Understanding and Improving Persistent Transactions on Optane$^{\text{TM}}$ DC Memory

1st Pantea Zardoshti
Lehigh University
USA
zardoshti@lehigh.edu

2nd Michael Spear
Lehigh University
USA
spear@lehigh.edu

3rd Aida Vosoughi
Oracle Corp.
USA
aida.vosoughi@oracle.com

4th Garret Swart
Oracle Corp.
USA
garret.swart@oracle.com

***Abstract*—Storing data structures in high-capacity byte-addressable persistent memory instead of DRAM or a storage device offers the opportunity to (1) reduce cost and power consumption compared with DRAM, (2) decrease the latency and CPU resources needed for an I/O operation compared with storage, and (3) allow for fast recovery as the data structure remains in memory after a machine failure. The first commercial offering in this space is Intel® Optane$^{\text{TM}}$ Direct Connect (Optane$^{\text{TM}}$ DC) Persistent Memory. Optane$^{\text{TM}}$ DC promises access time within a constant factor of DRAM, with larger capacity, lower energy consumption, and persistence. We present an experimental evaluation of persistent transactional memory performance, and explore how Optane$^{\text{TM}}$ DC durability domains affect the overall results. Given that neither of the two available durability domains can deliver performance competitive with DRAM, we introduce and emulate a new durability domain, called PDRAM, in which the memory controller tracks enough information (and has enough reserve power) to make DRAM behave like a persistent cache of Optane$^{\text{TM}}$ DC memory.**

In this paper we compare the performance of these durability domains on several configurations of five persistent transactional memory applications. We find a large throughput difference, which emphasizes the importance of choosing the best durability domain for each application and system. At the same time, our results confirm that recently published persistent transactional memory algorithms are able to scale, and that recent optimizations for these algorithms lead to strong performance, with speedups as high as 6× at 16 threads.

***Index Terms*—Persistent Memory, Non-Volatile Memory, Transactional Memory, Storage, Concurrency, Optane$^{\text{TM}}$**

I. Introduction

In 2019, Intel® Optane$^{\text{TM}}$ Direct Connect Persistent Memory (Optane$^{\text{TM}}$ DC) became commercially available. Optane$^{\text{TM}}$ DC creates many new opportunities for system designers and programmers. At the simplest level, Optane$^{\text{TM}}$ DC can be thought of as a DRAM alternative that has higher density and lower power consumption, albeit at the cost of higher latency and lower throughput. More exciting is that Optane$^{\text{TM}}$ DC memory can be *persistent*: it can retain its contents for extended periods of time, without requiring any energy to do so. This means, for example, that Optane$^{\text{TM}}$ DC can be a new layer in the storage hierarchy [1], or even replace conventional disk and SSD devices when high performance is paramount. The impact of such a transition on software will be profound, as it would mean that the entire memory hierarchy would become byte-addressable, and persistence features would become available to programmers without the need for system calls.

Optane$^{\text{TM}}$ DC is one of many technologies for byte-addressable persistent memory (also know as non-volatile memory, or NVM). Past notable works include phase change memory (PCM) [2], [3], STT-MRAM [4]–[6], and resistive RAM (ReRAM) [7], [8]. Programmers seeking to exploit any of these technologies have traditionally faced a tradeoff between performance and ease of use. The easiest approach is to request that the operating system (OS) treat the NVM as a storage device. In this case, the OS will create a filesystem atop the NVM, and programs can load and store files from that filesystem, instead of an SSD or disk [9]. With this approach, the latency of the storage device itself is orders of magnitude faster than SSD, and the program does not require changes in order to use the NVM. However, potential performance gains are lost: interactions with the NVM require system calls, and the programmer must provide code to serialize and de-serialize data when interacting with files.

The extreme alternative is for programmers to create hand-crafted algorithms and data structures that operate directly upon an NVM region that is mapped into a process's address space. E.g., a program might use the Linux DAX filesystem to directly map a file from NVM into its virtual address space. It could then operate on the addresses within the mapped region, which would directly modify the persistent representation of data. The programmer must ensure that this code is resilient to failure at any point in its execution. Typically this is achieved through careful management of special persistence-oriented assembly instructions that allow a program to guarantee a correctness criteria like *linearizable durability* [10]. Unfortunately, even simple persistent data structures are considered publishable research results [11]–[13].

In between these two points is the idea of persistent language-level transactions [14]–[24] and persistent critical sections [25]. In these approaches, hereafter referred to as persistent transactional memory (PTM), programmers map a file from NVM into the virtual address space. However, they then identify the regions of their program that might access these nonvolatile regions, by marking lexically scoped transactions. The compiler instruments the loads and stores within these regions, so that each load and store is performed by a run-time library. The library typically uses undo or redo logging so that transactions can appear to execute atomically. With redo logging, the writes of a transaction are kept in a private, persistent "redo" log until commit time, and program data is only updated after the transaction reaches its commit point. With undo logging, the writes of a transaction are performed during transaction execution, but the old values are kept in a private, persistent "undo" log that can be used to restore program state in the event that the transaction aborts. With either method, (1) if a failure happens before a transaction finishes, it can be rolled back; and (2) if a failure happens after a transaction finishes, all of its changes are guaranteed to be in persistent memory. In addition to logging, the current best-performing PTM algorithms use a table of versioned locks to coordinate the speculative memory accesses of locations by concurrent threads, using techniques from software transactional memory (STM) [26], [27].

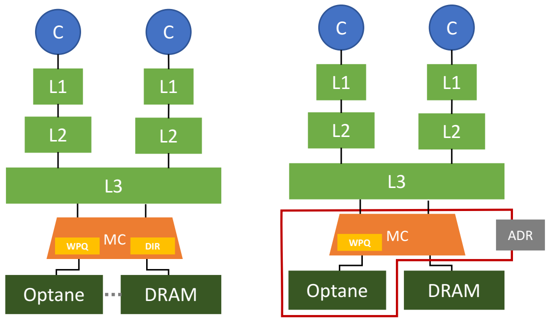
In this paper, we focus on PTM performance on Optane^{TM} DC systems. Whereas most past PTM research has either simulated NVM performance or assumed that DRAM performance is an adequate proxy for NVM performance, we present PTM results on a real Optane^{TM} DC system. In Section III we experimentally demonstrate that Optane^{TM} DC performance is not predicted well by DRAM: not only are Optane^{TM} DC latencies higher than DRAM, but the nature of transactional execution leads to worse scalability for transactions on Optane^{TM} DC than transactions on DRAM. This finding considers two different models of hardware support for durability, described in Section II. We then propose and evaluate two new hardware durability models in Section IV. Our new models re-purpose existing features of Optane^{TM} DC systems to let DRAM serve as a persistent cache of the NVM. This evaluation informs our conclusions in Section V.

