explicitly create multiple regions, logs or temporary data copies, to maintain multiple versions of data. Different from software, hardware views a memory system as a hierarchy with multiple levels of processor caches and a main memory. This hierarchy naturally stores different versions of data in different levels.

Leveraging this hierarchy, we design a multiversioned persistent memory that includes a last-level NV cache and a NV memory (Figure 2(c)). The dirty NV cache lines are one version and the clean data in the NV memory are another. Both versions have the same address so this persistent memory hierarchy directly performs in-place updates to real data structures. We allow in-flight and committing persistent memory transactions to overwrite data values in processor caches (including the NV cache), but not in the NV memory. Therefore, the version stored in the NV memory is persistent if a system crashes when a persistent memory transaction is executing or committing. We allow NV cache lines of committed persistent memory transactions to be written back to the NV memory. However, we do not allow evictions from higher-level volatile caches to overwrite a NV cache line that is being written back. Therefore, the version stored in the NV cache is persistent if a system crashes when writing back a NV cache line.

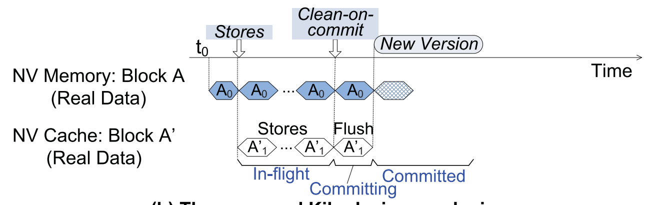
Our work is different from previous work that use a disk cache or flash buffer to improve the persistence performance, such as eNVy [48], and UBJ [29]. The file cache and flash buffer in these designs are simply used as buffers of the journal or temporary copies of data which still serve for logging or COW, rather than as a way of enabling in-place updates.

3.3 Ordering Control by Clean-on-commit

We employ an optimized flushing operation called *clean-on-commit* to preserve the ordering of persistent memory updates, when a persistent memory transaction is committing. Unlike previous work, Kiln allows cache controllers to issue flush requests without explicitly executing instructions such as `clflush` or `mfence`. We allow out-of-order write-backs of any dirty cache lines in the volatile caches, including those being updated by in-flight persistent memory transactions. The cache controllers will track the dirty cache lines that are updated by an in-flight persistent memory transaction and still remain in the volatile caches. The architecture extension in the NV cache (Section 4) will track the dirty NV



(a) Previous persistent memory designs employing logging and flushing.



(b) The proposed Kiln design employing in-place updates and clean-on-commit.

Figure 3: Comparison of the timeline of Kiln and previous persistent memory designs. Block A represents the data block (with a size of multiple cache lines) of an old valid version. Block A' represents the new version being updated.

cache lines updated by an in-flight persistent memory transaction. When a persistent memory transaction commits, typically a large portion (demonstrated in Section 6) of its dirty cache lines have already been written to the NV cache. Therefore, only the remaining dirty cache lines updated by the transaction in volatile caches need to be flushed. After all the dirty cache lines that belong to the committing transaction are flushed to the NV cache, the state of the transaction transitions from committing to committed.

The clean-on-commit operation is improved over the ordering mechanisms of previous designs in four aspects. First, clean-on-commit only flushes the volatile dirty cache lines of the committing persistent memory transactions. Many previous designs [44, 45] employ flushing instructions (e.g., `clflush`) that unnecessarily flush the dirty cache lines out of the cache hierarchy. Second, the memory traffic to perform the flushes is significantly reduced because we only flush a small number of cache lines. Third, the bandwidth of processor-cache buses is much higher than that of the off-chip memory bus, and therefore the flush operations can be completed much faster. Finally, clean-on-commit will be issued in the same order as the commits of persistent memory transactions, and therefore does not employ memory fence or barrier instructions which block other memory accesses. However, clean-on-commit requires bookkeeping functionality to be added to the volatile cache controllers. We will discuss the mechanisms and the overhead in Section 4.

3.4 Timeline of a Transaction

With in-place updates and clean-on-commit, Kiln provides a way to reduce the latency of data persistence by committing the persistent memory transactions right after all the updates arrive at the NV cache, rather than waiting for the updates to be flushed to the NV memory. Figure 3 shows the execution timeline of Kiln compared to that of a persistent memory system with redo logging. We do not show an example with undo logging, because its performance is usually worse than that of redo logging.

Figure 3(a) shows the sequence of updating persistent memory that employs redo logging to a journal in the NV memory. An in-flight persistent memory transaction keeps adding new data values and their addresses to a journal. This is followed by flush and memory fence operations to ensure that all the journal updates reach the NV memory immediately after they are issued. A persistent memory transaction becomes committed after the last instruction in a transaction is executed and all the logs are flushed into the NV memory. Then, the system can overwrite the real data structures in the NV memory. Figure 3(b) shows the timeline of Kiln. After executing the last instruction in an in-flight transaction, the state of the transaction becomes committing. *Committing* a persistent memory transaction consists of two steps. First, Kiln performs clean-on-commit to flush all the corresponding dirty cache lines remaining in volatile caches. Then, Kiln updates the state of every corresponding NV cache line, from uncommitted to committed. After these two steps are completed, a persistent memory transaction becomes *committed*.

