

# Ribbon: High Performance Cache Line Flushing for Persistent Memory

Kai Wu

University of California, Merced  
kwu42@ucmerced.edu

Jie Ren

University of California, Merced  
jren6@ucmerced.edu

Ivy Peng

Lawrence Livermore National Laboratory  
peng8@llnl.gov

Dong Li

University of California, Merced  
dli35@ucmerced.edu

## ABSTRACT

Cache line flushing (CLF) is a fundamental building block for programming persistent memory (PM). CLF is prevalent in PM-aware workloads to ensure crash consistency. It also imposes high overhead. Extensive works have explored persistency semantics and CLF policies, but few have looked into the CLF mechanism. This work aims to improve the performance of CLF mechanism based on the performance characterization of well-established workloads on real PM hardware. We reveal that the performance of CLF is highly sensitive to the concurrency of CLF and cache line status.

We introduce Ribbon, a runtime system that improves the performance of CLF mechanism through concurrency control and proactive CLF. Ribbon detects CLF bottleneck in oversupplied and insufficient concurrency, and adapts accordingly. Ribbon also proactively transforms dirty or non-resident cache lines into clean resident status to reduce the latency of CLF. Furthermore, we investigate the cause for low dirtiness in flushed cache lines in in-memory database workloads. We provide cache line coalescing as an application-specific solution that achieves up to 33.3% (13.8% on average) improvement. Our evaluation of a variety of workloads in four configurations on PM shows that Ribbon achieves up to 49.8% improvement (14.8% on average) of the overall application performance.

## CCS CONCEPTS

• **Hardware** → **Emerging technologies**; • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Concurrency control*.

## KEYWORDS

persistent memory; Optane; cache flush; runtime; concurrency

### ACM Reference Format:

Kai Wu, Ivy Peng, Jie Ren, and Dong Li. 2020. Ribbon: High Performance Cache Line Flushing for Persistent Memory. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8075-1/20/10...\$15.00

<https://doi.org/10.1145/3410463.3414625>

(PACT '20), October 3–7, 2020, Virtual Event, GA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3410463.3414625>

## 1 INTRODUCTION

Persistent memory (PM) technologies, such as Intel Optane DC PM [25, 56], provide large capacity, high performance, and a convenient programming interface. Data access to PM can use load/store instructions as if to DRAM. However, the volatile cache hierarchy on the processor imposes challenges on data persistency and program correctness. A store instruction may only update data in the cache, not persisting data in PM immediately. When data is written from the cache back to memory, the order of writes may differ from the program order due to cache replacement policies.

Data in PM needs to be in consistency state to be able to recover the program after a system or application crash. Therefore, cache line flushing (CLF) is a fundamental building block for programming PM. Most PM-aware systems and applications [3, 4, 8, 11, 15, 19, 20, 39, 41, 55, 57–59, 63, 64] rely on CLF and memory fences to ensure that data is persisted in the correct order so that the state in PM is recoverable.

CLF can be an expensive operation. CLF triggers cache-line-sized write to the memory controller, even if the cache line is only partially dirty. Also, CLF needs persist barriers, e.g., the memory fence, to ensure that flushed data has reached the persistent domain before any subsequent stores to the same cache line could happen. Our preliminary evaluation shows that CLF can reduce system throughput by 62% for database applications like Redis. Hence, CLF creates a performance bottleneck on PM and may significantly reduce the performance benefits promised by PM.

Most of the existing techniques focus on optimizing persistency semantics, other than the CLF mechanism [2, 15, 24, 30, 42, 46, 52, 62]. Skipping CLF [2, 46] or relaxing constraints on persist barriers [15, 24, 30, 42, 52, 62], these techniques improve application performance by reducing CLF. Each technique may have a different fault model and recovery mechanism that is designed for specific application characteristics. Still, these techniques use CLF to implement their persistency semantics.

In this paper, we focus on the CLF mechanism, instead of persistency semantics. Therefore, our work applies to general PM-aware applications. We reveal the characteristics of CLF on real PM hardware. Based on our performance study, we introduce a runtime system called *Ribbon* that decouples CLF from the application and applies model-guided optimizations for the best performance. Applying Ribbon on a PM-aware application does not change its

persistence semantics, i.e., fault models and recovery mechanisms, so that the program correctness is retained.

Our performance study of CLF on real PM hardware reveals three optimization insights. *First*, concurrent CLF can create resource contention on the hardware buffer inside PM devices and memory controllers, which causes performance loss. We define CLF concurrency as the number of threads performing CLF simultaneously. *Second*, the status of a cache line can impact the performance of CLF considerably. For instance, flushing a clean cache line could be 3.3 times faster than flushing a dirty cache line. *Third*, many flushed cache lines have low dirtiness, wasting memory bandwidth and decreasing the efficiency of CLF. The dirtiness of a cache line is quantified as the fraction of dirty bytes in the cache line. Since a cache line is the finest granularity to enforce data persistency, the whole cache line has to be flushed, even if only one byte is dirty. Our evaluation of Redis with YCSB (Load and A-F) and TPC-C workloads shows that the average dirtiness of flushed cache lines is only 47%.

We introduce three techniques in Ribbon to improve the CLF mechanism. First, Ribbon controls the intensity of CLF by thread-level concurrency throttling. Optimal concurrency control needs to address two challenges. How to avoid the impact of concurrency control on application computation? How to determine the appropriate CLF concurrency? Simply changing thread-level parallelism can reduce thread-level parallelism available for the application. Our solution is to decouple CLF from the application. We instrument and collect CLF in the application and manage a group of flushing threads to perform CLF. This design supports flexible concurrency control without impacting application threads. Furthermore, we introduce an adaptive algorithm to select the concurrency level of these flushing threads. The algorithm achieves a balance between mitigating contention on PM devices and increasing CLF parallelism for utilizing memory bandwidth.

We propose a proactive CLF technique to increase the possibility of flushing clean cache lines. Flushing a clean cache line is significantly faster than flushing dirty one. Proactive CLF may change the status of a cache line from dirty to clean before the application starts flushing this cache line. Ribbon leverages hardware performance counters in the sampling mode to opportunistically detect modified cache lines with negligible performance overhead.

Ribbon coalesces cache lines of low dirtiness to reduce the number of cache lines to flush. We find that unaligned cache-line flushing and uncoordinated cache-line flushing are the main reasons for low dirtiness in flushed cache lines. These problems stem from the fact that existing memory allocation mechanisms are designed for DRAM. Ribbon introduces a customized memory allocation mechanism to coalesce cache-line flushing and improve efficiency.

We summarize our contributions as follows.

- We characterize the performance of the CLF mechanism in PM-aware workloads on real PM hardware;
- We propose decoupled concurrency control, proactive CLF, and cache line coalescing to improve performance of the CLF mechanism;
- We design and implement Ribbon, a runtime to optimize PM-aware applications automatically;

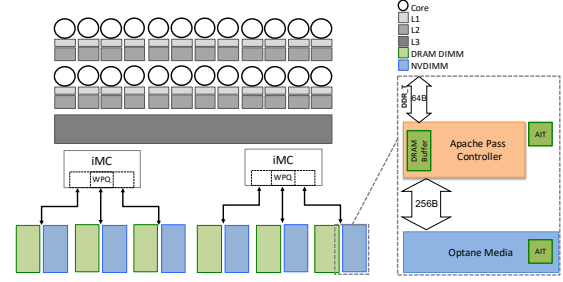


Figure 1: The Intel Optane persistent memory architecture.

- We evaluate Ribbon on a variety of PM-aware workloads and achieve up to 49.8% improvement (14.8% on average) in the overall application performance.

## 2 BACKGROUND AND MOTIVATION

In this section, we introduce the state-of-art persistent memory architecture and review common CLF policies.

### 2.1 Persistent Memory Architecture

In the most recent PM architecture (i.e., Intel Optane DC Persistent Memory Module, shortened as Optane), PM and DRAM are placed side-by-side and connected to CPU through memory bus. Figure 1 illustrates this architecture on one socket. Two integrated memory controllers (iMC) manage a total of six memory channels, each connecting to two DIMMs – a DRAM DIMM and an NVDIMM. Data is guaranteed to become persistent only after it reaches iMC. In cases of power failure, data in write pending queue (WPQ) in iMC will be flushed to NVDIMM by hardware. When WPQ has high occupancy, write blocking effect could stall CPU if threads have to wait for the WPQ to drain [56].