II. Background

Figures 1 and 2 present a depiction of an x86 system outfitted with Optane^{TM} DCmemory. The letter "C" represents a core, the L3 cache is shared among cores, and the L1 and L2 caches are shared among the hyperthreads of a core. The memory controller (MC) is able to interact with both the Optane^{TM} DCmemory and DRAM. Stores to the Optane^{TM} modules must pass through the Write Pending Queue (WPQ) within the memory controller.

A. *Optane^{TM} Operating Modes*

Current Optane^{TM} DC©-based systems can operate in two modes, both of which retain some traditional DRAM in addition to Optane^{TM} DCmemory. The first mode is "Memory Mode", depicted in Figure 1(a). Memory Mode treats DRAM like a cache of the Optane^{TM} DCmemory, and disregards persistence. This is represented by the gray line between the DRAM and Optane^{TM} modules: in Memory Mode, the system operates as if there was a memory hierarchy in which DRAM sat between the L3 and the Optane^{TM} memory, and data moved across the gray line. In contrast, "AppDirect Mode" (Figure 1(b)) treats the Optane^{TM} DC and DRAM as separate memories. The red box indicates that both the Optane^{TM}



(a) Memory Mode (b) AppDirect Mode

Fig. 1: Optane^{TM} Operating Modes

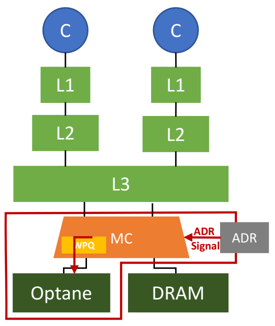
memory and the memory controller are persistent: once a store reaches the boundary of the Asynchronous DRAM Refresh (ADR), there is sufficient reserve power to guarantee that the store will pass through the WPQ to the Optane^{TM} memory and be written, even if the system experiences a power failure.

To achieve the illusion of DRAM caching pages of Optane^{TM} DC© memory, in Memory Mode the on-CPU memory controller maintains a table (DIR) that remaps physical addresses in the DRAM to physical addresses in the NVM. When a page of NVM is listed in the table, loads and stores route to DRAM instead. From a programmer's perspective, this gives the illusion of a substantially larger memory than is possible using only DRAM, which runs at roughly the speed of DRAM, but which is not persistent. The memory controller is responsible for implementing optimizations, such as prefetching and asynchronous writeback, to hide the higher latency of the Optane^{TM} DCmemory. While low-level characteristics of the Optane^{TM} DCmemory imply that data written in Memory Mode retains its value upon power failure, contents are encrypted/decrypted using a unique key that is regenerated upon reboot. Thus upon system restart, the contents of Optane^{TM} DCmemory in Memory Mode are effectively reset to random.

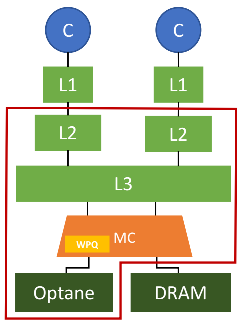
In AppDirect Mode, the OS and applications are aware that physical pages of DRAM and Optane^{TM} memory are disjoint. By mapping these physical pages into different regions of virtual memory, a program can, by way of regular loads and stores, explicitly persist program data to the Optane^{TM} DCmemory. This complicates the programming model, by requiring the programmer to partition data into volatile and nonvolatile spaces. In addition to persistence, important factors include memory access latency and data structure size.

B. *Optane^{TM} Persistence Domains*

To benefit from persistence, it is not enough to simply run an application in AppDirect Mode, because some parts of the system are not persistent. Clearly the Optane^{TM} DIMMs themselves are persistent, and any store that is acknowledged by Optane^{TM} will not be lost on power failure. On the other hand, L1 caches are currently not persistent, and thus it is not enough for a program to issue a store to a virtual address that



(a) ADR



(b) eADR

Fig. 2: Optane^{TM} durability domains. ADR appears on the left, and eADR on the right. With ADR, only stores that reach the memory controller's queues are guaranteed to persist. With eADR, stores to addresses in the Optane^{TM} will be flushed from the cache on any system failure, and will persist.

maps to Optane^{TM} DCmemory: that store may idle at any level of the memory hierarchy due to caching policies. Note, too, that while the CPU may execute stores in one order, they may be written back to the Optane^{TM} in another order. This raises challenges analogous to processor memory consistency [28].

Systems vary in terms of which of their components are persistent, i.e., which are part of the "Durability Domain" [29]. Figure 2 presents the two available durability domains for Optane^{TM} DCsystems. In the figure, components within the red box are considered to be durable. We do not show the simplest domain, "No Power Reserve", as it has been deprecated [30]. In that domain, only the Optane^{TM} DCDIMMs themselves were durable, and programs had to ensure that stores reached the Optane^{TM} if they were to be persisted. This proved to be too cumbersome and slow [31], [32].

The domain in Figure 2(a) introduces a small amount of reserve power. This power is sufficient to flush the memory controller's write queues even when the system loses power. The Intel Asynchronous DRAM Refresh (ADR) provides this guarantee [31]. To ensure that a write to cache line `X` is seen during an ADR, a programmer issues the `clwb X` instruction to flush the data back to the memory controller. Of course, subsequent loads and stores can be reordered with respect to the `clwb`. To order two persistent flushes (e.g., the initialization of data and the setting of a flag to indicate that the initialization is complete), a program must perform the first store and `clwb` it, then issue a store fence (`sfence`) [33], and then perform the second store and `clwb` it. The overhead of these flush and fence instructions can be reduced through a transactional programming interface.

The final domain, extended ADR ("eADR") provides more power reserve than ADR. In addition to providing enough power to flush the WPQ, there is also enough reserve power to allow the system to execute instructions that cause all of the data in the caches to be flushed to the Optane^{TM} DIMMs. It is easiest to imagine that this reserve power is an auxiliary battery that is employed upon power failure to gracefully shut down the system [34]–[37]. With eADR, it is generally not necessary for programs to explicitly execute `clwb` and fence instructions. However, the OS must be able to handle a power-fail signal by flushing caches and queues before the reserve energy is depleted. The OS must also be able to detect if it the reserve was insufficient, and reliably report the failure to the application.

