Ribbon: High-Performance Cache Line Flushing for Persistent Memory

Kai Wu[1], Ivy Peng[2], Jie Ren[1], Dong Li[1]
{kwu42,jren6,dli35}@ucmerced.edu
[1]UC Merced, [2]Lawrence Livermore National Laboratory

1 INTRODUCTION

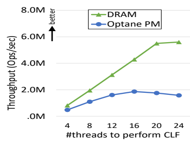
Cache line flushing (CLF) is a fundamental building block for programming persistent memory (PM). PM-aware workloads often rely on CLF to ensure crash consistency. However, CLF creates a performance bottleneck on PM, which may significantly reduce the performance benefits promised by PM. For instance, our evaluation shows that CLF can reduce system throughput by 62% for database applications like Redis when running *dbench* to perform *randomfill* operations.

Most existing works focus on optimizing persistency semantics (e.g., skipping CLF or relaxing constraints on persist barriers), other than the CLF mechanism. They use different fault models or recovery mechanisms that are designed for specific application characteristics. All these techniques use the CLF mechanism to realize their persistency semantics.

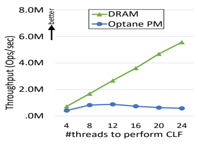
Unlike the previous works, we focus on the CLF mechanism itself, instead of persistency semantics. Therefore, our work is generally applicable to PM-aware applications. We reveal the characteristics of CLF on real PM hardware. Based on the characteristics, we introduce a runtime system, called Ribbon . Ribbon decouples CLF from the application and applies model-guided optimizations to the CLF mechanism. Applying Ribbon on a PM-aware application does not change its persistency semantics so that the program correctness is retained.

Our performance study of CLF on real PM hardware (i.e., Intel Optane DC PM) 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. On Optane PM (Figure 1), 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 due to 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 CLF exacerbates the scaling limitation. *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*, flushed cache lines may 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.

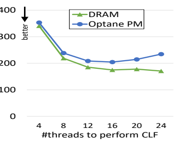
[*]The full paper is published in the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT'20) http://pasalabs.org/papers/2020/PACT20_Ribbon.pdf. LLNL-ABS-800045.



(a) PMEMKV-256.



(b) PMEMKV-1024.



(c) Streamcluster.

Figure 1: Performance at increased numbers of threads performing CLF.

Our evaluation of Redis with YCSB (loads and A-F) and TPC-C workloads shows that the average dirtiness of flushed cache lines is only 47%.

Driven by the observations, we introduce Ribbon – a runtime system that optimizes the CLF mechanism through concurrency control that adapts the intensity of CLF and uses proactive CLF to increase the probability of flushing clean cache lines. Ribbon automatically detects CLF bottleneck in oversupplied or insufficient concurrency, and adapts accordingly. Ribbon also proactively transforms dirty or non-resident cache lines into clean resident ones to reduce the latency of CLF. Furthermore, we investigate the cause for low dirtiness of flushed cache lines in seven representative PM-aware workloads and provide an application-specific solution to coalesce cache lines that achieves up to 33.3% (22% 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 performance.

2 RIBBON DESIGN

Ribbon uses decoupled concurrency control of CLF and proactive cache line status transformation at runtime. For workloads with low dirtiness of flushed cache lines, CLF coalescing is also employed.

1. Concurrency Control on Cache Line Flushing. Concurrency control provides a bi-directional adjustment that automatically (1) throttles CLF concurrency when memory contention is detected; (2) boosts CLF concurrency when PM is underutilized. Figure 2 depicts the general workflow of the concurrency control.

Ribbon creates a thin conceptual layer between the application and the underlying memory. Within the conceptual layer, Ribbon decouples the application and CLF by instrumenting CLF and memory fence instructions, i.e., `clwb`, `clflushopt`, `clflush`, and `sfence`. Instead of executing these instructions at application threads, each application thread collects the intercepted instructions and enqueues them into an FIFO queue. These intercepting CLF requests are fetched and executed by a group of helper threads, named *flushing threads*, from FIFO queues. These flushing threads use the intercepted memory fence instructions as deadlines to finish CLF. Each application thread has its private FIFO queue, while a flushing thread may handle multiple queues. Our design uses a circular buffer to reduce contention on the FIFO queue between threads.