The inset in Figure 1 depicts the internal architecture of Optane. The host CPU and Optane communicate at 64-bytes granularity through the non-standard DDR\_T protocol, while Optane internal transactions are in 256 bytes. Within the Optane device, there is a controller (the Apache Pass controller) that manages address mapping for wear-leveling. There is also a small DRAM buffer within the Optane device to improve the reuse of fetched data and reduce write-amplification [25].

### 2.2 Cache Line Flushing

On-chip data caches are mostly implemented with volatile memory like SRAM. Because of the prevalence of volatile caches, data corruption could occur if updates to a data object stay in the cache but have not reached the persistent domain when a crash happens. A persistent domain refers to the part of the memory hierarchy that can retain data through a power failure. For instance, the system from iMC to Optane media is the persistent domain on the Optane architecture [25]. For data persistency and consistency, the programmer typically employs ISA-specific CLF instructions, such as `clflush`, `clflushopt`, and `clwb` on x86 machines [23], to ensure that data in a cache line is pushed to the persistent domain. The order of two CLF can be enforced by an `sfence` instruction, which

ensures the second CLF does not happen before the first one reaches the persistent domain.

The standard practice to ensure persistence of a data object in PM is to flush all cache blocks<sup>1</sup> of the data object [23], even though the data object may not be fully cached. Because of the complexity and overhead of tracking dirty cache lines or checking resident cache blocks for a particular data object in the existing hardware, every cache block of the data object is flushed by software, exemplified in Listing 1. The example is a code snippet from Intel PMDK [23].

**Listing 1: An example of persisting a data object**

```
1 /*Loop through cacheline aligned chunks*/
2 /*covering a target data object*/
3 cache_block_flush(const void *addr, size_t len)
4 {
5     unsigned __int64 ptr;
6     for (ptr = (unsigned __int64)addr & ~(
7         FLUSH_ALIGN - 1);
8         ptr < (unsigned __int64)addr + len;
9         ptr += FLUSH_ALIGN)
10         /* clflush / clflush_opt / clwb */
11         flush((char *)ptr);
12     /* clflush_opt and clwb needs a fence */
13     _mm_sfence();
14 }
```

## 2.3 Optimization of Cache Line Flushing

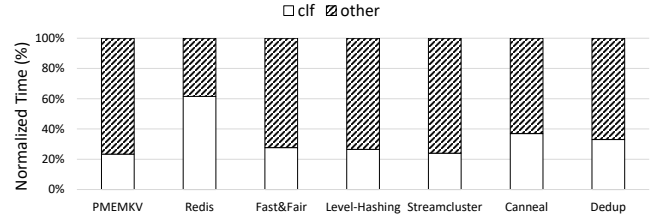
Flushing cache lines from the volatile cache into the persistent domain is the building block for programming persistent memory. Active research in different PM access interfaces – libraries [9, 23, 52], multi-threaded programming models [7, 18, 19], and file systems [11, 15, 54, 55] – proposes optimizations to mitigate the high overhead of CLF. We categorize existing CLF optimizations into five classes, summarized as follows.

**Eager CLF** triggers CLF explicitly at the application level after the data value is updated. There is no delay of CLF and no skip of CLF. This kind of CLF provides strict persistency [42], but often introduces excessive constraints on write ordering, limiting the concurrency of writes. Frequently performing eager CLF could impose high performance cost [2, 45, 46, 58, 61].

**Asynchronous CLF** removes CLF from the critical path of the application, such that CLF overhead is hidden. Asynchronous CLF can be implemented by a helper thread that performs CLF in parallel with application execution [17]. The effectiveness of asynchronous CLF depends on workload characteristics: if the time interval between CLF and the next memory fence is too short, then asynchronous CLF is not effective, and exposed to the critical path.

**Deferred CLF** relaxes the constraints of write ordering to improve performance. This method groups data modifications into failure-atomic intervals and delays CLF to the end of each interval. This method ensures data consistency across intervals. Once the system crashes, all or none of the data modifications in the interval become visible. The existing studies determine the interval length based on either a user-defined value [10, 40] or application semantics [7].

<sup>1</sup>We distinguish cache line and cache block in the paper. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line.



**Figure 2: The overhead of CLF in common PM-aware applications.**

**Passive CLF** relies on natural cache eviction from the cache hierarchy to persist data. Lazy persistence [2] is one such optimization. With passive CLF, the system itself does not trigger CLF. Dirty data is written back to PM, depending on the hardware eviction. In the event of system failure, the system uses checksums to detect inconsistent data and recovers the program by recomputing inconsistent data. Lazy persistency trades CLF overhead with recovery overhead.

**Bypassing CLF** avoids storing modified data in the cache hierarchy and, instead, writing to PM directly [16, 60]. Specific non-temporal instructions on x86-64 architecture (e.g., `movnti` and `movntdq`) provide such support. Still, fence instructions are used to ensure the update is persisted. Bypassing CLF could avoid the overhead in cache and CLF instructions to gain performance if there is little data reuse in the cache [56].

Most of existing efforts focus on the CLF policy, i.e., when to use CLF or how to avoid CLF. However, there is a lack of study to improve the CLF mechanism itself, and the performance characterization of CLF on PM hardware remains to be studied, which is the focus of this paper.

## 3 PERFORMANCE ANALYSIS OF CLF

We use the Intel Optane PM hardware (specifications in Table 3) for the performance analysis.

**Overhead of CLF in PM-aware applications.** We quantify the cost of CLF in seven representative PM-aware applications. These applications are in-memory databases (Intel’s PMEMKV [21] and Redis [6]), PM-optimized index data structures (Fast&Fair [20] and Level-Hashing [63]), and multi-threaded C/C++ applications (Streamcluster, Canneal and Dedup) from Parsec [5] benchmark suite. These applications rely on various persistency semantics and fault models to enable crash consistency, but all use the CLF mechanism. Table 4 summarizes the applications. For Parsec applications, we use the *native* input problem and report execution time. For other workloads, we run *dbench* to perform *randomfill* operations and report system throughput. Figure 2 shows the CLF overhead in each benchmark in the hatched bars.

The results highlight the impact of CLF on these PM-aware workloads. For all workloads, CLF significantly affects the performance by 24%-62%. Redis shows the highest performance loss because relies on frequent CLF to persist data objects and logs to implement database transactions. The high overhead in PM-aware workloads motivates our work to optimize the performance of the CLF mechanism.

**The performance impact of CLF concurrency.** We increase the number of threads to perform CLF and measure the performance

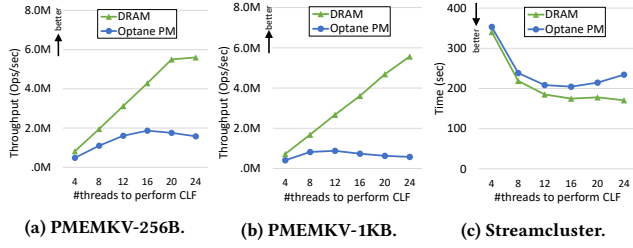


Figure 3: Performance at increased numbers of threads performing CLF.

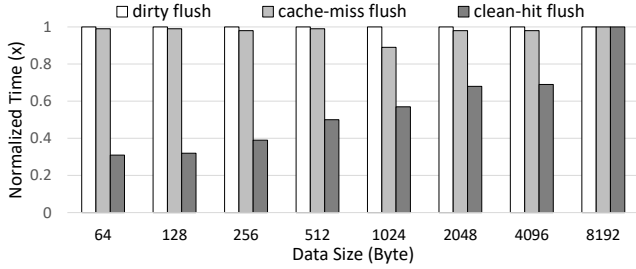


Figure 4: Performance of flushing cache lines in different status.

of PMEMKV and Streamcluster on DRAM and Optane, respectively. Table 4 in Section 6.1 provides more details of the workloads. For PMEMKV, the key size is 20 bytes, and the value size is 256 bytes (Figure 3a) and 1 KB (Figure 3b). Figure 3c reports Streamcluster performance.