Note that in ADR, a store may become visible to other cores (via the L3) before it has persisted. In eADR, a store becomes persistent and durable when it reaches the L2. This has surprising consequences for programmers. As an example, with ADR, programmers cannot use Intel's Transactional Synchronization eXtensions (TSX): `clwb` causes a store to leave the L1, which also causes the transaction to abort. In contrast, with eADR programmers can use TSX: when the transaction commits, its changes become visible to other threads, and simultaneously they cross into the durability domain.

III. PTM Performance on Optane^{TM}

Whereas past work would either simulate NVM, or else assume that DRAM latencies were a reasonable proxy for NVM, the availability of Optane^{TM} DC systems allows experimentation that reveals the true latencies and bottlenecks. In this section, we focus on two questions. The first is quite simply "How effectively do measurements on DRAM systems predict performance on Optane^{TM} DC systems?" The second question is "What is the performance impact of providing enough reserve power to operate in the eADR durability domain?" An especially important aspect of this latter question is that past work has studied persistent versions of various STM algorithms [38], and concluded that explicit fences and flushes favor certain algorithms over others. It is important to know whether these findings also hold with eADR, which does not require those fences and flushes.

A. Experimental Platform

All experiments in this paper were conducted on a system containing two 2.30 GHz Intel Xeon Gold 5218 CPUs. Each CPU has 16 cores / 32 threads. Due to known scalability bottlenecks in PTM algorithms when crossing chips,Optane^{TM} experiments were limited to a maximum of 32 threads, with all threads pinned to a single chip. The machine ran Linux kernel version 4.14.35.

The system memory consists of two parts: 192GB of DRAM and 1.5 TB of Optane^{TM} DC memory. The Optane^{TM} memory was split across 12 DIMMs, and interleaving was enabled. This is the recommended configuration for maximizing the throughput of the Optane^{TM} memory. Since we limited experiments to a single chip, only half of the DRAM and half of the Optane^{TM} DC memory was available to the experiments. Note that the latencies of a `clwb` instruction are the same whether the cache line is being flushed to DRAM or Optane^{TM} DC. However, the latencies of loads and stores to DRAM are lower than the latencies of loads and stores to Optane^{TM} DC memory.

Software was compiled using LLVM/Clang 6.0 with O3 optimizations. We used the open-source LLVM PTM plugin [39], which provides a suite of different PTM algorithms [38]. We used the best-performing redo-based PTM ("orec-lazy") and the best-performing undo-based PTM ("orec-eager"), with every optimization enabled. We then tuned the algorithms for Optane^{TM} . The most significant modification was to the hash table used for undo and redo logging: we split it, placing the index in DRAM, with the copy of program data in Optane^{TM} memory. Experiments use the DAX filesystem and Makalu allocator [40] to manage memory from the persistent heap.

We consider every open-source multi-threaded PTM benchmark we could find. This led to the following experiments:

- The write-only TATP telecom application benchmark from DudeTM [16].
- Two microbenchmarks that stress the B+ Tree from DudeTM. The first is an insert-only workload that performs 2M insertions of unique keys into a tree that is initially empty; the second performs an equal mix of inserts, lookups, and removes using a key range of 2^{21}.
- Two configurations of the write-only TPCC benchmark from DudeTM, one using a B+ Tree, the other using a Hash Table.
- Two configurations of the Vacation travel reservation benchmark [41] from Whisper [42], at high and low contention, respectively.
- The memcached key/value store [32], [43], [44]. For this experiment, we ran memaslap on the second CPU to generate a stream of requests for memcached to process. The get/set ratio was set to 50/50, with 128B keys and 1KB values [45].

Each trial was run five times, and the average throughput is reported. With the exception of the B+Tree insert-only workload, each trial of each benchmark ran for one minute. We did not observe significant variance. Due to space constraints we defer discussion of memcached until Section IV-E, where we focus on the impact of large data sets.

B. Comparing DRAM and Optane^{TM} Behaviors in ADR

Figures 3 and 4 present the behavior of each benchmark at various thread levels, to understand whether past results that approximated Optane^{TM} latencies with DRAM can lead to reasonable conclusions about Optane^{TM} performance. In this subsection, we focus on the four curves marked "ADR". The "U" and "R" suffixes indicate whether an experiment used undo logging or redo logging. Past work has shown that when the working set of a transaction is not statically known (as is the case for all of our experiments), then undo logging incurs a fencing overhead linear in the number of writes (these serve to order the flushes of writes to the undo log *before* speculative writes to persistent data). Curves labeled "DRAM" correspond to executions in which the persistent data is stored in an 80GB DRAM ramdisk; that is, the data is not truly persistent. Curves labeled "Optane^{TM} " use Optane^{TM} DC memory in AppDirect mode for the persistent data. Both sets of curves have the same numbers of `clwb` instructions, and these instructions exhibit similar latencies regardless of whether data ultimately routes from the WPQ to DRAM or Optane^{TM} memory (86 ns and 94 ns, respectively [46]). The load latency on L3 misses is roughly $3\times$ higher for Optane^{TM} than DRAM [46].

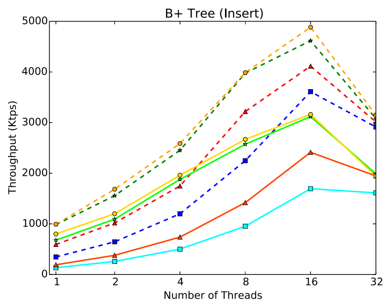
Threads	1	2	4	8	16	32
DRAM_ADR	0	21.5	28.68	33.31	46.13	63.43
DRAM_eADR	0	27.55	36.57	44.99	59.21	87.02
Optane_ADR	0	26.56	24.96	25.15	34.51	49.56
Optane_eADR	0	25.96	29.13	31.29	47.3	70.31

TABLE I: Ratio of commits to aborts for TPCC (Hash Table) with redo logging (ADR).

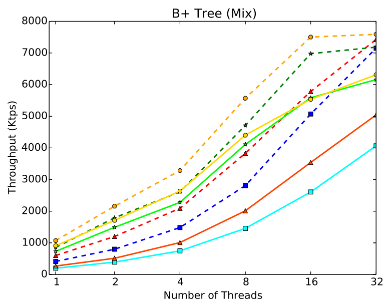
Our first finding is that past recommendations regarding the costs of undo logging remain true: in almost every case, redo logging outperforms undo logging. This is despite the higher instruction count for redo logging (due to reads performing lookups in the redo log), and a direct consequence of the cost of fences for undo logging. While these fences could be aggregated via static analysis for workloads whose write sets are predictable, in our workloads such analysis is not possible. The only outlier for this finding is the TATP workload: every TATP transaction performs a small number of writes, and thus the cost of fences is not as significant as in other workloads.

