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

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.


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
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 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 s_fence 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 ¹ 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

```
/* Loop through cacheline aligned chunks*/
/* covering a target data object*/
cache_block_flush ( const void * addr , size_t len )
{
    unsigned __int64 ptr ;
    for ( ptr = ( unsigned __int64 ) addr & ~ ( FLUSH_ALIGN - 1 ) ;
        ptr < ( unsigned __int64 ) addr + len ;
        ptr += FLUSH_ALIGN )
    {
        /* cfflush/cfflush_opt/clwb*/
        flush ( (char *)ptr ) ;
        /* cfflush_opt and clwb needs a fence*/
        /* to ensure its completeness*/
        _mm_sfence ( ) ;
    }
}
```

### 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 concurrence 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].

²We 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.

---

20% 40% 60% 80% 100%
PMEMKV Redis Fast&Fair Level-Hashing Streamcluster Canneal Dedup

Normalized Time (%)

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 persistency [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 it 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...
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 bidirectional. At a low concurrency level, there is no sufficient writeback 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 peaks. 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.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>YCSB</th>
<th>TPC-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.43</td>
<td>0.44</td>
</tr>
<tr>
<td>B</td>
<td>0.55</td>
<td>0.56</td>
</tr>
<tr>
<td>C</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>D</td>
<td>0.49</td>
<td>0.52</td>
</tr>
</tbody>
</table>

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 overweight 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 writes 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
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 $\lceil N/M \rceil + 1$ 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 ($N_{UM_{thr}}$) 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 concurrence points to estimate $N_{UM_{thr}}$ 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 $N_{UM_{thr}}$ to be $P4$. If $BW_{1}^{pmm}$ is higher than $BW_{2}^{pmm}$, but $BW_{4}^{pmm}$ is lower than $BW_{1}^{pmm}$, it means that the peak is between $P2$ and $P3$. The control thread sets $N_{UM_{thr}}$ 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 $BW_{2}^{pmm}$, the control thread selects $N_{UM_{thr}}$ 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 $N_{UM_{thr}}$, 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 $N_{UM_{thr}}$ to be the maximum available concurrency.
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, Ribbon uses a thread to proactively issue a flush (the thread is named the proactive thread). Later on, when the application threadflushes 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 threadflushes 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. 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

```c
#define VALUE_LEN 100
#define KEY_LEN 24

// The original code without coalescing

void setGenericCommand( client *c, char *key, char *val ) {
    TX_BEGIN( server.pm_pool ) {
        char newVal = alloc_mem(VALUE_LEN);
        dupStringObjectPM(newVal, val);
        flush(newVal, VALUE_LEN);
        _mm_sfence();
        char newKey = alloc_mem(KEY_LEN);
        setKeyPM(c->db, key, newkey, newVal);
        flush(newkey, KEY_LEN);
        _mm_sfence();
    } TX_ONABORT { ... } TX_END ... }

// The code with coalescing

void setGenericCommand_coalescing( client *c, char *key, char *val ) {
    TX_BEGIN( server.pm_pool ) {
        char mem = alloc_mem(VALUE_LEN + KEY_LEN);
        char *newVal = get_mem(0);
        dupStringObjectPM(newVal, val);
        __force_aligned(newVal);
        setKeyPM(c->db, key, newKey, newVal);
        flush(mem, VALUE_LEN + KEY_LEN);
        _mm_sfence();
    } TX_ONABORT { ... } TX_END ... }
```

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

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...
### 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.

<table>
<thead>
<tr>
<th>API name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ribbon_start(int numAppT)</td>
<td>Initializes the runtime system and resource (e.g., flushing threads and FIFO queues)</td>
</tr>
<tr>
<td>int eleF(quipeur)</td>
<td>Put CLF requests into flushing queues</td>
</tr>
<tr>
<td>int ribbon_flush(void* addr, size_t len)</td>
<td>Memory allocation for coalescing CLF</td>
</tr>
<tr>
<td>void* ribbon_alloc(size_t len, int type)</td>
<td>Terminate runtime and release resources</td>
</tr>
<tr>
<td>int ribbon_stop()</td>
<td>Enures all pending CLF requests are flushed</td>
</tr>
<tr>
<td>int ribbon_free(void* addr)</td>
<td>Free a memory allocation</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Table 2: Ribbon APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>API name</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>int ribbon_start(int numAppT)</td>
</tr>
<tr>
<td>int eleF(quipeur)</td>
</tr>
<tr>
<td>int ribbon_flush(void* addr, size_t len)</td>
</tr>
<tr>
<td>void* ribbon_alloc(size_t len, int type)</td>
</tr>
<tr>
<td>int ribbon_stop()</td>
</tr>
<tr>
<td>int ribbon_free(void* addr)</td>
</tr>
</tbody>
</table>

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.

### 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.

<table>
<thead>
<tr>
<th>API name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ribbon_start(int numAppT)</td>
<td>Initializes the runtime system and resource (e.g., flushing threads and FIFO queues)</td>
</tr>
<tr>
<td>int eleF(quipeur)</td>
<td>Put CLF requests into flushing queues</td>
</tr>
<tr>
<td>int ribbon_flush(void* addr, size_t len)</td>
<td>Memory allocation for coalescing CLF</td>
</tr>
<tr>
<td>void* ribbon_alloc(size_t len, int type)</td>
<td>Terminate runtime and release resources</td>
</tr>
<tr>
<td>int ribbon_stop()</td>
<td>Enures all pending CLF requests are flushed</td>
</tr>
<tr>
<td>int ribbon_free(void* addr)</td>
<td>Free a memory allocation</td>
</tr>
</tbody>
</table>

### 6 EXPERIMENTAL RESULTS

**In this section, we evaluate the performance of Ribbon.**

**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).

<table>
<thead>
<tr>
<th>BW Variance Threshold</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>9.7%</td>
</tr>
<tr>
<td>20%</td>
<td>16.5%</td>
</tr>
<tr>
<td>30%</td>
<td>13.8%</td>
</tr>
<tr>
<td>40%</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interval (sec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.5%</td>
</tr>
<tr>
<td>3</td>
<td>13.8%</td>
</tr>
<tr>
<td>5</td>
<td>11.3%</td>
</tr>
</tbody>
</table>

### Table 6: Sensitivity study on proactive CLF

<table>
<thead>
<tr>
<th>#app threads</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.4%</td>
</tr>
<tr>
<td>2</td>
<td>6.6%</td>
</tr>
<tr>
<td>4</td>
<td>7.3%</td>
</tr>
<tr>
<td>8</td>
<td>9.3%</td>
</tr>
<tr>
<td>12</td>
<td>9.4%</td>
</tr>
<tr>
<td>16</td>
<td>9.8%</td>
</tr>
<tr>
<td>17-24</td>
<td>9.3%</td>
</tr>
</tbody>
</table>

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
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 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.
We also compare the CLF coalescing technology with the combination of the concurrency control and proactive CLF (w. cc+pclf). 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

Persistence 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 persistence ordering. Ribbon is generally applicable to various persistency models.

CLF-oriented optimizations. Lazy Persistency [2] avoids eager cache flushing and relies on natural evicton 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-cache line 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 reusable 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 persist concurrency, while orders of persists across epochs are enforced. SCMFs [54] and NOVA [55] are PM-aware file systems with failuratomic, 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. Their 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.


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.