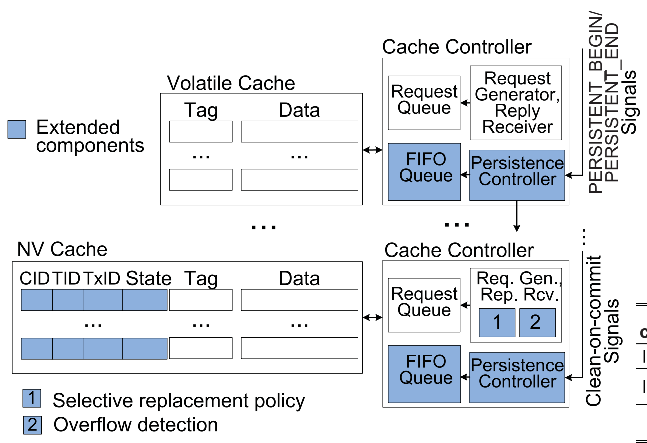
Compared with redo log based persistent memory, Kiln executes faster with both a single persistent memory transaction and a sequence of them. As discussed in Section 3.3, clean-on-commit is much more efficient than executing flush and memory fence instructions. Therefore, Kiln completes a single persistent memory transaction faster than the redo logging method, despite the longer last-level cache (NV cache) access latency. Kiln also executes much faster than redo log based persistent memory when running a sequence of transactions. The redo logging mechanism only flushes log updates when a transaction is committing. The real data updates of a committed transaction can still remain in volatile caches. Therefore, the NV memory needs to keep the log updates after a transaction is committed, until all the real data updates arrive at the NV memory. As a result, a redo log based persistent memory needs to periodically perform a truncation operation, which flushes real data updates from caches to the NV memory and then releases (free of reclamation) the corresponding log entries. Instead, Kiln releases NV memory data blocks right after the corresponding transaction is committed. Therefore, Kiln reduces the total time of completing a sequence of persistent memory transactions by eliminating the truncation operations.

3.5 Discussion

Durable TM transactions: Persistent memory transactions are similar to database and file system transactions, which make atomic, consistent, and durable modifications to the storage system. TM, a concurrency control mechanism which also borrows the concept of "transaction" from the database community for controlling shared memory access, also supports atomic and consistent memory accesses. However, directly enabling durability with TM is suboptimal for persistent memory updates, if not impossible. STM records every speculative store in a log. Therefore, employing STM with durable memory transactions still requires maintenance of a journal. For example, recent studies employing STM for persistent memory updates, including Mnemosyne [45] and NV-heaps [12], both maintain a redo log in the persistent memory. Another type of TM implementation, hardware transactional memory (HTM), does

```
persistent(inorder) {
  read x1, x2, x3;
  do some processing;
  write y1, y2;
}
persistent(inorder) {
  write py1, py2;
}
persistent {
  read x1, x2;
  do some processing;
  write z1, z2;
}
```

(a) Example code.



(b) Cache architecture extensions.

(Bits)	L1	L2	L3
CID	0	0	3
TID	1	1	1
TxID	5	5	8
State	2	2	2
Way	2	3	4
Set	8	9	16
Entries	16	16	128
Total (bytes)	288	320	544

(c) Storage overhead of the FIFO queues.

Miss casued by	Cache set filled by	Detect
In-flight Txi	In-flight Txi	Overflow
In-flight Txi	In-flight Tx	Set timer
Non-persistent	In-flight Tx	Set timer

(d) Detection of NV cache overflow.

Figure 4: Software and architecture extensions developed to facilitate Kiln.

not necessarily require logs. Commodity HTM implementations, such as the transactional synchronization extensions specified by the Intel Haswell processor [24] and the transactional memory processor instructions supported by the IBM zEC12 [25], buffer speculative stores at processors' private caches (in particular, the L1 caches) and overwrite the lower-level caches and memory when transactions commit. These HTM implementations need to support fast recovery from transaction aborts, and therefore ensure atomicity only at higher-level caches. Unless the entire cache hierarchy is made nonvolatile, it is impossible to ensure atomic updates crossing the persistence boundary by directly adopting these HTM implementations. Other HTM implementations, such as IBM Blue Gene/Q's hardware support for TM [46] and LogTM [34], allow the speculative stores to enter the lower-level caches. However, they have other downsides. The IBM Blue Gene/Q [46] requires write-through L1 caches or invalidating the entire L1 cache at the beginning of each transaction. LogTM [34] maintains a hardware-based undo log to buffer the speculative stores. Recovery with the persistent memory from system failures is performed off-line or off the critical path of program execution, and therefore can tolerate much longer recovery latency. Employing durable HTM transactions to update the persistent memory can be unnecessarily cumbersome and inflexible. With Kiln, race-free isolated data accesses in multi-threaded or multi-process programs can be guaranteed by TM or any other concurrency control mechanisms, such as mutexes, semaphores, or lock-free/wait-free data structures and algorithms.

Critical-data persistence vs. whole-system persistence: Kiln supports persistence for user-defined critical data structures typically used in databases or file systems, such as search trees, hash tables, and graphs. This is especially useful for servers running database and file system services. Another research direction focuses on the persistence of the entire system, called whole-system persistence (WSP) [35], supporting instant program restart or resuming after failures. This method makes a persistent copy of the entire memory upon failures, by employing flush-on-fail, i.e., flush all register and cache states to the NV memory. With sufficient backup power sources, a system employing Kiln can also provide high-performance WSP support by mapping all the data to the NV memory address space and performing the same flush-on-fail operation.

4. SOFTWARE INTERFACE AND ARCHITECTURE IMPLEMENTATIONS

In this section, we address the implementation details. First, we provide a software interface for users to define the boundary of a persistent memory transaction. Second, we provide a finite-state machine for every NV cache line to ensure that the persistent memory is in a consistent valid state with only the committed transaction data. Third, we implement a set of cache architecture extensions, including the extended tags and the selective replacement policy at the NV cache, and track logic and FIFO queues in the cache controllers. Fourth, we provide a solution to detect the NV cache overflow and present a fall-back path to resolve the overflow. Finally, we will discuss the physical implementation choices, including the memory technologies used in the NV cache and the NV memory and integration technologies.