We also found that the timing of `clwb` instructions does not affect performance. In the redo log experiments, writes to the redo log must be flushed before the transaction commits. The flushes could be done incrementally, upon each write to the redo log, or in a tight loop immediately before committing. We expected the latter option to increase pressure on the WPQs, and increase latency. However, our experiments showed no noticeable difference in performance: performing many flushes at once did not create more pressure on the WPQ than staggering the flushes during transaction execution.

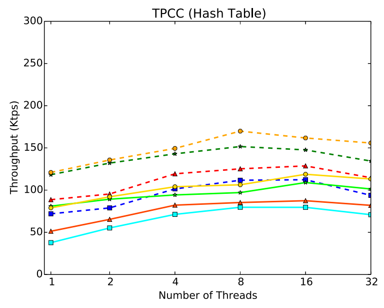
Finally, we see that scalability on Optane^{TM} is *worse* than scalability on DRAM. For example, in the Vacation workloads the maximum throughput is reached at a lower thread count, and the gap at peak throughput is substantially larger than the gap at low thread counts. To explore this behavior in more detail, Tables I and II report the number of commits per abort for the TPCC (Hash Table) workload. There are two important trends. The first is that the ratios are lower for Optane^{TM} than DRAM at every thread level. The second is that the ratio decreases more rapidly for Optane^{TM} than for DRAM. During a transaction's execution, it is inevitable that some of the added fences and flushes must occur while a transaction is holding locks. These fences and flushes extend the duration of the critical section, and thus increasing the window of contention during which other transactions will abort. In addition, it is known that Optane^{TM} DC reads tend to scale with the thread count, whereas writes reach their maximum throughput quickly. For example, Izraelevitz et al. needed 17 threads to reach the maximum read throughput of Optane^{TM} DC, but only 4 to reach the maximum write throughput [46].



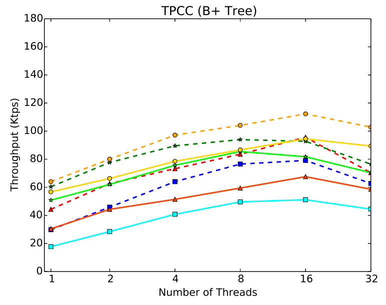
(a) B+ Tree Insert



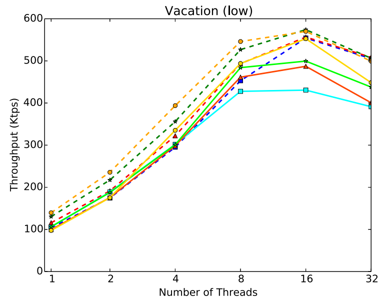
(b) B+ Tree Mix



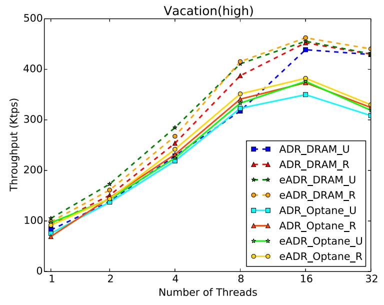
(c) TPCC (Hash Table)



(d) TPCC (B+ Tree)

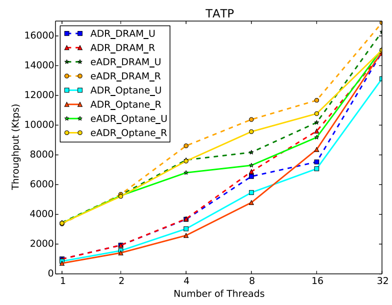


(e) Vacation (low)



(f) Vacation (high)

Fig. 3: Performance comparison between DRAM (not persistent) and Optane^{TM} for the B+Tree, TPCC, and Vacation workloads.



(a) TATP

Fig. 4: Performance comparison between DRAM (not persistent) and Optane^{TM} for the TATP workload.

Threads	1	2	4	8	16	32
DRAM_ADR	0	3.44	2.42	2.05	1.82	1.66
DRAM_eADR	0	4.63	2.99	2.64	2.02	1.65
Optane_ADR	0	3.34	2.16	1.71	1.39	1.4
Optane_eADR	0	3.76	2.7	1.73	1.22	1.12

TABLE II: Ratio of commits to aborts for TPCC (Hash Table) with undo logging (ADR).

C. Contrasting eADR and ADR Performance

Next, we compare the performance of the system under the ADR and eADR durability domains. For the purposes of these experiments, we assume that the system has enough reserve power to flush all cached Optane^{TM} pages back to Optane^{TM} DIMMs in the event of a system failure. Then, we can transform the ADR algorithms to eADR by eliding `clwb` and fence instructions.

Returning to Figures 3 and 4, the most significant finding is that eADR provides substantial performance gains for every workload except Vacation. When we focus on the "redo" PTMs, this result speaks to the latency of `clwb` instructions, as they are the only aspect of the algorithm that changes. Clearly, avoiding the need to flush cache lines to the memory controller has a significant impact on performance. In addition, even Vacation sees improvements, but these improvements are muted somewhat. This is largely a consequence of Vacation having non-trivial amounts of work between transactions: the fraction of the program that is transactional (and hence affected by eADR) is greater in the other workloads.

To understand these gains in more detail, we created an incorrect version of our PTM algorithms, in which ADR algorithms continued to use correct `clwb` instructions, but did not issue any memory fences. A snapshot of the latency improvements appear in Table III. In comparing the numbers in the table to the results in Figures 3 and 4, the main finding is that a substantial fraction of the improvement results from removing fences.

Even with these advantages, eADR still does not reach the performance of DRAM. There are two related factors which introduce latency. The first is that the WPQs are bounded, and become saturated. The second is that write latency is higher for Optane^{TM} than for DRAM. Note that while the eADR PTMs do not explicitly issue `clwb` instructions, data still evicts

	TPCC	TATP	Vacation (low)	Vacation (high)
Undo	8%	10%	17%	12%
Redo	10%	10%-27%	7%-17%	7%-17%

TABLE III: Speedup from removing memory fences from write instrumentation in ADR algorithms.

from the L3 to Optane^{TM} , through the WPQs. In separate experiments, we measured the performance counters for L3 hits and misses, as well as for DRAM and `pmem` throughput. These measurements showed that eADR workloads were writing back to the Optane^{TM} with a lower bandwidth than DRAM writeback; this explains the remaining latency. The known problem of WPQ saturation [46] explains the decrease in scalability.