On Optane PM (Figure 3), all workloads reach their peak performance at a small number of threads, and then the performance starts degrading. In contrast, performance on DRAM sustains scaling as the concurrency increases. Optane shows lower scalability than DRAM because the contention at the internal buffer of Optane and the WPQ in iMC. The increasing performance gap between DRAM and Optane at a large number of threads reveals that high frequency of CLF exacerbates the scaling limitation.

We identify two optimization directions to improve CLF performance. First, the adaption in CLF concurrency should be bi-directional. At a low concurrency level, there is no sufficient write-back traffic to exploit memory bandwidth so that PM is underutilized. In this scenario, increasing the concurrency to flush cache lines becomes essential. At a high concurrency level, PM cannot cope with high CLF rate at the application level, and concurrency throttling becomes critical. Given the above two optimization directions, the challenges remain in how to efficiently and timely detect whether PM is under- or over-utilized? Furthermore, what is the appropriate concurrency level?

Second, different workload characteristics, such as the value size in key-value stores and query intensity, could lead to different concurrency peak. For instance, in PMEMKV, using the 1 KB value size in Figure 3b reaches the peak point using 12 threads, while using the 256-byte value size in Figure 3a reaches the peak point using 16 threads. The different concurrency peaks necessitate a dynamic solution that enables flexible controlling of CLF concurrency.

Table 1: Average dirtiness of flushed cache lines.

Workloads	YCSB							TPC-C
	Load	A	B	C	D	E	F	
Dirtiness	0.43	0.55	0.56	0	0.51	0.51	0.47	0.32

**The performance impact of cache lines status.** We develop micro-benchmarks to persist data objects of various sizes. Also, we control the locality and dirtiness of flushed cache blocks of those data objects, in order to measure the cost of flushing dirty (resident) cache lines, non-resident cache lines, and clean resident cache lines. Figure 4 presents the measured overhead of these three CLF cases.

At a small data size, e.g., 64-byte, flushing a clean cache line resident in the cache hierarchy is significantly cheaper (3.3x) than flushing a dirty cache line. Such low overhead is because of reduced overhead in cache coherence directory lookup, and also because of the elimination of writeback traffic. As a comparison, when flushing a cache line that has been evicted from the cache hierarchy, i.e., non-resident, the cost is much higher than flushing a resident cache line. The difference between a dirty flush and a cache-miss flush indicates the cost of looking up the whole cache coherence directory in our machine is high and overweighs the benefit of eliminated writeback.

The low cost of flushing a clean resident cache line motivates us to design a *proactive flushing* mechanism to ‘transform’ dirty or non-resident flushing into clean-hit flushing ahead of time. The key idea is to complete the transformation before the latency of CLF is exposed to the critical path.

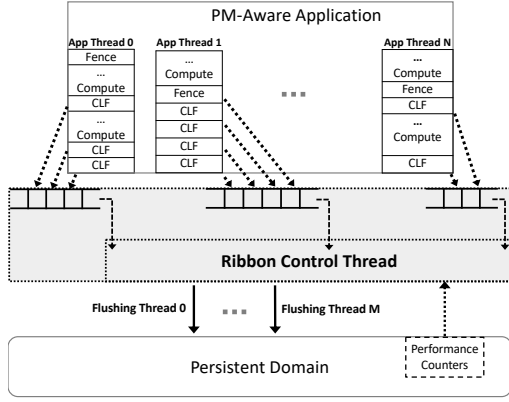
**Dirtiness of flushed cache lines.** We quantify the average dirtiness of flushed cache lines, denoted as  $R_{db}$ , as the ratio between the modified bytes and the cache line size. Therefore, a workload with  $R_{db}$  cache line dirtiness would waste  $(1 - R_{db})$  bandwidth from the cache hierarchy to the memory subsystem. Moreover, write amplification inside the PM hardware buffer may further increase the number of clean bytes written back to PM. For instance, if only one byte in four consecutive cache lines is updated, 256 bytes will be eventually written to Optane PM, because the internal transactions have a granularity of 256 bytes. Table 1 shows the results for running YCSB [12] and TPC-C [32] workloads against Redis. In general, the dirtiness is less than 0.6 in all workloads, indicating more than half memory bandwidth is wasted for writing back clean data to PM. Thus, improving cache line dirtiness could benefit CLF performance on such PM hardware.

## 4 DESIGN

We design Ribbon to accelerate the CLF mechanism in PM-aware applications without impacting program correctness and crash recovery. Ribbon decouples the concurrency control of CLF from the application. It also proactively transforms cache lines to clean status. It uses CLF coalescing, an application-specific optimization for workloads that exhibit low dirtiness in flushed cache lines.

### 4.1 Decoupled Concurrency Control of CLF

Ribbon decouples CLF from the application and adjusts the level of CLF concurrency (the number of threads performing CLF) adaptively. Ribbon throttles CLF concurrency if contention on PM devices is detected. Conversely, it ramps up CLF concurrency when



**Figure 5: Ribbon decouples CLF from the application to its control thread. By detecting contention or underutilization on PM, Ribbon changes the number of flushing threads to adapt the CLF concurrency.**

PM bandwidth is underutilized. We illustrate the workflow in Figure 5.

**CLF Decoupling** The decoupling design in Ribbon creates a thin layer (the gray box in Figure 5) between the application and PM. CLF and fence instructions from the application, such as `clwb`, `clflushopt`, `clflush`, and `sfence`, are collected and queued in this layer. Ribbon uses a group of *flushing threads* to execute these intercepted instructions, respecting the order between flush and fence instructions as in the program order. Therefore, the sequence of flush and fence is unchanged, and consistent semantics is preserved. Furthermore, Ribbon can adapt the CLF concurrency by changing the number of flushing threads.

Ribbon uses FIFO queues as a coordination mechanism between the application and flushing threads. Each application thread has a private FIFO queue, while one flushing thread may work with multiple FIFO queues. CLFs from an application thread are enqueued at the head of its queue. At the queue tail, a flushing thread dequeues and executes CLFs. Ribbon uses a circular buffer to implement the queue, and only exchanges two integers, i.e., the head and tail indexes, among threads to have a lock-less queue implementation. Synchronization between the threads is rare because, on each queue, the application thread only updates the head and the flushing threads only update the tail.

Assume there are  $N$  application threads and  $M$  flushing threads. Each flushing thread handles at most  $\lfloor N/M \rfloor + 1$  application threads (queues). Ribbon throttles the CLF concurrency by reducing  $M$  to be  $M < N$ . Conversely, increasing  $M$  to  $M > N$  would increase the CLF concurrency. Separately, a control thread detects performance bottlenecks in PM and adjusts the number of flushing threads.

Ribbon ensures that the flushing threads execute CLF and fence instructions in the same order as in the application thread. Each memory fence instruction in the application thread acts as the deadline for the flushing threads to finish all CLFs issued before it. Therefore, CLFs after a fence cannot be executed until CLFs before the fence are cleared from the queue. When an application thread issues a memory fence instruction, but there are pending CLF requests in the queue, Ribbon blocks the application thread.

This interaction is essential for throttling the CLF concurrency and ensuring program correctness, i.e., reducing the draining rate of CLFs from the queue, without overflowing the queue.

**Determining the concurrency level of CLF.** A *control thread* monitors the traffic to PM and adjusts the concurrency level of CLF ( $NUM_{thr}$ ) at runtime.

The control thread monitors hardware counters in PM at interval  $T$  to track the write bandwidth to PM DIMMs ( $BW_{pmm}$ ). System evaluation shows that when the concurrency level increases, the bandwidth to PM first increases to a peak and then starts decreasing [25, 43, 56].  $BW_{pmm}$  reflects the speed at which the memory controller drains write requests from the WPQ. When memory contention occurs in the WPQ, reducing the concurrency level would improve  $BW_{pmm}$ . We call the concurrency levels below the one that reaches the peak performance to be the *scaling region* and above to be the *contention region*. The control thread samples  $BW_{pmm}$  at four concurrency points to estimate  $NUM_{thr}$  for achieving the peak  $BW_{pmm}$ .