4.1 Software Interface and ISA Extension

To define the beginning and end of a persistent memory transaction, we provide the software interface,

`persistent{...}`

to define persistent memory transactions. Furthermore, we provide a software interface that allows the users to declare strong and relaxed ordering control. The strong ordering is denoted by

`#pragma persistence_inorder`

With the strong ordering control declared, Kiln applies clean-on-commit for each persistent memory transaction. Without this declaration, the users can specify the transactions that require ordering with an attribute called *inorder*, i.e., using

`persistent(inorder){...}`

Ordering is maintained within persistent memory transactions with the inorder attribute. Clean-on-commit operations on transactions without this attribute may be delayed. Figure 4(a) shows an example of using the software interface with relaxed ordering control. In this example, the pointers `py1` and `py2` will be updated after the updates to their data objects `y1` and `y2` are flushed to the NV cache. The updates to `z1` and `z2` may remain in volatile caches without being forced to the NV cache.

We also extend the ISA with a pair of new instructions, `PERSISTENT_BEGIN` and `PERSISTENT_END`. The software interface can be translated to ISA instructions with simple

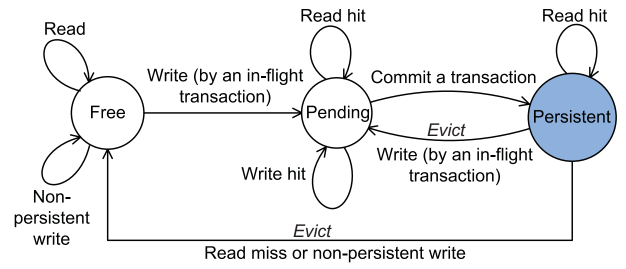

Read
Read hit
Read hit
Write (by an in-flight transaction)
Commit a transaction
Free
Pending
Persistent
Evict
Write (by an in-flight transaction)
Write hit
Non-persistent write
Evict
Read miss or non-persistent write

Figure 5: The state transition of NV cache lines.

modifications to the compiler. Similar ISA and software interface extensions have been implemented to support HTM, such as those of Intel's Haswell processors [24] and IBM's zEC12 [25]. We provide a separate set of extensions with persistent memory transactions so that HTM can be simultaneously used as the concurrency control mechanism.

4.2 Maintaining the State of NV Cache Lines

The NV cache is shared by *non-persistent* cache lines (mapped to the DRAM address space), the cache lines being updated by in-flight persistent transactions, and the cache lines with the committed transactions. Each NV cache line is assigned one of three states: free, pending, and persistent (Figure 5). A *free* cache line stores non-persistent data mapped to the DRAM address space. A *pending* cache line is updated by an in-flight persistent memory transaction, storing the new data value. A cache line with the latest version of a committed persistent memory transaction is called *persistent*. As shown in Figure 5, various access events at a NV cache line can trigger state transitions of the cache line. Note that read or write misses do not apply to a pending cache line, due to our selective replacement policy presented in Section 4.3. Although the state transition can be integrated with a cache coherency protocol, doing this can increase the complexity of maintaining coherence. Therefore, we maintain the state transition separately.

4.3 Cache Extensions

We develop a set of cache architecture extensions (Figure 4(b)) to facilitate Kiln, including additional regions in the NV cache tags, a selective replacement policy, and tracking logic and tables in the cache controllers.

Additional regions in the NV cache tags: We add four additional fields to each cache tag, including the core ID (CID), the hardware thread ID (TID), the persistent memory transaction ID (TxID), and the cache line state. The first three IDs are used to distinguish between different persistent memory transactions initiated by different processor cores. The cache line state is used to maintain the state transition among the states of *free*, *pending*, and *persistent*. The storage overhead of each tag entry is $log_2 N + log_2 T + log_2 M$, plus 2 bits for the cache line state. Here N and T are the number of cores and hardware threads per core, and M is the number of maximum in-flight persistent memory transactions supported by Kiln. If strong ordering is enforced (i.e., `#pragma persistence_inorder` is declared), the number of in-flight persistent memory transactions is limited by the total number of hardware threads, i.e., $N \times T$. The TxID of a persistent memory transaction can be reused after it is committed. We can estimate the storage overhead in the NV cache tags with the following case. If we support 256 in-flight persistent memory transactions on a processor with eight cores and two hardware threads per core, we need an additional 14 bits in each NV cache tag, which only adds 2.7% to a 64-byte cache line. If strong ordering control is enforced, the maximum number of in-flight persistent memory transactions is far less than 256, 16 in this example.

Selective NV cache replacement policy: Existing cache replacement policies are not designed for data persistence. To prevent the in-flight persistent memory transactions from corrupting the data structures stored in the NV memory, we implement a simple selective NV cache replacement policy extension: we do not allow the evictions of pending cache lines. Read and write misses at pending cache lines are thus not allowed in Figure 5. Our extension can work with most existing cache replacement policies. In practice, we adopted LRU as the basic replacement policy in Section 6.We leave the exploration of more sophisticated optimizations of cache replacement policy as future work.

Tracking in-flight persistent memory transactions in cache controllers: We extend cache controllers with FIFO queues and persistence controllers, as illustrated in Figure 4. The FIFO queues are used to track all the dirty cache lines updated by in-flight persistent memory transactions. Each FIFO queue entry is a copy of the extended tag information (CID, TID, and TxID) and the location of a dirty cache line (its set and way number). We evaluated the storage overhead of FIFO queues in a cache hierarchy described in Table 2 (choose option (b) for L3 cache). We employ the number of FIFO entries that is sufficient to accommodate the workloads described in Table 3: the FIFO queues at each L1 and L2 cache have 16 entries; the one at the L3 cache has 128 entries. Figure 4(c) lists the storage overhead of the FIFO queues at each L1, L2, and L3 cache. Note that the storage device in cache controllers is volatile for fast access and easy fabrication. The information stored in the FIFO queues will be lost if the processor loses power. In this case, all the in-flight persistent memory transactions need to be re-executed after the system restarts. Persistence controllers are in charge of enqueuing the FIFO and issuing the clean-on-commit operations. They also allocate TxIDs to the new persistent memory transactions. The persistence controller at the L1 cache controllers are extended to detect the boundary of each persistent memory transaction, by receiving the `PERSISTENT_BEGIN` and `PERSISTENT_END` signals from the processor cores. The request generator in the NV cache controller is extended to implement the selective replacement policy and the overflow detection mechanisms.