IV. New Models for Persistence

In Section III, we observed that eADR can substantially improve performance versus ADR, primarily because eADR does not require explicit fences and flushes. In this section, we introduce two new durability models, which are able to deliver better performance than eADR. While neither is available in hardware today, nor does either require substantially different support than is available in Optane^{TM} DC systems today.

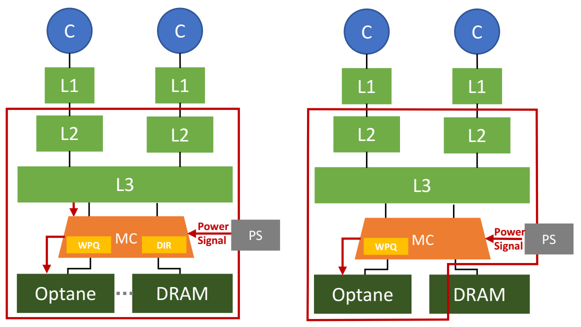
The fundamental enabling mechanism for our new durability models is the directory used by the memory controller when the system runs in Memory Mode. Recall from Section II that Optane^{TM} can run in either AppDirect Mode or Memory Mode. In AppDirect Mode, a filesystem on the Optane^{TM} DC memory is mapped into the virtual address space of the program. In Memory Mode, the memory controller maintains a directory in DRAM, and uses the directory to create the illusion that DRAM is a cache of physical Optane^{TM} DC pages. The controller is then responsible for writing DRAM pages back to Optane^{TM} when those physical DRAM pages are to be used to cache different physical Optane^{TM} pages.

A. The Persistent DRAM Durability Domain

Our first new durability domain, PDRAM, gives the illusion that all of DRAM is persistent. It combines the persistence of AppDirect Mode with the caching behavior of Memory Mode. In more detail, let F be a range of persistent physical pages in AppDirect Mode that are managed as a file. To use the mechanisms of Memory Mode to cache pages of F in DRAM, few changes are required. Let D_i be the ith page of DRAM, and let P_j be the jth page of Optane^{TM} memory allocated to F. Note that the directory in Memory Mode already provides the following behaviors:

- If D_i is to cache P_j, then D_i must be initialized with data from P_j before the first read or write of D_i.
- While D_i is caching P_j, reads and writes of P_j can be satisfied by routing them to D_i.
- If D_i is dirty, and D_i is needed to cache some new page P_k, then D_i must be written back to P_j first.

In addition to tracking which pages are dirty, the memory controller already implements policies that asynchronously write dirty pages from DRAM to Optane^{TM} , and that prefetch pages from Optane^{TM} to DRAM.



(a) PDRAM (b) PDRAM-Lite

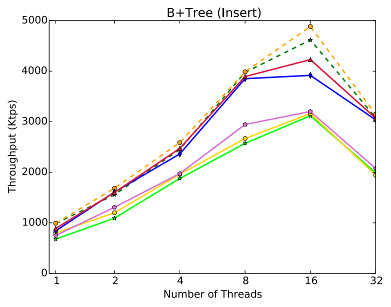
Fig. 5: Proposed Durability Domains. In PDRAM, every DRAM page can potentially cache a page of Optane^{TM} memory, and sufficient battery power is required to flush every DRAM page to Optane^{TM} on a power failure. In PDRAM-Lite, a bounded number of DRAM pages can cache Optane^{TM} memory.

Given the above properties, the only reason why P_j is not persistent is energy: if some large number of pages D_i are dirty, then on a system failure, there must be enough reserve power to flush all data from the caches to DRAM, and then write all of the dirty pages of DRAM to the Optane^{TM} memory. With a limited number of WPQs, and writeback occurring at cache-line granularity, a single 4KB page would require 64 writebacks, which would exceed the WPQ capacity. Thus the required reserve power would need to be enough to keep the entire CPU and memory system running for quite some time.

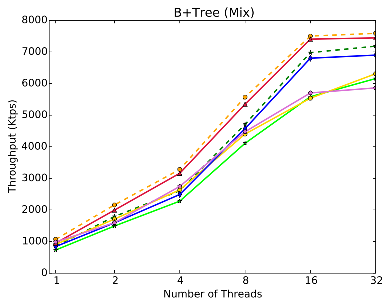
Figure 5(a) depicts the PDRAM Durability Domain. Like eADR, it treats the caches as persistent. However, it requires a directory in DRAM, so that it can potentially flush all of DRAM to Optane^{TM} on a power signal.

B. The PDRAM-Lite Durability Domain

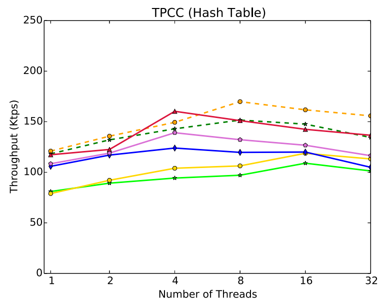
While the mechanisms for enabling PDRAM are largely present in existing systems (to support Memory Mode), our PDRAM proposal is still idealistic, in that it requires a significant amount of reserve power, most likely in the form of an external battery. We note that making *all* of DRAM into a cache of Optane^{TM} memory may not be advantageous. ADR increases Optane DIMM power draw, because its lack of write coalescing leads to more power-hungry writes. eADR requires 1s of reserve power (capacitors) for write back on a power failure (power leakage and additional manufacturing cost are assumed to be negligible). PDRAM would use more power to drive its DRAM cache. Assuming RAM consumes 50% of system power, if half of DRAM was used as a PDRAM cache, system power requirements could increase by as much as 25%, and $> 10s$ of reserve power could be needed. This would necessitate a lithium-ion battery, bringing non-negligible leakage (though likely still under 3W). We expect