The control thread first samples the bandwidth at the concurrency level  $P1$  which is equal to the number of flushing threads that saturate bandwidth on hardware.  $P1$  is architecture-dependent and on the Optane PM, system evaluation reveals that the peak write bandwidth is achieved at four threads [25]. Therefore,  $1-P1$  threads in PM-aware workloads have to be in the scaling region. The control thread records the bandwidth to PM at  $P1$  to be  $BW_1^{pmm}$ . Then, it chooses a sample point at the number of cores ( $P4$ ) and measures  $BW_4^{pmm}$ . On our PM hardware,  $P4$  is equal to 24. Next, samples are taken at  $P2 = P1 + 1$  and  $P3 = P4 - 1$ , namely  $BW_2^{pmm}$  and  $BW_3^{pmm}$ . If  $BW_2^{pmm}$  is higher than  $BW_1^{pmm}$ , and  $BW_4^{pmm}$  is also higher than  $BW_3^{pmm}$ , it means that even the maximum parallelism has not reached the contention region. Thus, the control thread selects  $NUM_{thr}$  to be  $P4$ . If  $BW_2^{pmm}$  is higher than  $BW_1^{pmm}$ , but  $BW_4^{pmm}$  is lower than  $BW_3^{pmm}$ , it means that the peak is between  $P2$  and  $P3$ . The control thread sets  $NUM_{thr}$  to be the intersection between the two lines connecting  $P1$  to  $P2$  and  $P3$  to  $P4$ , respectively. Finally, if  $BW_2^{pmm}$  is lower than  $BW_1^{pmm}$ , and  $BW_4^{pmm}$  is also lower than  $BW_3^{pmm}$ , the control thread selects  $NUM_{thr}$  to be  $P1$ . In practice, the number of flushing threads is subject to the number of idle threads, and contemporary many-core platforms can provide abundant thread-level parallelism. If there are no enough idle threads to support  $NUM_{thr}$  flushing threads, Ribbon automatically disables concurrency control and regresses to use application threads to perform CLF.

We sweep all levels of CLF concurrency in all evaluated workloads and find that this algorithm can always determine the optimal concurrency level. Figure 6 reports all workloads (except one phase in Streamcluster) exhibit a similar trend, i.e., reaching a peak at a low concurrency level and then decreasing performance as concurrency increases. The dashed line and the intersection illustrate the optimal concurrency level for PMEMKV. Streamcluster contains two phases of  $BW_{pmm}$ . The first phase follows the scaling trend of other applications in Figure 6. In the second phase (shown in Figure 7), Streamcluster does not enter the contention as its bandwidth continues increasing. The control thread determines  $NUM_{thr}$  to be the maximum available concurrency.

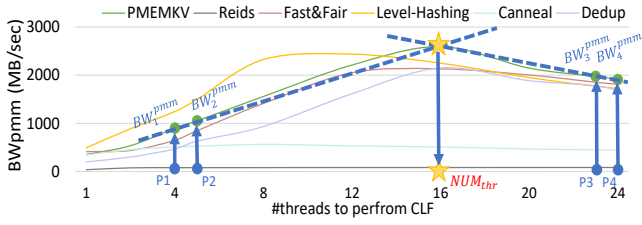


Figure 6: PM bandwidth when running benchmarks with various numbers of flushing threads. The number of application threads is 24.

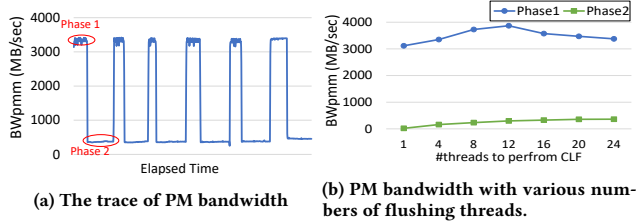


Figure 7: PM bandwidth of Streamcluster. The number of application threads is 24.

The control thread repeats the above procedure of determining concurrency level of CLF, if the variation of  $BW_{pmm}$  is higher than a threshold, indicating there is a change in execution phases of the application and there is a need to adjust concurrency level. Based on our study, the variation threshold should be set between 20% and 30% of  $BW_{pmm}$  for best performance. If the threshold is too low (e.g., less than 20%), Ribbon triggers concurrency throttling frequently, which causes performance loss. If the threshold is too high (e.g., more than 30%), Ribbon cannot timely capture the change of execution phases, which loses opportunities for performance improvement. We use 20% in Ribbon; We study the sensitivity of application performance to this parameter in Section 6.3.

The time interval  $T$  to track  $BW_{pmm}$  has impact on performance. On the one hand, if  $T$  is too large, infrequent monitoring may fail to capture bandwidth saturation. On the other hand, if  $T$  is too small, runtime overhead is large, thereby amortizing the performance benefit of concurrency control. We set  $T$  to one second in Ribbon to strike a balance between monitoring effectiveness and cost. We study the sensitivity of application performance to this parameter in Section 6.3.

## 4.2 Proactive Cache Line Flushing

Ribbon proactively flushes cache lines to transform cache lines to clean state. The proactive CLF increases the chance of flushing a clean cache line in the critical path of the application, which has lower latency than flushing a dirty cache line. We present the workflow in Figure 8.

Ribbon leverages the precise address sampling capability in hardware performance counters, e.g., Precise Event-Based Sampling (PEBS) from Intel processor or Instruction-based Sampling (IBS) from AMD processor, to collect the virtual memory addresses of store instructions. If a cache line is found to be updated recently,

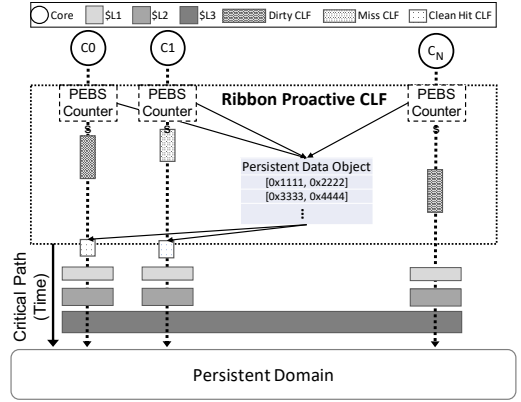


Figure 8: Proactive cache line flushing to improve performance.

Ribbon uses a thread to proactively issue a flush (the thread is named the proactive thread). Later on, when the application thread flushes the cache line, it is likely to be in clean status. Note that the cache block may have been evicted by hardware before the proactive thread flushes it. However, a redundant flush by the proactive thread has no impact on program correctness. This approach increases the probability of clean cache lines flushed by the application, which shortens the latency on the critical path.

The proactive CLF can slightly increase write traffic (see Section 6.3). For instance, if a cache block is written multiple times followed by one CLF in the program, using the proactive CLF may generate more than one CLF. To avoid the negative impact of extra write traffic due to the proactive CLF, Ribbon disables it once CLF concurrency is reduced because of reaching bandwidth bottleneck; The proactive CLF is re-enabled if CLF concurrency is increased.

Ribbon separates the proactive thread and flushing threads as two independent groups. The design is synchronization-free between the proactive thread and flushing threads. The design does not change which cache lines should be flushed. It also ensures that the consistency semantics in the program retains because no CLF is skipped due to the proactive CLF.

## 4.3 Coalescing Cache Line Flushing

We propose cache line coalescing as an application-specific optimization for workloads that exhibit low dirtiness in flushed cache lines. An Application is suitable for this optimization if multiple CLFs in the application meet two requirement: *First*, the multiple CLFs occur in proximity in time; *Second*, the flushed data objects are coalescable to fewer cache blocks. The first requirement ensures crash consistency after CLF coalescing. CLF coalescing delays those to-be-coalesced CLFs that happen early in the bundle of CLFs from being coalesced. However, if all CLFs in the bundle happen sequentially with no other non-coalescing CLFs occurring between these to-be-coalesced CLFs in the application, delaying the to-be-coalesced CLFs has no impact on crash consistency. The second requirement is the necessary condition to have potential performance benefits.