4.4 NV Cache Overflow and Fall-back Path

NV cache overflow is the case when a miss at the NV cache can never be serviced because no victim can be found for replacement. In this case, the program cannot make forward progress without the NV cache overflow being resolved. Because we do not allow pending cache lines to be evicted from the NV cache, the overflow may be caused by one of two reasons: (1) the *capacity* is smaller than the total size of in-flight persistent memory transactions or (2) the *associativity* is insufficient to accommodate all in-flight persistent memory transactions that conflict at the same cache set.

Detecting NV cache overflow: We can detect an NV cache overflow when searching for an eviction victim at the NV cache. Figure 4(d) lists the scenarios which can lead to

Table 2: Parameters of the evaluated multi-core system.

Processor/Technology	Intel Core i7 like/22 nm
Cores	8 (2.5GHz), 16 threads
L1 Cache (Private)	Volatile (SRAM), 64KB, 4-way, 64B blocks, 1.6ns latency
L2 Cache (Private)	Volatile (SRAM), 256KB, 8-way, 64B blocks, 4.4ns latency
L3 Cache (Shared)	(a) Volatile (SRAM), 16MB, 16-way, 64B blocks, 10ns latency
	(b) Nonvolatile (STT-MRAM), 64MB, 16-way, 64B blocks, 15ns (19ns) read (write) latency
Memory Controller	Two dual-channel memory controllers, FR-FCFS
Memory Technology	30 nm
DRAM DIMM	DDR4-2133, 2GB
NV Memory DIMM	STT-MRAM, 2GB, 25ns row-hit latency, 65ns (76ns) read (write) row-conflict latency
Power and Energy	Processor (with L1 and L2): 149W (peak).
	L3 (SRAM): read/write: 0.58nJ/access; L3 (STT-MRAM): read (write): 0.61 (0.67) nJ/access.
	NV memory : row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 1.00 (2.89) pJ/bit

NV cache overflows. NV cache overflows are hard to detect if the cache set is filled by a mix of different in-flight persistent memory transactions. It is possible that the program can continue to make progress after one of the in-flight persistent memory transactions is committed and advance one of the cache lines in the set to the *persistent* state. Unfortunately, simply waiting for next available victim will incur performance overhead and even deadlocks. Instead, we stall memory requests when the request queue at the higher level cache is almost full (e.g., 80% filled) and then provide a fall-back path.

Fall-back path: We provide a fall-back path to resolve the issue of NV cache overflows, allowing the pending cache lines to be written back to the NV memory and maintain multiversioning in the NV memory with *hardware-controlled COW* similar to that used in eNVy [48]. When an NV cache overflow is detected, Kiln will notify the operating system by interrupt to allocate new pages to buffer the pending cache lines evicted from the NV cache. A mapping table will be created in the NV memory and updated with the physical addresses of buffered pending cache lines. When a persistent memory transaction is committed, the page table will be updated to invalidate the old data values and enable the new data values according to the mapping table. Then the corresponding mapping table entries can be discarded.

Commodity processors typically employ several megabytes of last-level cache with high associativity (e.g., 16-way). The density of NVRAM is much higher than SRAM, so the capacity of the NV cache can be as large as tens of or over one hundred megabytes. The associativity of the NV cache can also be higher than SRAM-based caches. Therefore, Kiln can support in-flight persistent transactions with memory footprints up to tens of megabytes. The memory footprints of the in-flight persistent memory transactions are determined by the granularity of modifications performed to the persistent data structures and upper-bounded by the size of data structure elements (e.g., tree nodes, table entries, graph edges, etc.). Furthermore, small-granularity data updates may dominate some commercial and future real-world workloads. For example, several key-value workload characteristics published recently by Facebook [5] showed that most queries employ keys of less than 32 bytes and values of no more than a few hundred bytes. For this type of workload, NV cache overflow will be less of an issue.

4.5 Recovery

Kiln allows easy and fast system recovery mechanisms, because most of the persistent updates are applied in-place to the real in-memory data structures. Upon restart from an abnormal termination, the system can go through the following steps for recovery. First, we scan the NV cache tags and invalidate the cache lines in the pending state because they are partially updated data structures in process by in-flight persistent memory transactions before failure. Next, we scan the page table in the NV memory to identify the temporary data copies (if any) due to NV cache overflows. These data copies were updated by in-flight memory transactions as well, and hence can be invalidated. These recovery steps can be performed by hardware, reusing the tracking logic and FIFO queues in cache controllers.

Table 3: Benchmarks used in our experiments.

Benchmarks	Description
BTree [7]	Inserts/deletes nodes in a B-tree.
Hash [12]	Inserts/deletes entries in a hash table.
RBTree [12]	Inserts/deletes nodes in a red-black tree.
SDG [41]	Inserts/deletes edges in a scalable large graph.
SPS [12]	Random swaps between entries in an array.
SSCA2 [6]	A scalable large graph analysis benchmark.