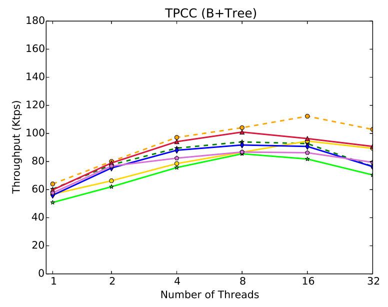
(a) B+Tree Insert



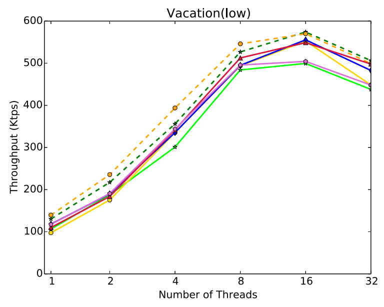
(b) B+Tree Mix



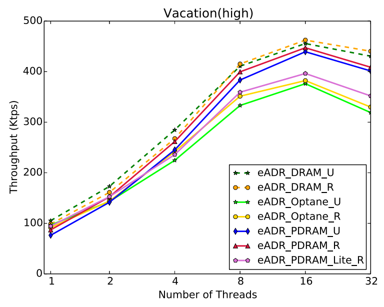
(c) TPCC (Hash Table)



(d) TPCC (B+Tree)

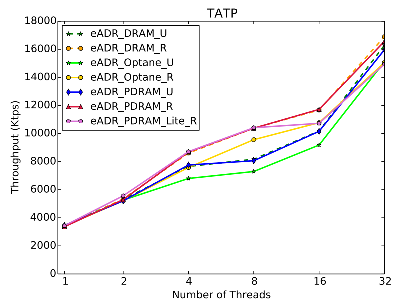


(e) Vacation (low)



(f) Vacation (high)

Fig. 6: Performance comparison between different durability models for the B+Tree, TPCC, and Vacation workloads.



(a) TATP

Fig. 7: Durability model performance (TATP workload).

PDRAM-Lite's cache to be a small fraction of DRAM, with corresponding decreases in system and reserve power.

On the one hand, certain memory regions (such as the stack, or the lookup tables of a redo log) typically do not require persistence. Additionally, the specific case of redo-based PTM has simpler persistence requirements than undo-based PTM. As we shall see, for some workloads a redo-based PTM can get by with a lightweight variation on PDRAM, which we call PDRAM-Lite, and show in Figure 5.

From the previous experiments presented in this paper, we can conclude that PTM favors redo logging over undo logging even under the eADR durability domain. This is primarily because of the reduction in fences: with undo logging, each update to persistent state must be preceded by a store to the persistent undo log, ordered via `sfence`. In redo logging, the only fences are to ensure that all redo log entries are persisted before writeback begins, and to order writeback with respect to status updates. Thus if a transaction performs W writes, undo logging will perform $O(W)$ fences, and redo logging will perform $O(1)$ fences.

If we consider the timing of a system failure with respect to a program that uses redo-based PTM, we see that storing the redo log persistently is necessary, but the persistence is often overkill. Suppose that a transaction T has not yet reached its commit point. In that case, if a system failure occurs, the recovery procedure will re-try T; the previous redo log is discarded. Furthermore, in the common case where system failures do not happen, an about-to-commit transaction will persist its redo log, mark itself committed, write back the redo log, and then discard the redo log. Redo logs are *ephemeral* and *rarely require persistence*.

At the same time, notice that in redo-based PTM, a transaction only performs stores to the Optane^{TM} memory *at commit time*. Until the commit point, a redo-based transaction keeps its entire write working set in the (highly compact) redo log. For transactions with modest write set sizes, it would be possible to use a small amount of PDRAM for the transactions' redo logs, without caching any other Optane^{TM} pages in DRAM. We refer to this approach as PDRAM-Lite. As shown in Figure 5, a smaller directory is needed to track the small number of DRAM pages that serve as a cache of Optane^{TM}

pages. Furthermore, when a power failure signal arrives, after flushing the caches, the recovery operation can decide which pages of PDRAM-Lite to flush to Optane^{TM} by checking the state of the corresponding transactions. For any transaction that is still in-flight, its redo log can be skipped.

While PDRAM-Lite will require more reserve power than eADR, we expect that in many workloads, only small amounts of memory will suffice, and thus the energy overhead will be modest. For example, the Vacation benchmark never requires more than 37 contiguous cache lines (roughly half a page) for its redo log. TPCC (Hash Table) requires at most 36 cache lines. If these are representative of emerging PTM workloads, then a handful of pages per thread, with a fall-back to using Optane^{TM} memory directly, should suffice.

C. Simulating PDRAM

To simulate PDRAM-Lite, we modified the redo log implementation for the eADR redo-based PTM implementations, placing the entire log in DRAM. In additional testing, we found that the difference in latency between DRAM and Memory Mode was negligible for applications with working sets up to 16MB, and thus we expect the latency of redo log accesses in PDRAM-Lite to be a fair approximation. As with eADR, we ignore the fact that our system does not have enough reserve power to provide durability guarantees.

More interestingly, we found that it is possible to simulate the full PDRAM proposal on existing Optane^{TM} DC systems. The mechanics of PDRAM are already employed in Memory Mode; the challenge is only to make the data persist. We found that when running in Memory Mode, we could create an 750GB RAM Disk on the Optane^{TM} memory, and then `mmap()` it into the virtual address space of the application. Loads and stores would typically route to the DRAM cache, but would ultimately (e.g., on program termination) be flushed back to the Optane^{TM} memory. As with PDRAM-Lite, this simulation does not account for the added reserve power that would be required to flush caches and write back dirty pages from DRAM. However, the latencies we observed were in keeping with expected latencies for DRAM and Optane^{TM} DC.

D. Evaluation of PDRAM and PDRAM-Lite

Figures 6 and 7 repeat the experiments of Section III. As before, the "DRAM" curves show the performance of the PTM workloads and algorithms when only accessing DRAM. The eADR curves are also the same. However, we now add two curves that simulate PDRAM, using redo and undo logging, as well as a redo logging PTM that simulates PDRAM-Lite.

The first goal of these experiments is to determine whether PDRAM can bridge the gap between DRAM and eADR. The result is largely affirmative. In TATP, the B+Tree microbenchmarks, and Vacation, PDRAM matches DRAM performance up until Optane^{TM} scalability bottlenecks (WPQ saturation) occur. The same is generally true for TPCC.

The second goal is to determine whether PDRAM-Lite offers sufficient value. The result here is less clear: PDRAM-Lite outperforms eADR in every case, but the gains are marginal for all but TATP and TPCC. In fact, this result confirms a finding from [46]: Optane^{TM} DC throughputs are much closer to DRAM throughput for regular access patterns than for irregular patterns. Moving the redo log into DRAM does not have a significant impact on latency, since the compact log, with its regular access pattern, did not have much worse latency than DRAM to begin with.