Listing 2 shows an example from Redis. Lines 8 and 12 use two CLFs for persisting *newVal* and *newKey*, respectively. Coalescing

Listing 2: An example of CLF coalescing

```

1 #define KEY_LEN 24
2 #define VALUE_LEN 100
3 /*The original code without coalescing*/
4 void setGenericCommand(client *c, char *key, char *
   val ...) { ...
5     TX_BEGIN(server.pm_pool) {
6         char* newVal = alloc_mem(VALUE_LEN);
7         dupStringObjectPM(newVal, val);
8         flush(newVal, VALUE_LEN);
9         _mm_sfence();
10        char* newKey = alloc_mem(KEY_LEN);
11        setKeyPM(c->db, key, newKey, newVal);
12        flush(newKey, KEY_LEN);
13        _mm_sfence();
14    } TX_ONABORT { ... } TX_END ... }
15
16 /*The code with coalescing*/
17 void setGenericCommand_coalescing(client *c, char *
   key, char *val ...) { ...
18     TX_BEGIN(server.pm_pool) {
19         char* mem = alloc_mem(VALUE_LEN + KEY_LEN);
20         char* newVal = get_mem(0);
21         dupStringObjectPM(newVal, val);
22         char* newKey = get_mem(VALUE_LEN);
23         setKeyPM(c->db, key, newKey, newVal);
24         flush(mem, VALUE_LEN + KEY_LEN);
25         _mm_sfence();
26     } TX_ONABORT { ... } TX_END ... }

```

these CLFs will delay the first CLF. Between these two CLFs, there are no other CLF. Therefore, the delay of the first CLF still maintains execution correctness after a restart, i.e., the two CLFs either both succeed or fail, which is consistent with the original execution. After the coalescing, the situation that the first CLF succeeds but the second one fails is impossible, guaranteeing the consistency.

After examining PM-aware applications in Table 4, we find that in-memory databases, such as PMEMKV and Redis, and customized PM data indexes, such as Fast&Fair (B+-tree) and Level-Hashing, are prone to the low dirtiness. Parallel computing codes, such as streamcluster, canaal, and dedup from Pasesc, often do not have the low dirtiness. Furthermore, we find **unaligned CLF** and **uncoordinated CLF** are the main reasons for low dirtiness in flushed cache lines.

The unaligned CLF happens when a persistent data object is unaligned with cache lines. For example, a persistent data object is 100 bytes. Ideally, the object should use two cache blocks of 64 bytes. However, the object may be unaligned at the memory allocation and ended up occupying three cache blocks. Once the object is updated, three cache blocks, i.e., 192 bytes, have to be flushed, increasing the number of CLF by 50%. Uncoordinated CLFs happen when multiple associated data objects are allocated into separate cache blocks. Here, data objects are associated if they are always updated together. Therefore, coalescing them into the same cache blocks will reduce the number of CLFs.

Implementing cache line coalescing requires replacing memory allocation and combining cache line flushes and memory fences. This transformation could be done automatically by the compiler. In practice, we find that automatic conversion is challenging because even the same application logic can have different implementations in different applications. Without application knowledge, automatic

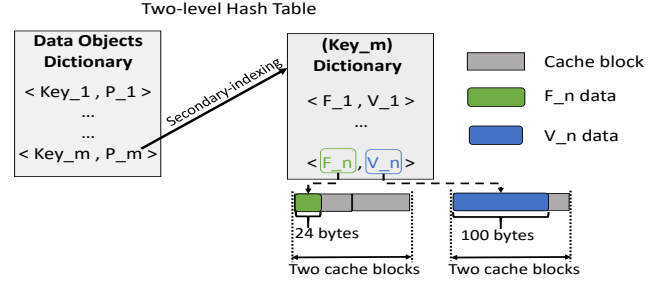


Figure 9: Uncoordinated cache-line flushing in the two-level hash table in Redis.

transformation is error-prone. Therefore, we provide a simple interface and leverage the programmer’s application knowledge in implementation.

The remainder of the section uses Redis as an example. We use a PM-aware version of Redis, i.e., Redis-libpmemobj [22]. As a key-value store system, Redis provides fast access to key-value pairs. Each key-value pair includes a unique key ID and their data (value). For each key-value pair, the key and value objects are allocated separately on different cache blocks. Figure 9 gives a case where the value object in a key-value pair is a complex data structure. This case comes from the secondary indexing in Redis. In this case, Redis updates the key (i.e.,  $F_n$  in Figure 9, which is the secondary-level key) and value (i.e.,  $V_n$  in Figure 9) together. Coalescing  $F_n$  and  $V_n$  objects into a fewer contiguous cache blocks reduces the number of CLFs.

To coalesce CLFs for Redis, we introduce a new memory allocation mechanism. The old implementation in Redis-libpmemobj uses the memory allocation API from PMDK’s libpmemobj library, which does not consider semantics correlation between memory allocations (i.e., memory allocations for a pair of key and value). In the new implementation, we introduce a customized memory allocation API that takes an argument indicating whether the memory allocation is for a key or a value object. In the original implementation of Redis, the memory allocation for a value object happens before the memory allocation for the corresponding key object. Hence, if the memory allocation is for a value object, in our implementation of Redis, the memory allocation not only allocates memory for the value, but also for the key. The key and value objects are co-located into continuous cache blocks, which enables CLF coalescing. If the memory allocation is for a key object, no memory allocation happens, but the previously allocated memory for the key object is returned. Also, the new implementation attempts to avoid unaligned CLF.

#### 4.4 Impact of Ribbon on Program Correctness

PM-aware applications optimized with Ribbon maintain their program correctness because their fault models and recovery mechanisms remain unchanged. Ribbon does not eliminate any cache flush or fence instructions, nor changes their order in the original program. Thus, the original consistency semantics in these programs are preserved even in the presence of crashes. The advantage of Ribbon is to reduce the latency of these CLF instructions on the critical path by improving the bandwidth to PM or increasing

Table 2: Ribbon APIs

API name	Description
int ribbon_start(int numAppT, int eleFQueue) int ribbon_flush(void* addr, size_t len) void* ribbon_alloc(size_t len, int type) int ribbon_stop() int ribbon_fence() int ribbon_free(void* addr)	Initialize the runtime system and resource (e.g., flushing threads and FIFO queues)  Put CLF requests into flushing queues  Memory allocation for coalescing CLF  Terminate runtime and release resources Ensure all pending CLF requests are flushed Free a memory allocation

the probability of a clean cache line. Although the proactive CLF may introduce additional cache flushes, they do not occur on the critical path. Also, changing the state of cache lines has no impact on the fault model in these PM-aware applications because cache line eviction and replacement is hardware-managed and outside the application control. Coalescing multiple cache lines into one does not eliminate the flush and fence instructions in the program. However, it can reduce write amplification so that these instructions could complete at reduced latency. When a crash occurs, each program will be restored to a consistent state by employing its original recovery mechanism, e.g., undo/redo logging.

## 5 IMPLEMENTATION

**Programming APIs.** Ribbon is implemented as a user-level library to provide CLF performance optimization. Ribbon provides a small set of APIs and is designed to minimize the porting efforts in existing PM-aware applications and libraries, such as Intel PMDK [23], Mnemosyne [53], and NVthreads [19]. Table 2 summarizes main APIs.

*ribbon\_start()* initializes the flushing threads, control thread and proactive CLF thread. This routine creates a pool of flushing threads and FIFO queues, and initializes performance counters. This routine is called only once before main execution phase starts. *ribbon\_stop()* frees all runtime resources created in *ribbon\_start()*. This routine is called only once before the end of the main program. *ribbon\_flush()* and *ribbon\_fence* are used to intercept cache flush and memory fence calls in the program. *ribbon\_flush()* places a CLF request at the head of the private FIFO queue of the issuing thread. *ribbon\_fence()* checks if all pending requests in the FIFO queue have drained. If not, Ribbon blocks the application thread. *ribbon\_alloc()* and *ribbon\_free()* are used to replace the memory allocation and free APIs in the *pmemobj* library in Redis. The two APIs are used to allocate and free memory from/to PM for coalescing CLF.