4.6 Physical Implementation

In principle, our persistent memory architecture design does not rely on any specific physical implementation of processors and memories. For example, all components of the processor and memory can be packaged in a single package with silicon interposer technology, which has been widely explored by academia and industry to develop high-performance system-in-package designs [16, 17]. The NV cache and the NV memory can both be implemented by STT-MRAM, which provides the best latency and endurance among NVRAM technologies. Everspin [26] recently launched the DDR3 compatible STT-MRAM components, which is projected to be able to scale to Gb densities (close to NAND flash). Existing work has demonstrated the feasibility of STT-MRAM used in lower-level caches [49] in multi-core processors. The NV cache can be stacked on top of the CPU die for large capacity and high bandwidth, or packaged with the NV memory, sitting beside the processor with higher-level caches. In this case, the processor can be fabricated without the effort of integrating different memory technologies. We can also implement the NV memory with resistive RAM (ReRAM) or PCM, because they are byte-addressable and nonvolatile just like STT-MRAM. The main memory, including the NV memory and DRAMs, can be implemented with an off-chip DIMM interface or wide I/O interface [27, 37]. The wide I/O implementation can achieve higher memory bandwidth

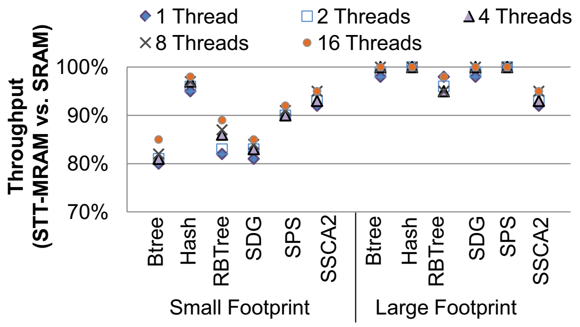


Figure 6: Performance comparison between two native systems adopting STT-MRAM and SRAM as L3 cache respectively. Results show that the two systems have similar performance.

between the processor and the main memory for better performance, however it incurs complexity and higher cost.

5. EXPERIMENTAL SETUP

We evaluated the performance and power of our persistent memory design on a multi-core system. In this section, we describe our simulation framework, processor and memory configurations, and benchmarks.

5.1 Simulation Framework

Our experiments are conducted using McSim [3], a Pin [32]-based multi- and many-core cycle-accurate simulation infrastructure. McSim models out-of-order cores, caches, directories, on-chip networks, and memory channels. Table 2 lists the detailed parameters and architecture configurations of the processor and memory system in our simulation. The multi-core processor consists of eight out-of-order cores, each of which is similar to one of the Intel Core i7 cores [1]. Each processor core incorporates SRAM-based volatile private L1 and L2 caches. Kiln employs an STT-MRAM based L3 cache (the NV cache) (option (b) in Table 2). Option (a) in Table 2 lists the parameters of a system with SRAM as L3 cache, which is used to validate the performance of Option (b). Note that the parameters of the two systems are calculated based on the same silicon area, i.e., a 16MB SRAM-based cache occupies the same silicon area of 64MB STT-MRAM based cache. Both L3 caches are 16-way set-associative and multi-banked. The processor cores and L3 cache banks communicate with each other through a crossbar interconnect. A two-level hierarchical directory-based MESI protocol is employed to maintain cache coherence at the private caches and the L3 cache. The DRAM and the NV memory are modeled as off-chip DIMMs. Memory requests to DRAM and the NV memory are managed by two dual-channel memory controllers. The timing and energy parameters of the NV cache and NV memory are calculated with NVSim [15], a performance, power, and area estimation tool for NVRAM.

Our simulation framework models Kiln's in-place updates, clean-on-commit functionality, and architecture extensions. We also model HTM based on Hammond et al.'s work [21] as one of our two concurrency control mechanisms used in the experiments. The implementations of most commodity HTM, e.g., Intel's Haswell processor [24] and IBM's zEC12 processor [25], are similar to Hammond et al.'s work. The memory footprint of transactions is limited up to the capacity of private caches. Overflow at the private caches will result in transaction abort (re-execution) or transferring the control to software.

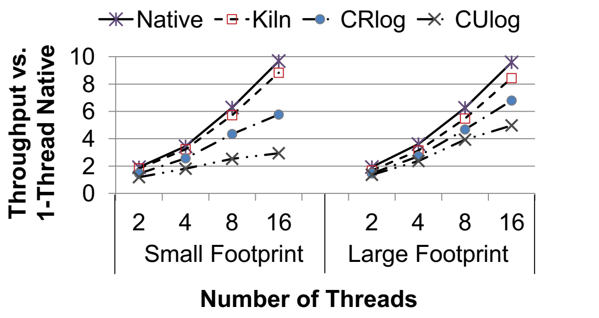


Figure 8: Performance gap vs. number of threads.

5.2 Benchmarks

The persistence interface of most existing software applications are optimized for accesses to disk-based storage devices. Currently, no existing public benchmark suites can be used to evaluate the Kiln design. Therefore, we constructed a set of benchmarks as described in Table 3. The data structures and functionality of these benchmarks are similar to those in the benchmark suite used by NV-heaps [12]. The benchmarks perform search, insert, and delete to data structures used in databases and file systems, including a search tree, hash table, sparse graph, and array. Two sets of experiments are conducted to insert and delete the data elements (tree nodes, table entries, graph edges, etc.) with small (512 bytes) and large (512 kilobytes) granularity, respectively. They will be referred to as workloads of small and large footprints in the rest of the paper. Each persistent memory transaction inserts or deletes a single data element. The benchmarks are written with the strong ordering control interface (Section 4) to force all the transactions to commit inorder. HTM is used as the concurrency control mechanism for workloads of small footprint, while mutex lock is used for workloads of large footprint. We also implemented another version of the benchmarks, which perform undo and redo logging at word granularity to provide persistence support. We only evaluate the hardware performance of various persistent memory designs, so we do not count the latency of executing the logging instructions. We collect the performance and power results of the running phase of the benchmarks, skipping the initialization phase.