E. Exploring the Impact of Workload Size with Memcached

We conclude our evaluation by investigating the impact of working set size on the throughput of transactions. Figure 8 presents the throughput (requests per second) of memcached with a single worker thread. We vary the working set size by changing the number of items stored in the cache. In the experiment, a set of client threads, running on a separate NUMA socket, issue an equal mix of get and set commands, using random keys. This leads to poor locality, such that every read will effectively be handled by the smallest level of the memory hierarchy that is capable of holding the entire working set. In addition, by limiting the server to a single thread, we avoid saturating the read or write bandwidth of the Optane^{TM} DC memory. In this way, we are able to isolate the latency of Optane^{TM} DC vs. DRAM.

The experiment considers a small working set (32MB), which fits in the L3 cache. It then considers working sets starting at 32GB and increasing in increments of 64GB. At 96GB, the working set cannot fit in DRAM, nor can it be completely cached in DRAM (e.g., for PDRAM). In this manner, we are able to observe how the Optane^{TM} DC memory behaves for ADR, eADR, PDRAM, and PDRAM-Lite.

For the PDRAM-Lite approach, we observe a broad trend throughout the experiments: its performance is only marginally better than Optane^{TM} performance with the eADR durability model and redo logging. This result is reasonable, since the highest Optane^{TM} overheads relate to random reads and writes; the PDRAM-Lite model only focuses on writes to the redo log, which are regular and hence do not incur the highest latencies.

Next, we observe a precipitous drop in performance for all configurations when the working set increases from 32MB to 32GB. This is expected, since the L3 is able to cache both DRAM and Optane^{TM} locations at 32MB (for ADR, only Optane^{TM} reads benefit from caching; for eADR and the PDRAM approaches, Optane^{TM} writes also benefit). At the same time, fitting the working set in the L3 does not overcome the fundamental differences between the algorithms. Stores to Optane^{TM} still must eventually flush, and flush more slowly than flushes of stores to DRAM. Additionally, `clwb` and fence instructions (in ADR) have unavoidable latencies.

The next interesting point on the X axis is the movement from 32GB to 96GB. At this point, DRAM no longer caches the entire working set, which affects PDRAM and PDRAM-Lite systems. Note that for the DRAM curves, operation beyond 96GB is not possible. For ADR and eADR curves, no drop-off was expected, since these algorithms do not use DRAM to store persistent data. Surprisingly, the PDRAM and PDRAM-Lite algorithms did not experience any slowdowns

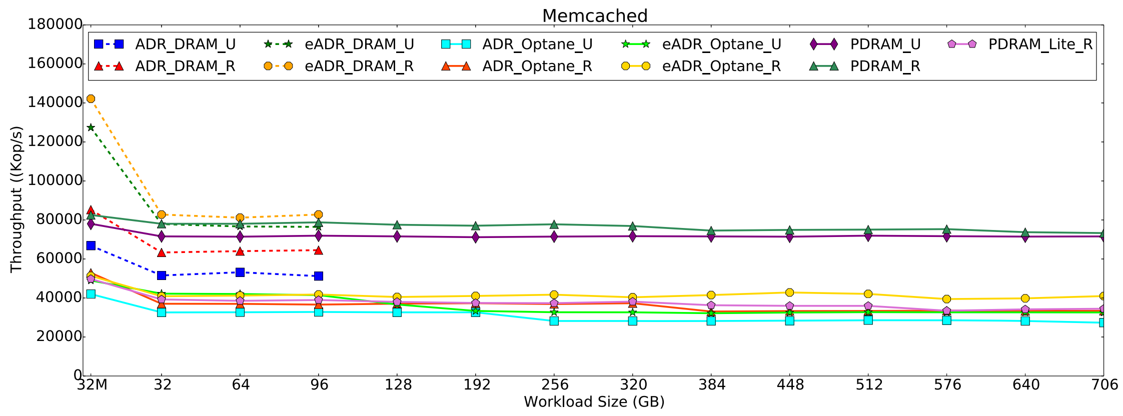


Fig. 8: Performance comparison of memcached at different working set sizes. Beyond 32M, the working set does not fit in the L3 cache, and beyond 96 GB, the working set does not fit in DRAM.

at this point; only eADR+Undo slowed down at 96GB, and ADR+Undo had a similar slowdown at 192GB. Delving deeper into why these specific combinations degrade is future work. Our suspicion is that hash table re-balancing may be occurring more frequently for these workload combinations, but additional testing is required.

We also observed a slowdown at 320GB. One important factor at this point is that memcached stores an index as well as values; when the index is cacheable but the data is not, performance does not degrade as rapidly. For our machine and workload, 320GB is the point where there ceases to be much profitable caching of the index, and the entire workload runs at the speed of the DRAM or Optane^{TM} memory.

V. Conclusions and Future Work

In this paper, we studied the behavior of highly-optimized PTM algorithms on a system with Intel® Optane^{TM} Direct Connect (Optane^{TM} DC) memory. To the best of our knowledge, this is the first work to directly study PTM performance on Optane^{TM} DC.

Our main finding is that the durability model significantly impacts PTM performance on Optane^{TM} DC.. ADR, which requires explicit flushes of cache lines to the memory controller's WPQs, and explicit fences to ensure the ordering of flushes with respect to stores, is substantially slower than eADR, which assumes enough reserve power to flush caches to the Optane^{TM} DC in the event of a power failure. Despite eADR's higher performance, it is still below DRAM. Characteristics like bounded WPQs appear to create higher single-thread latency and worse scalability, because the Optane^{TM} DC memory bandwidth can saturate with many fewer writing threads than are needed to saturate DRAM bandwidth.

Inasmuch as systems rarely fail, the defensive measures used by PTM algorithms are usually overkill. In recognition of that reality, we introduced two new durability domains that made all (PDRAM) or some (PDRAM-Lite) of DRAM to be persistent. While hardware does not support this behavior today, we argued that the necessary support is present in Optane^{TM} DC systems, to support the non-durable Memory Mode of operation. We then presented realistic software emulation of these durability domains, and evaluated PTM performance. While PDRAM performed as expected, and largely closed the gap between Optane^{TM} DC and DRAM, PDRAM-Lite did not. Workload and Optane^{TM} DC characteristics simply do not result in high latency at the places in the PTM algorithm where PDRAM-Lite can deliver improvement.