Using the above APIs to replace CLF and memory fence can be done automatically by a compiler. To enable CLF coalescing in Redis, we make modifications manually. The statistics of code modification given by git diff is: 10 files changed, 293 insertions(+), 64 deletions(-).

**System optimization.** Ribbon includes several optimization techniques to enable high performance. We use FIFO queues to coordinate between the application thread and flushing threads. When the number of flushing threads is more than the number of application threads, multiple flushing threads fetch CLF requests

Table 3: Experiment Platform Specifications

Processor	2 <sup>nd</sup> Gen Intel® Xeon® Scalable processor
Cores	2.4 GHz (3.9 GHz Turbo frequency) × 24 cores (48 HT)
L1-icache	private, 32 KB, 8-way set associative, write-back
L1-dcache	private, 32 KB, 8-way set associative, write-back
L2-cache	private, 1MB, 16-way set associative, write-back
L3-Cache	shared, 35.75 MB, 11-way set associative, non-inclusive write-back
DRAM	16-GB DDR4 DIMM x 6 per socket
PM	128-GB Optane DC NVDIMM x 6 per socket
Interconnect	Intel® UPI at 10.4 GT/s, 10.4GT/s, and 9.6 GT/s

Table 4: A summary of evaluated workloads

Application	Program Type	PM Access Layer
PMEMKV	Database	Library/PMDK(undo&redo)
Redis	Database	Library/PMDK(undo&redo)
Fast&Fair (B+-tree)	PM-aware index	Native (add custom assembly instructions)
Level-Hashing	PM-aware index	Native (add custom assembly instructions)
Streamcluster	Lock-based parallel code	Library/NVthreads (redo)
Canneal	Lock-based parallel code	Library/NVthreads (redo)
Dedup	Lock-based parallel code	Library/NVthreads (redo)

from one FIFO queue, which raise contention. To avoid the contention, we dedicate one flushing thread to fetch CLF requests from the queue and then assigns them to other flushing threads. Our implementation uses the most recent *clwb* instruction to flush cache blocks.

## 6 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of Ribbon.

### 6.1 Methodology

**Experiment platform.** We evaluate Ribbon on the Intel Optane persistent memory. Table 3 describes the configuration of the testbed. The system consists of two sockets, each with two integrated memory controllers (iMCs) and six memory channels. Each DRAM DIMM has 16 GB capacity while a PM DIMM has 128 GB capacity. In total, the system has 192 GB DRAM, and 1.5 TB Intel Optane DC persistent memory. We use one socket for performance study to eliminate NUMA effects. The persistent domain starts from iMC, i.e., a memory fence only returns after the flushed data has reached iMC.

**Applications with various PM access interfaces.** We select seven representative PM-aware workloads from diverse domains, including in-memory database (PMEMKV [21] and Redis [1]), PM-aware index data structures (Fast&Fair [20] and Level-Hashing [63]) and C++ parallel computing applications (Streamcluster, Canneal, and Dedup from Parsec benchmark suite [5]). For PMEMKV, we use its *cmap* storage engine.

These applications also use different interfaces to access PM, such as high-level PM-aware libraries and native direct interaction. Table 4 summarizes the application characteristics and PM access interfaces for each workload. PMEMKV and Redis use *libpmemobj* from Intel PMDK [23] library to access and persist data. *libpmemobj* is a logging-based transaction system, which implements undo logging to protect user data and redo logging to protect metadata. The Parsec applications guarantee data consistency by the NVthreads library [19]. NVthreads supports a redo-logging for multi-threaded

C/C++ programs. The two PM-aware index data structures use custom assembly instructions to flush data from the cache to PM, and add fences to ensure the order between these flushes and other application accesses to the data.

## 6.2 Overall Performance

We evaluate each workload at a low and high thread-level parallelism (using 4 and 24 application threads respectively). For Redis, we cannot change the number of threads to run it, because it is a single-thread server; To evaluate Redis, we change the number of client threads (using 4 and 24). PMEMKV, Redis, Fast&Fair, and Level-Hashing run the *dbench* benchmark to execute one hundred million *randomfill* operations. The key size is 20 bytes, and three value sizes (256 bytes, 1 KB, and 4 KB) are tested. Streamcluster, Canneal, and Dedup use the *Native* input problem in [19].

Ribbon demonstrates its generality in these PM-aware frameworks that employ different fault models, recovery mechanisms, and interfaces to access PM. Ribbon achieves performance improvement in all seven workloads at different application concurrency, without changing any CLF policy. Figures 10 and 11 present the performance of Ribbon (w. *cc+pclf*) in comparison to the original implementation (*baseline*). At four application threads, Ribbon increases the concurrency of CLF and achieves up to 17.6% improvement (9.3% on average). In contrast, at 24 application threads, Ribbon detects memory contention and improves the performance by up to 49.8% (20.2% on average).

Ribbon brings performance benefits to all tested workloads. Among them, Ribbon delivers more performance benefits to those that use large value sizes (1 KB and 4 KB in our evaluation) and high application threads concurrency (24 application threads in our evaluation). These cases can result in memory contention or lack of CLF parallelism, which provides more opportunity to Ribbon.

We analyze the effectiveness of each optimization technique by breaking down their contribution to performance improvement. In particular, we apply the concurrency control first (w. *cc*) and measure performance improvement. Then, on top of it we apply the proactive CLF (w. *pclf*) and measure performance improvement. Figures 12 and 13 presents the breakdown with four and 24 application threads.

We find that the concurrency control and proactive CLF contribute comparably to the performance improvement at a low number of application threads (Figures 12). At a large number of application threads, most performance improvement attribute to the concurrency control technique (Figure 13). The difference is because the contention on PM devices increases when CLFs are issued by more threads, which compete in inserting flushed data to WPQ (the start of the persistent domain). Therefore, CLF tends to create a performance bottleneck at a large number of application threads, which is addressed by the concurrency control. Note that Redis also benefits substantially from the proactive CLF even at the high number of client threads because it is a single-threaded server, and CLF contention is not its bottleneck.

## 6.3 Sensitivity Evaluation

We use Streamcluster with *Native* input problem for sensitivity study because this workload has execution phases with various

**Table 5: Sensitivity study on bandwidth variance threshold and monitor interval (App threads = 24).**

BW Variance Threshold	Improvement	Interval (sec)	Improvement
10%	7.4%	0.1	9.7%
20%	16.5%	1	16.5%
30%	17.3%	5	13.8%
50%	14.6%	10	11.3%

(a)

(b)

**Table 6: Sensitivity study on proactive CLF**

#app threads	1	2	4	8	12	16	17-24
Improvement	5.4%	6.6%	7.5%	6.3%	4%	2.1%	0
Normalized BW cost	6.9%	6.3%	5.6%	7.2%	8.4%	8.9%	0

bandwidth consumption, imposing challenges on concurrency control and proactive CLF.

**Sensitivity on bandwidth variance threshold.** We use four thresholds for study. Table V(a) shows the results and the tradeoff between low and high threshold values. 20%-30% leads to the largest improvement (Ribbon uses 20%).

**Sensitivity on monitor interval  $T$ .** We use four intervals for study. Table V(b) shows the improvement achieved at various interval values. The highest improvement is achieved at one second. (Ribbon uses one second for  $T$ ).

**Sensitivity on proactive CLF.** We evaluate how the proactive CLF responses given various bandwidth consumption of the application. Ribbon should avoid the negative impact of the proactive CLF on memory bandwidth. To evaluate the proactive CLF itself, we disable the concurrency control, but integrate the algorithm of determining concurrency level into the proactive CLF to detect bandwidth contention. When the concurrency level needs to be reduced according to the algorithm, we do not change concurrency but disable the proactive CLF. We sweep the number of application threads from one to 24. We report the bandwidth consumption of the proactive CLF normalized to the total bandwidth consumption in Table 6. We report application performance normalized to that without the proactive CLF.