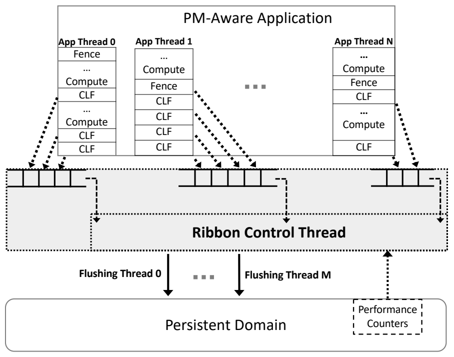


Figure 2: Concurrency control on cache line flushing.

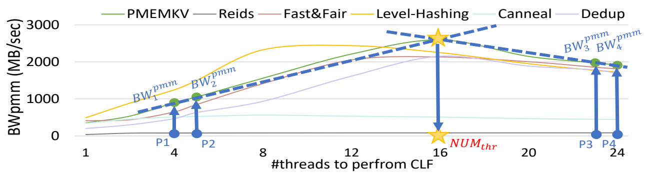


Figure 3: PM bandwidth at various numbers of flushing threads.

The head and tail indexes are the only two variables exchanged among threads. Synchronization is rare since application threads only update the head, while flushing threads only update the tail.

To determine the concurrency of flushing threads (NUM_{thr}), Ribbon launches a helper thread (named *control thread*) at the beginning of the application and checks a set of performance counters periodically. In particular, Ribbon records the write bandwidth to PM DIMMs (BW_{pmm}) at an interval T. System evaluation shows that when the concurrency level increases, the bandwidth to PM first increases to a peak and then starts decreasing [1, 2]. 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}. Figure 3 shows an example. P1-P4 are four sampling concurrency points of PMEMKV (a benchmark in Intel PMDK). The dashed line and the intersection illustrate the optimal concurrency level for PMEMKV. Please check our full paper for more details.

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. At the beginning of the application, Ribbon creates a helper thread (named *proactive thread*) to perform CLF proactively.

The proactive thread uses performance counters to detect which cache block has been updated recently. In particular, the proactive thread uses the sampling mode commonly found in hardware performance counters (e.g., Precise Event-Based Sampling from Intel processor or Instruction-based Sampling from AMD processor) to collect memory store events periodically. Leveraging the sampling mode, the proactive thread is able to collect virtual memory addresses whose associated memory references cause the memory store events. With those memory addresses, the proactive thread identifies dirty cache blocks. After the proactive thread finds a dirty cache block, the proactive thread immediately flushes it.

3. Coalescing Cache Line Flushing. Ribbon coalesces cache lines with low dirtiness to reduce the number of cache lines to flush. We reveal two reasons that account for the low dirtiness of flushed cache lines: unaligned cache-line flushing and uncoordinated cache-line flushing. The unaligned CLF happens when a persistent object is not well aligned with cache lines. The uncoordinated CLF happens when the correlation of memory allocations is ignored (i.e., allocating multiple associated data objects into separate cache blocks).

To address the above two coalescing problems, Ribbon introduces a new memory allocation mechanism to coalesce cache-line flushing and improve efficiency by well-aligning cache lines and considering the semantic correlation between memory allocations. We use Redis (i.e., Redis-libpmemobj) as an example. The traditional Redis implementation 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, Ribbon 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.

3 PERFORMANCE

We evaluate Ribbon on Intel Optane DC with seven representative PM-aware workloads (PMEMKV, Redis, Fast&Fair (B+-tree), Level-Hashing, and three Parsec workloads (Streamcluster, Canneal and Dedup)) at a low and high thread-level concurrency, i.e., running at 4 and 24 application threads, respectively. When there are four application threads, Ribbon increases the concurrency of CLF and achieves up to 17.6% improvement (9.3% on average) over the original implementation. When there are 24 application threads, Ribbon effectively detects memory contention and achieves up to 49.8% improvement (20.2% on average) over the original implementation. We also evaluate the effectiveness of CLF coalescing by running YCSB and TPC-C against Redis at 24 clients threads. The cache line dirtiness is improved from 0.32-0.56 to 0.4-0.68. The increased dirtiness results in 18%-33% speedup for all workloads.

REFERENCES

[1] I. B. Peng, M. B. Gokhale, and E. W. Green. System evaluation of the intel optane byte-addressable NVM. In *MEMSYS 2019*.
[2] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX FAST 2020*.