6. RESULTS

In this section, we present the evaluation results and analyze the reasons for these results.

6.1 Volatile Vs. Nonvolatile Last-level Cache

We first compare throughput (in terms of the executed insert/delete operations per second) of two systems with the L3 cache implemented by SRAM and STT-MRAM, without providing persistence support (Figure 6). Despite its lower latency, the SRAM-based last-level cache is only a quarter the capacity of STT-MRAM based cache on the same silicon area. Our results show that using an STT-MRAM based L3 cache can achieve on average 91% and 99% of the performance using SRAM-based L3 cache on workloads of small and large footprints, respectively. These results show that employing NV cache in a non-persistent manner as the last-level cache does not remarkably change the system

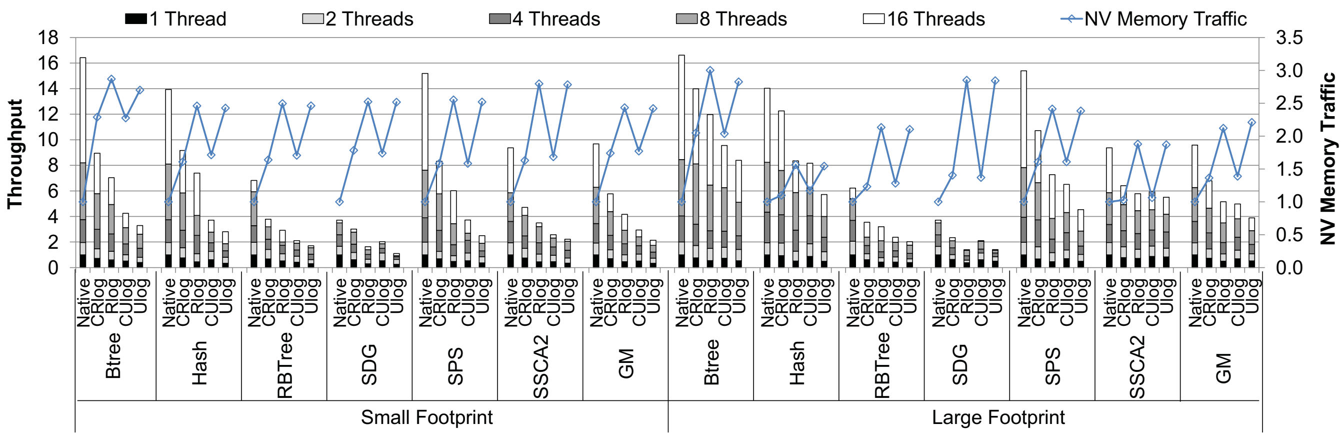


Figure 7: Performance of systems that adopt a NV L3 cache, but with logging for atomicity and flush and memory fence for ordering. We evaluate the throughput (bars) and NV memory traffic (broken lines). All the throughputs are normalized against the native system running 1 thread. For NV memory traffic, we only show the normalized results running 16 threads.

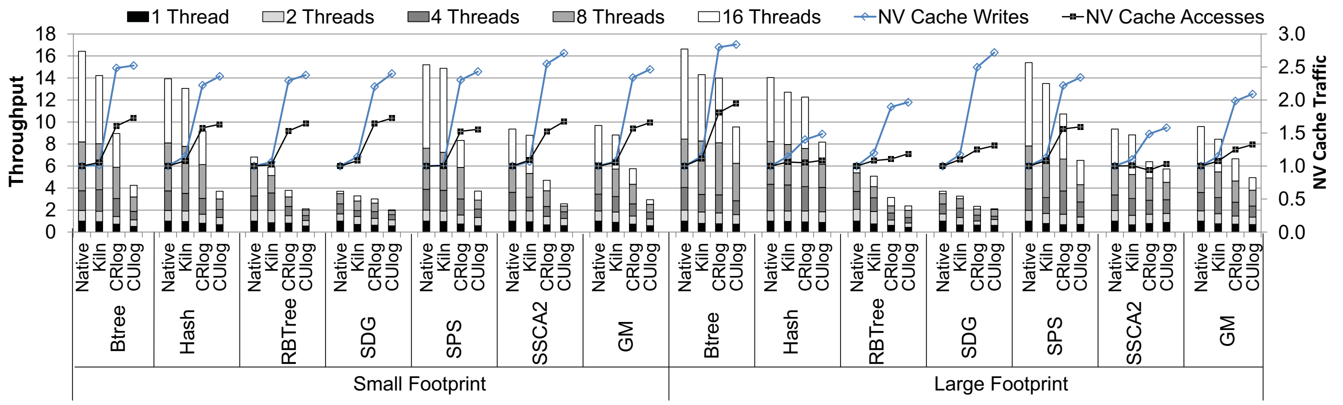


Figure 9: The throughput (bars) and NV cache traffic (broken lines) of Kiln. All the throughputs are normalized against the native system running 1 thread. For NV cache traffic, we only show the normalized results running 16 threads.

performance due to the latency and capacity trade-offs of SRAM and STT-MRAM technologies. In the following experiments, we use the configuration of STT-MRAM based L3 cache as the baseline native system.

6.2 Log-based Persistent Memory Performance

A log-based persistent memory system can adopt a NV L3 cache, with logging to ensure atomicity and flush and memory fence to ensure ordering. In this system, the logs become persistent once they arrive at the NV cache. We want to demonstrate that the performance of such an optimized log-based system is not scalable as the number of threads increases.