The proactive CLF improves the performance by 2.1%-7.5% when the number of applications increases from one to 16. In these cases, the proactive CLF takes a small portion (5.6%-8.9%) of the total bandwidth consumption. When the application uses more than 16 application threads, the proactive CLF is disabled because of the detection of bandwidth contention. As a result, there is no performance improvement.

## 6.4 Heavily Loaded System Evaluation

We evaluate Ribbon on a heavily loaded machine to understand the impact of Ribbon on application performance. In this evaluation, we co-run three different application combinations. For each combination, we run two applications, each using 24 application threads. PMEMKV, Level-Hashing, and Fast&Fair run *dbench* to execute one hundred million *randomfill* operations and use 256 bytes as the value size. Streamcluster and Canneal use the *native* input problem. We report the experimental results in Figure 14.

We observe that all workloads can benefit from Ribbon significantly. Compared with the system without Ribbon, Ribbon improves

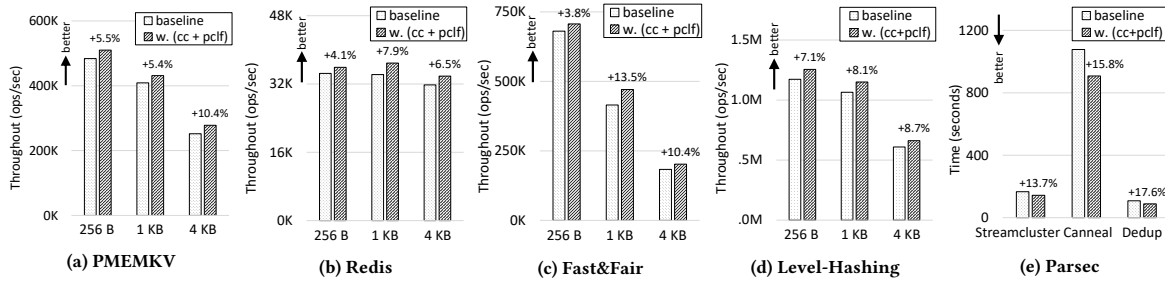


Figure 10: Overall performance (App threads = 4).

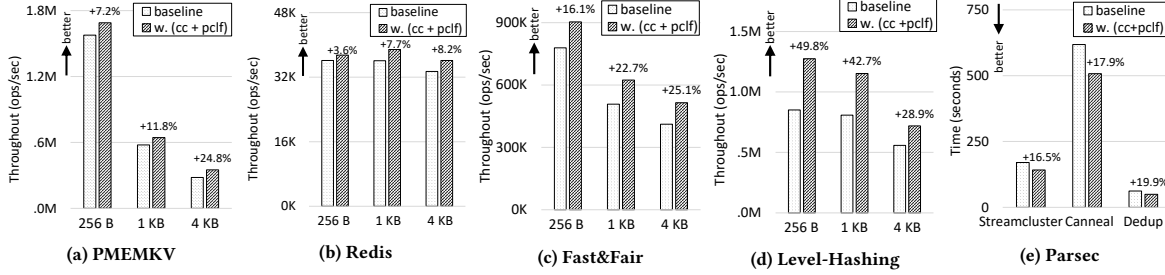


Figure 11: Overall performance (App threads = 24).

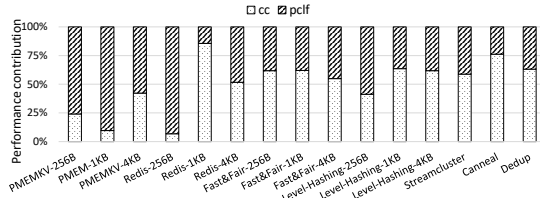


Figure 12: A breakdown of performance improvement from the concurrency control and proactive CLF (App threads = 4).

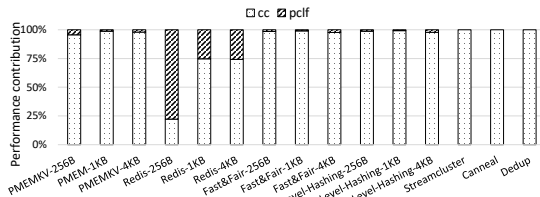


Figure 13: A breakdown of performance improvement from the concurrency control and proactive CLF (App threads = 24).

the performance of PMEMKV and Streamcluster by 20.4% and 27.7%, respectively. When Level-Hashing and Canneal co-run on the same machine, Ribbon speeds up the two applications by 17.3% and 13.9%, respectively. Fast&Fair and PMEMKV co-run achieve the most improvement from Ribbon, reaching 45.2% and 25.6% improvement, respectively. When multiple applications share a machine, Ribbon predicts the optimal system-wide CLF concurrency according to the method described in Section 4.1. Ribbon decides the number of

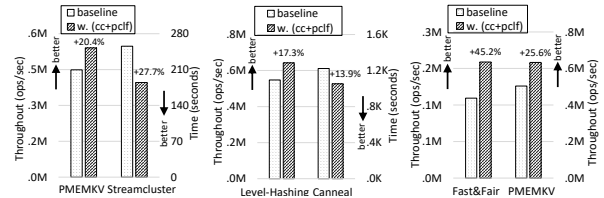


Figure 14: Heavily loaded system

flushing threads for each application based on the CLF throughput ratio of the two applications.

## 6.5 Coalescing of Cache Line Flushing

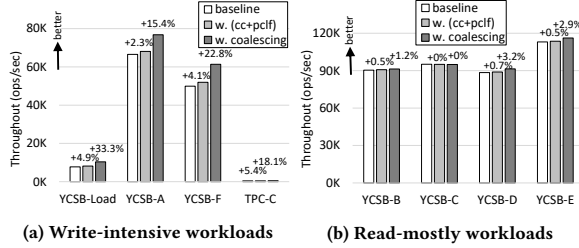
We evaluate the effectiveness of CLF coalescing in Redis running YCSB [13] and TPC-C [32] benchmarks. For YCSB, we use its default configuration. The key and value sizes are 24 bytes and 100 bytes, respectively. We run 24 clients threads.

Our first evaluation compares the dirtiness of flushed cache lines with and without CLF coalescing. Table 7 presents the results. The baseline version results in 0.32 to 0.56 cache line dirtiness in tested workloads, except for the read-only workloads (YCSB-C). After the optimization, the cache line dirtiness is increased to 0.4-0.68. For each workload, the coalescing effectively reduces traffic and CLF by 20%-45%.

We quantify the impact of the improved cache line dirtiness on the overall performance, as reported in Figure 15. The increased dirtiness results in 18% to 33% performance improvement (w. coalescing) for write-intensive workloads. For the read-mostly workloads, performance improvements are less than write-based workloads, because these read-only workloads generate far less write traffic.

**Table 7: Quantify the dirtiness of flushed cache lines in Redis.**

Workloads	YCSB							TPC-C
	Load	A	B	C	D	E	F	
<b>w.o coalescing</b>	0.43	0.55	0.56	0	0.51	0.51	0.47	0.32
<b>w. coalescing</b>	0.62	0.66	0.67	0	0.63	0.63	0.68	0.40



**Figure 15: The performance improvement by the CLF coalescing.**

We also compare the CLF coalescing technology with the combination of the concurrency control and proactive CLF (*w. cc+pcif*). We observe that the CLF coalescing achieves 5.3x higher performance improvements than the combination. This performance improvement highlights the effectiveness of the CLF coalescing.

## 7 RELATED WORK

**Persistency models** have been proposed to characterize and direct CLF. Pelley et al. [42] introduce strict and relaxed persistency and consider persistency models as an extension to memory consistency model. They propose strict, epoch, and strand persistency models and provide a persistent queue implementation. Other works [15, 24, 30, 52, 62] propose various optimizations to relax the constraints on persistency ordering. Ribbon is generally applicable to various persistency models.