Our findings raise a number of questions that we leave to future work. First and foremost is the question of reserve power. The ADR durability domain exists today, with enough reserve power. It is hypothesized that modest batteries would enable eADR. We do not have an estimate of the energy overhead to support PDRAM, nor do we have a formula or model for estimating reserve power requirements for a workload. As future work, we plan to investigate the energy consumption of the durability domains.

Another open question is whether hardware transactional memory (HTM), or hardware acceleration of STM, is a viable strategy for accelerating PTM. In particular, while Intel® Transactional Synchronization Extensions are incompatible with PTM in ADR, they might work with eADR and PDRAM. If so, it may be that HTM techniques reduce latency and aid scalability, or that HTM just causes the WPQs to saturate with fewer writing threads. Studying HTM behavior in eADR is an exciting topic for future work.

Acknowledgments

We thank Brian Hirano for many insightful conversations during the conduct of this research. At Lehigh, this work was supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) under Grant CCF-1723624. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Intel.

References

[1] G. Goindi, “Exadata Persistent Memory Accelerator: Partnering with Intel on Optane DC Persistent Memory,” 2018, https://blogs.oracle.com/exadata/persistent-memory-accelerator.

[2] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-change technology and the future of main memory,” *IEEE micro*, vol. 30, no. 1, 2010.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[4] A. Tulapurkar, Y. Suzuki, A. Fukushima, H. Kubota, H. Maehara, K. Tsunekawa, D. Djayaprawira, N. Watanabe, and S. Yuasa, “Spin-torque diode effect in magnetic tunnel junctions,” *Nature*, vol. 438, no. 7066, p. 339, 2005.

[5] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto *et al.*, “A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram,” in *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* IEEE, 2005, pp. 459–462.

[6] Y. Huai, “Spin-transfer torque mram (stt-mram): Challenges and prospects,” 2009.

[7] A. Sawa, “Resistive switching in transition metal oxides,” *Materials today*, vol. 11, no. 6, pp. 28–36, 2008.

[8] H. Akinaga and H. Shima, “Resistive random access memory (reram) based on metal oxides,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[9] J. Condit, E. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O through byte-addressable, persistent memory,” in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, Montana, USA, Oct. 2009.

[10] J. Izraelevitz, H. Mendes, and M. L. Scott, “Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model,” in *Proceedings of the 30th International Symposium on Distributed Computing*, Paris, France, Sep. 2016.

[11] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, “Fine-Grain Checkpointing with In-Cache-Line Logging,” in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, Apr. 2019.

[12] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott, “Dalí: A Periodically Persistent Hash Map,” in *Proceedings of the 31st International Symposium on Distributed Computing*, Vienna, Austria, Oct. 2017.

[13] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, “A Persistent Lock-Free Queue for Non-Volatile Memory,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vienna, Austria, Feb. 2018.

[14] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ACM SIGARCH Computer Architecture News*, March 2011.

[15] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories,” in *Proceedings of the Sixteenth ASPLOS*, New York, NY, USA, Mar. 2011.

[16] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “DudeTM: Building Durable Transactions with Decoupling for Persistent Memory,” in *Proceedings of the 22nd ASPLOS*, Xi’an, China, Apr. 2017.

[17] E. R. Giles, K. Doshi, and P. Varman, “Softwrap: A lightweight framework for transactional support of storage class memory,” in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on.* IEEE, 2015, pp. 1–14.

[18] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 399–411, 2016.

[19] Intel, “pmem.io: Persistent memory programming blog.” 2014, https://pmem.io/.

[20] B. Bridge, “Nvm support for c applications.(2015),” 2015.

[21] Intel Corporation, “Persistent memory development kit,” http://pmem.io/pmdk/.

[22] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, “OneFile: A Wait-Free Persistent Transactional Memory,” in *Proceedings of the 49th International Conference on Dependable Systems and Networks*, Portland, OR, Jun. 2019.

[23] A. Correia, P. Felber, and P. Ramalhete, “Romulus: Efficient algorithms for persistent transactional memory,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2018, pp. 271–282.

[24] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, “Pisces: a scalable and efficient persistent transactional memory,” in *2019 {USENIX} Annual Technical Conference (ATC*, 2019, pp. 913–928.

[25] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging Locks for Non-Volatile Memory Consistency,” in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 433–452.

[26] D. Dice, O. Shalev, and N. Shavit, “Transactional Locking II,” in *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

[27] P. Felber, C. Fetzer, and T. Riegel, “Dynamic Performance Tuning of Word-Based Software Transactional Memory,” in *Proceedings of the 13th PPoPP*, Salt Lake City, UT, Feb. 2008.

[28] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[29] Intel Corporation, “Nvml: Implementing persistent memory applications,” https://www.snia.org/sites/default/files/.

[30] SNIA, “NVM Programming Technical Work Group,” https://www.snia.org/forums/sssi/nvmp/.

[31] Intel, “NVDIMM Block Window Driver Writer’s Guide.” http://pmem.io/documents/NVDIMM_DriverWritersGuide-July-2016.pdf.

[32] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, “Persistent memory transactions,” in *arXiv preprint arXiv:1804.00701*, 2018.

[33] *Intel Architecture Instruction Set Extensions Programming Reference*, 319433rd ed., Intel Corp., Feb. 2012.

[34] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, “The rio file cache: Surviving operating system crashes,” in *Acm Sigplan Notices*, vol. 31, no. 9. ACM, 1996, pp. 74–83.

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

[36] D. Narayanan and O. Hodson, “Whole-system persistence,” *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 401–410, 2012.

[37] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey III, “Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience.” in *EDBT*, 2015, pp. 689–694.

[38] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear, “Optimizing Persistent Memory Transactions,” in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, Sep. 2019.

[39] P. Zardoshti, T. Zhou, P. Balaji, M. L. Scott, and M. Spear, “Simplifying Transactional Memory Support in C++,” p. 25, 2019.

[40] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, “Makalu: Fast recoverable allocation of non-volatile memory,” in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 677–694.

[41] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford Transactional Applications for Multi-processing,” in *Proceedings of IISWC*, Seattle, WA, Sep. 2008.

[42] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with whisper,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS’17, Xi’an, China, April 2017.

[43] memcached.org, “Memcached, a distributed memory object caching system.” 2014, http://memcached.org/.

[44] W. Ruan, T. Vyas, Y. Liu, and M. Spear, “Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[45] M. Zhuang and B. Aker, “memaslap-load testing and benchmarking a server,” 2003.

[46] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module,” 2019.