A log-based system can adopt two types of logs, redo and undo logs. We denote the resultant systems as CRlog and CUlog, respectively. Rlog and Ulog denote the systems where the logs are only stored in the NV memory. CXlog uses Kiln's cache controller extensions to track the dirty cache lines of logs and flush them into the NV cache. Xlog uses `clflush` and `mfence` to write logs in to the NV memory. However, the latency of executing these two instructions is not counted as discussed in Section 5. Figure 7 shows the comparisons between CXlog (CRlog and CUlog) and Xlog (Rlog and Ulog) for throughput of insert/delete operations and NV memory traffic. The results show that the throughput of CRlog and CUlog increases by an average of 38% and 33% compared with Rlog and Ulog, with workloads of small footprints running 16 threads. The corresponding NV memory traffic is reduced by 28% (CRlog) and 26% (CUlog) on average. With workloads of large footprints, the average improvement of throughput is 31% (CRlog) and 28% (CUlog). The corresponding NV memory traffic reductions are 35% and 37%.

While CXlog significantly reduces the number of accesses to the NV memory, it can still incur an over 50% increase in the memory traffic compared with the native system (denoted as the Native). In addition, the throughput of CXlog does not scale well when the number of threads increases from two to 16 (Figure 8). With two threads, the performance gap between CUlog and Native is less than 38% and 30% with small and large footprints, respectively. However, when the number of threads increases to 16, this performance gap also significantly increases up to 70% and 50% with small and large footprints, respectively. CRlog performs better than CUlog, however, the performance gap still increases from around 25% to 45% when the number of threads increases from two to 16. With a large number of threads running concurrently, the log size grows quickly and

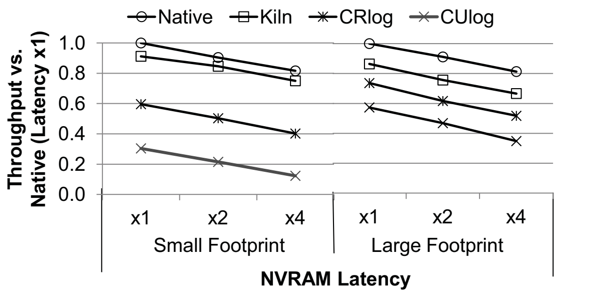


Figure 10: Throughput of insert/delete operations of 16-thread workloads with longer NVRAM latencies, normalized to the Native throughput with $\times 1$ latency and 16 threads.

the NV cache will soon be filled by logs. Furthermore, the logs in the NV cache, which will not be reused anymore, can also lead to early evictions of reusable cache lines of the real data structures. Sophisticated replacement policies can be employed to prioritize the evictions of logs. However, this will be equivalent to bypassing the NV cache or flushing the logs all the way down to the NV memory.

6.3 Performance of Kiln

The following experiments evaluate Kiln performance in terms of the throughput of insert/delete operations.

Throughput and NV cache traffic: For workloads of small and large footprints running 16 threads, Kiln achieves on average 91% and 88% of the throughput of the Native system (Figure 9). Therefore, the performance of Kiln is 1.6× and 3× of that of CRlog and CUlog with workloads of small footprints, and 1.2× and 1.5× of that of CRlog and CUlog with workloads of large footprints. Kiln performs worse for workloads of large memory footprints because the large number of pending cache lines (not allowed to be evicted to the NV memory) leads to early evictions of other reusable cache lines. Although CRlog allows the persistent data to be updated immediately after the logs reach the NV cache, it still does not perform as well as Kiln because CRlog needs to maintain the ordering of the log updates with `clfush` and `mfence`, which prevent the cache controllers from re-ordering the memory requests and block subsequent loads and stores. While log-based persistent memory designs double the write traffic to the NV cache, Kiln only generates 8% additional writes and 5% additional total accesses in NV cache traffic compared to the Native, due to clean-on-commit operations.

Sensitivity to NVRAM latency: The evaluations above are conducted with fixed NV cache and the NV memory latencies. We also evaluated the performance variation with longer NVRAM latencies. Figure 10 shows the results of normalized throughput with doubling and quadrupling the original NV cache and NV memory latencies (the NV memory clock rate is determined accordingly), averaged across the benchmarks running 16 threads. We observe that the benefit of Kiln remains at longer NVRAM latencies for workloads of both small and large footprints. Even with quadrupled NVRAM latency, Kiln still achieves 92% and 82% of Native throughput with workloads of small and large footprints.

Frequency of NV cache overflow: The frequency of NV cache overflow significantly affect system performance. Here we study the frequency of NV cache overflow by further increasing the memory footprints of the persistent memory transactions. We count the number of NV cache overflows during 100K persistent memory transactions inserting and deleting to a hash table. The keys are four-byte integers. The value size ranges from 512KB to 64MB. Each persistent memory transaction inserts or deletes one entry of the hash table. When running a single thread, we do not observe any NV cache overflows even with the value size increased to 32MB. With multithreaded workloads, the frequency of NV cache overflow is lower than 0.1% (100 overflow events out of the total 100k transactions) when the total memory footprint of all the threads is smaller than 64MB. Unfortunately, the frequency reaches 100% when the total memory footprint of all the concurrent transactions is larger than the NV cache capacity. In such a case, Kiln falls back to hardware-controlled COW as described in Section 4.4, and the performance is similar to that of CRlog. We will leave the investigation of more efficient methods to resolve the overflow issue as future work.

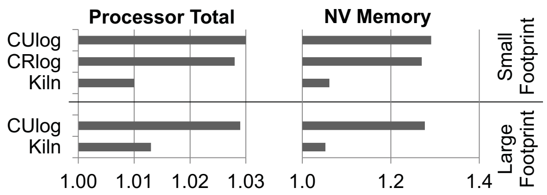


Figure 11: The average dynamic power consumption of processor (including the NV cache) and the NV memory, normalized to the Native (workloads running 16 threads).