**CLF-oriented optimizations.** Lazy Persistency [2] avoids eager cache flushing and relies on natural eviction from the cache hierarchy to persist data. Their solution detects persistency failures by calculating the checksum of each persistency region. This approach trades off rare persistency failure with a complex recovery procedure. NV-Tree [57] quantifies that CLF causes over 90% persistency cost in persistent B+-tree data structure. They propose to decouple tree leaves from internal nodes and only maintain the persistency of leaf nodes. In-cacheline log [10] supports fine-grained checkpointing that writes the cache hierarchy to PM at the beginning of each epoch. They place undo log and its logged data structure in the same cache line to reduce CLF. Link-free and soft algorithms [64] implement a durable concurrent set that only persists set members but avoids persisting pointers to eliminate unnecessary CLF. Software Cache [36] implements a resizable cache to combine writebacks and reduce CLF. Hardware modifications in the cache hierarchy and new instructions [39, 49] are also proposed to reduce the latency of CLF. Also, some cache designs use (relaxed) non-volatile memories [44, 50, 51], which naturally eliminates CLF.

Many other efforts that use CLF to enable crash consistency provide solutions in PM-aware programming models [7, 19], language-level persistency [18, 29]. Our solution is generally applicable to most of the existing software interfaces as their building block relies

on CLF. Unlike hardware-based solutions, we do not change hardware. We use commonly available hardware counters on existing architectures. We summarize software and hardware-based solutions, as well as optimizations for concurrency controls as follows.

**System software**, such as file systems PMFS [15] and BPFS [11], introduce a buffered epoch persistency model. Persistent operations within an epoch can be reordered to improve the persistency concurrency, while orders of persists across epochs are enforced. SCMFS [54] and NOVA [55] are PM-aware file systems with failure-atomic, scalability optimizations.

**Libraries**, such as Mnemosyne [52] and NV-Heaps [9], support programmer’s annotation of persistent data structures. Mnemosyne [52] keeps a per-thread log for improving concurrency and uses streaming writes to PM. NV-Heaps [9] provides type-safe pointers and garbage collection for failure atomicity on PM. Kamino-tx [38] and Intel’s PMDK [23] enable transactional updates to PM.

**Hardware-based solutions** extend existing instruction sets [26, 31, 48], modify cache hierarchy or add new interfaces to memory subsystems [27, 62], to provide low-overhead support for crash consistency on PM. Recently, works that rely on a hybrid of DRAM and PM memory subsystem [37, 41, 47] to speedup logging into DRAM and persist later to PM off the critical path.

**Concurrency control** has been studied on GPU and CPU to improve performance. Kayiran et al. [28] propose a mechanism to balance the system-wide memory and interconnect congestion and dynamically decide the level of GPU concurrency. Li et al. [33] reduce thread-level parallelism to mitigate page thrashing, which brings significant pressure on memory management, on Unified Memory. On the Optane architecture, Yang et al. [56] identify contention on a single DIMM, when a large number of threads access it. Their work proposes using non-interleaved memory mapping onto PM and binds each DIMM to a specific thread to avoid contention. Our approach requires no modification in virtual memory mapping and can dynamically adjust concurrency without statically binding NVDIMMs to threads. Curtis-Maury et al. [14] and Li et al. [34, 35] use performance models to select thread-level or process-level concurrency for best performance on CPU. Our design does not use performance models because and provides focused guidance on CLF.

## 8 CONCLUSIONS

CLF is critical for ensuring data consistency in persistent memory. It is a building block for many PM-aware applications and systems. However, the high overhead of CLF creates a new “memory wall” unseen in the traditional volatile memory. We analyze the performance of CLF in diverse PM-aware workloads on PM hardware. We design and implement Ribbon to optimize CLF mechanisms through a decoupled concurrency control and proactive CLF to change cache line status. Ribbon also uses cache line coalescing as an application-specific solution for those with low dirtiness in flushed cache lines, achieving an average 13.9% improvement (up to 33.3%). For a variety of workloads, Ribbon achieves up to 49.8% improvement (14.8% on average) of the performance.

## ACKNOWLEDGMENT

We thank Intel and the anonymous reviewers for their constructive comments. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF1718194). This research was supported by the Exascale Computing Project (17-SC-20-SC). LLNL-CONF-808913.

## REFERENCES

- [1] 2019. Redis. <http://redis.io/>.
- [2] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy Persistence: A High-Performing and Write-Efficient Software Persistence Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*.
- [3] Joy Arulraj, Andrew Pavlo, and Subramanya R Dullloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.
- [4] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. 2013. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. ACM, 4.
- [5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. <https://doi.org/10.1145/1454115.1454128>
- [6] J.L. Carlson. 2013. Redis in Action. In *Manning Publications: Greenwich*.
- [7] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 433–452.
- [8] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
- [9] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2012. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 47, 4 (2012), 105–118.
- [10] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 441–454.
- [11] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.
- [12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.
- [13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
- [14] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*.
- [15] Subramanya R Dullloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 15.
- [16] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. 2018. NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*.
- [17] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrap: A Lightweight Framework for Transactional Support of Storage Class Memory. In *2015 31st Symposium on Mass Storage Systems and Technologies*.
- [18] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. 2018. Persistence for synchronization-free regions. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 46–61.
- [19] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 468–482.
- [20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. 187–200.
- [21] Intel. [n.d.]. Key/Value Datastore for Persistent Memory. <https://github.com/pmem/pmemkv>.
- [22] Intel. [n.d.]. Redis, enhanced to use Persistent Memory - limited prototype. <https://github.com/pmem/redis/tree/3.2-nvml>.
- [23] Intel. 2014. Persistent Memory Development Kit. <http://pmem.io>
- [24] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 427–442.
- [25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amiraman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dullloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019).
- [26] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. [n.d.]. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*.
- [27] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 361–372.
- [28] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. 2014. Managing GPU concurrency in heterogeneous architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 114–126.
- [29] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Language-level persistence. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 481–493.
- [30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. *ACM SIGPLAN Notices* 51, 4 (2016), 399–411.
- [31] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [32] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*.
- [33] Chen Li, Rachata Ausavarungnirun, Christopher J Rosbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 49–63.
- [34] Dong Li, Bronis de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. 2010. Hybrid MPI/OpenMP Power-Aware Computing. In *International Parallel and Distributed Processing Symposium*.
- [35] Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis de Supinski, and Martin Schulz. 2010. Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems. In *International Parallel and Distributed Processing Symposium*.
- [36] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan. 2017. Adaptive Software Caching for Efficient NVRAM Data Persistence. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [37] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 329–343.
- [38] Amiraman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnathan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 499–512.
- [39] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 135–148.
- [40] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dali: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*.
- [41] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 371–386.
- [42] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistence. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*.
- [43] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM. <https://doi.org/10.1145/3357526.3357568>
- [44] Michelle Rasquinha, Dhruv Choudhary, Subho Chatterjee, Saibal Mukhopadhyay, and Sudhakar Yalamanchili. 2010. An energy efficient cache design using spin torque transfer (STT) RAM. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. ACM, 389–394.
- [45] Jie Ren, Kai Wu, and Dong Li. 2018. Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory. In *Proceedings of the Workshop on Memory Centric High Performance Computing (MCHPC'18)*.
- [46] Jie Ren, Kai Wu, and Dong Li. 2020. Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures. In *2017 IEEE International Conference on Cluster Computing*.
- [47] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *2015*

- 48th Annual IEEE/ACM International Symposium on Microarchitecture.
- [48] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 178–190.
  - [49] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the long latency of persist barriers using speculative execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 175–186.
  - [50] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 50–61.
  - [51] Zhenyu Sun, Xiuyuan Bi, Hai Helen Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. 2011. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*. ACM, 329–338.
  - [52] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 91–104.
  - [53] H. Volos, A. J. Tack, and M. M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Architectural Support for Programming Languages and Operating Systems*.
  - [54] Xiaojian Wu and AL Reddy. 2011. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 39.
  - [55] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*.
  - [56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.
  - [57] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 167–181.
  - [58] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. 2017. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *2017 IEEE International Conference on Cluster Computing*.
  - [59] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. 2019. Optimizing Persistent Memory Transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
  - [60] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-tolerant Persistent Memory Programming Library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*.
  - [61] Yiyang Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.
  - [62] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap between Systems with and without Persistent Support. In *MICRO*.
  - [63] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 461–476.
  - [64] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 128.