6.4 Dynamic Power

Maintaining data persistence with Kiln incurs additional processor and memory dynamic power consumption due to the extra bookkeeping activities in cache controllers and the increased accesses to caches and the NV memory. We calculated the processor's dynamic power consumption by feeding the simulation statistics of processor and cache activities into McPAT [30]. We calculated the NV memory power consumption based on the number of memory accesses broken down into row buffer hits and misses, the memory energy configuration listed in Table 2, and the total execution time of each benchmark. As shown in Figure 11, Kiln provides up to a 23% dynamic power reduction for the NV memory compared to CXlog due to fewer memory accesses (Figure 9). Compared to the Native, Kiln results in dynamic power overheads of only 1.2% and 5% to the processor and the NV memory.

7. CONCLUSIONS

NVRAM technologies can provide promising solutions to persistent memory design. However, current NVRAM-based persistent memory designs are inefficient due to increased latency and bandwidth demands due to log-based or COW mechanisms. In this paper, we propose Kiln, a persistent memory design which employs a multiversioned memory hierarchy consisting of an NV cache and NV memory, enabling in-place updates to in-memory data structures, without the redundant writes required by logging or COW. Kiln provides persistence support with only a 9% performance overhead to the native system, hence up to 2× performance improvement to the log-based NVRAM persistent memory. In addition, Kiln provides a simple and intuitive software interface, as well as easy and fast recovery from failures. Our work rethinks the design of persistent memory in light of emerging NVRAM technologies, which is a critical step in reaping the full advantages of NVRAM technologies beyond simply replacing of DRAM in main memory.

8. ACKNOWLEDGMENTS

We thank Hans J. Boehm, Dhruva Chakrabarti, Jichuan Chang, Terence Kelly, Parthasarathy Ranganathan, Joseph Tucek, Alistair Veitch, and the anonymous reviewers for their valuable feedback. We gratefully acknowledge Onur Mutlu for many insightful discussions on this work. We thank Joel Coburn for providing help for some of the benchmarks used in this paper. Jishen Zhao was supported in part by NSF 1218867, 1213052, and 0903432 while conducting this work. This research was also supported by the Department of Energy under Award Number DE - SC0005026.

9. REFERENCES

[1] Intel Core i7, http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[2] Java persistence API, http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html.

[3] J. H. Ahn, S. Li, S. O, and N. P. Jouppi. McSimA+: a manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *ISPASS*, 2013.

[4] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA*, 1987.

[5] B. Atikoglu et al. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[6] D. A. Bader et al. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *HiPC*, 2005.

[7] T. Bingmann. STX B+ Tree, Sept. 2008, http://idlebox.net/2007/stx-btree/.

[8] T. C. Bressoud, T. Clark, and T. Kan. The design and use of persistent memory on the dncp hardware fault-tolerant platform. In *DSN*, 2001.

[9] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *ACM Commun*, 1991.

[10] A. M. Caulfield et al. Providing safe, user space access to fast, solid state disks. In *ASPLOS*, 2012.

[11] P. M. Chen et al. The rio file cache: surviving operating system crashes. In *ASPLOS*, 1996.

[12] J. Coburn et al. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[13] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[14] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *VLDB*, 1989.

[15] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 2012.

[16] X. Dong et al. Simple but effective heterogeneous main memory with on-chip memory controller support. In *SC*, 2010.

[17] P. Dorsey. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. In *Xilinx White Papers*, 2010.

[18] F. Eskesen et al. Software exploitation of a fault-tolerant computer with a large memory. In *FTCS*, 1998.

[19] R. G. Gattell. Object data management: object-oriented and extended. 1994.

[20] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *SOSP*, 1987.

[21] L. Hammond et al. Transactional memory coherence and consistency. In *ISCA*, 2004.

[22] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[23] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Technical Conference*, 1994.

[24] Intel Corporation. Intel architecture instruction set extensions programming reference, 319433-012 edition. 2012.

[25] C. Jacobi et al. Transactional memory architecture and implementation for IBM System Z. In *MICRO*, 2012.

[26] J. Janesky. Device performance in a fully functional 800MHz DDR3 Spin Torque Magnetic Random Access Memory. In *IMW*, 2013.

[27] J.-S. Kim et al. A 1.2 V 12.8 GB/s 2 Gb mobile wide-I/O DRAM with 4x 128 I/Os using TSV based stacking. *JSSC*, 2012.

[28] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *ACM Commun*, 1991.

[29] E. Lee et al. Unioning of the buffer cache and journaling layers with non-volatile memory. In *FAST*, 2013.

[30] S. Li et al. McPAT: an integrated power,area,and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[31] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.

[32] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[33] A. Marquez et al. Fast portable orthogonally persistent Java. *Softw. Pract. Exper.*, 2000.

[34] K. E. Moore et al. LogTM: log-based transactional memory. In *HPCA*, 2006.

[35] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[36] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): a simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys*, 2013.

[37] J. Pawlowski. Hybrid memory cube. In *Hot Chips*, 2011.

[38] R. Ramakrishnan and J. Gehrke. Database management systems, third edition. 2007.

[39] M. Satyanarayanan et al. Lightweight recoverable virtual memory. In *SOSP*, 1993.

[40] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI*, 2006.

[41] J. Siek et al. Boost: adjacency list, ver. 1.52.0, http://www.boost.org/doc/libs/.

[42] V. Singhal, V. Kakkad, and P. R. Wilson. Texas: an efficient, portable persistent store. In *POS*, 1992.

[43] S. C. Tweedie. Journaling the Linux ext2fs sile system. In *Linux Expo*, 1987.

[44] S. Venkataraman et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[45] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, 2011.

[46] A. Wang et al. Evaluation of Blue Gene/Q hardware support for transactional memories. In *PACT*, 2012.

[47] S. J. White and D. J. DeWitt. Quickstore: a high performance mapped object store. In *SIGMOD*, 1994.

[48] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS*, 1994.

[49] X. Wu et al. Hybrid cache architecture with disparate memory technologies. In *ISCA*, 2009.