Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network

Xing Hu* Matheus Ogleari† Jishen Zhao‡ Shuangchen Li* Abanti Basak* Yuan Xie*
University of California, Santa Barbara*, University of California, Santa Cruz†, University of California, San Diego‡
{huxing,shuangchenli,abasak,yuanxie}@ece.ucsb.edu*, mogleari@ucsc.edu†, jzhao@ucsd.edu‡

***Abstract*—Emerging non-volatile memories (NVM), such as phase change memory (PCM) and Resistive RAM (ReRAM), incorporate the features of fast byte-addressability and data persistence, which are beneficial for data services such as file systems and databases. To support data persistence, a persistent memory system requires ordering for write requests. The data path of a persistent request consists of three segments: through the cache hierarchy to the memory controller, through the bus from the memory controller to memory devices, and through the network from a remote node to a local node. Previous work contributes significantly to improve the persistence parallelism in the first segment of the data path. However, we observe that the memory bus and the Remote Direct Memory Access (RDMA) network remain severely under-utilized because the persistence parallelism in these two segments is not fully leveraged during ordering.**

In this paper, we propose a novel architecture to further improve the persistence parallelism in the memory bus and the RDMA network. First, we utilize *inter-thread persistence parallelism* for barrier epoch management with better bank-level parallelism (BLP). Second, we enable *intra-thread persistence parallelism* for remote requests through RDMA network with buffered strict persistence. With these features, the architecture efficiently supports persistence through all three segments of the write datapath. Experimental results show that for local applications, the proposed mechanism can achieve 1.3× performance improvement, compared to the original buffered persistence work. In addition, it can achieve 1.93× performance improvement for remote applications serviced through the RDMA network.

I. Introduction

Non-volatile memory (NVM) technologies, such as phase-change memory (PCRAM), spin transfer-torque magnetic RAM (STT-RAM), and resistive memristor (ReRAM), emerges with promising potential to replace DRAM for main memory [27], [56]. Compared with DRAM technology, NVMs incorporate the features of fast byte-addressability and disk-like data persistence in addition to a superior storage density. Such emerging storage class memories (SCMs) can bridge the gap between memory and storage, enabling recoverable data structures in main memory, which is beneficial for key data services in data centers, such as file systems and databases [6], [10], [11], [14], [18], [22], [29], [37], [40], [45], [47], [54], [59]. Consequently, industrial efforts are noticeable in the area of NVMs. For example, Intel announced 3D XPoint production for both storage and memory in 2016 [41].

Ensuring recoverability of persistent data in data services, regardless of power failure or program crash, is not easy because of the volatile caches and memory operation reordering during execution. Previous work adopts both software versioning mechanisms and hardware persistent ordering capability to achieve this goal [16], [33], [50]. The correctness of versioning mechanisms relies on the hardware's capability to maintain the persistence ordering exactly in the way that software defines it. Specifically, hardware must ensure that the requests before a barrier are persisted before the requests after the barrier [14], [16], [33], [50]. These persistent requests go through the memory hierarchy with ordering control, following a datapath (as shown in Figure 1) that can be divided into three segments: (1) *through the cache hierarchy to the memory controller*; (2) *through the bus from the memory controller to memory devices*; (3) *through the Remote Direct Memory Access (RDMA) network from a remote node to a local node*.

For the first segment of the datapath (through the cache hierarchy to the memory controller), buffered persistence methods have been proposed to alleviate the persistent ordering overhead in the cache hierarchy, decoupling memory persistence from core execution for better performance [23], [25], [26], [35]. However, previous work pays little attention to the second and the third segments of the persistence datapath. We observe that the latter two datapath segments, which include the memory bus and RDMA network, are severely underutilized during data persistence, because of the failure to leverage the persistence parallelism. There are two types of persistence parallelism, **intra-thread** and **inter-thread**, in the persistent datapaths. When two threads are independent from each other, the persistent requests between them can be freely scheduled without restrictions, which leads to inter-thread persistence parallelism. If the hardware provides the ability to keep tracking of the ordering dependency and manage the orders to be persisted in the persistent domain finally, there will be overlaps among these requests when they are processed in the datapaths, which leads to intra-thread parallelism. We will explain the impact of these two kinds of parallelism.

Inefficiencies in the Memory Bus. The buffered epoch memory model provides the inter-thread persistence parallelism in the cache hierarchy which enables multiple request epochs (a request sequence divided by barriers) flowing through the cache hierarchy. However, prior work alleviates persistent barrier constraints without considering the requests memory location [23], [25], [42]. When the requests in the same epoch have bank conflicts, they still need to be serviced

one by one even without persistent restrictions. Experiments show that the memory request epochs sent to the memory controller may have low bank-level parallelism (BLP), resulting in bad memory scheduling efficiency [28], [38], [58].

Inefficiencies in the RDMA Network. In observing the opportunity of the remote NVM devices which has much faster speed than the local SSD storage, both academic and industry researchers pay much attention to the remote NVM systems [6], [18], [21], [34], [44], [48]. These systems rely on memory persistence through the RDMA network, which is referred to as **network persistence** throughout this paper. In this synchronous solution, transactions have to be divided to many epochs to be persisted in the remote NVM sequentially [48]. Therefore multiple sequential RDMA round trips are needed for data persistence. According to the statistics, more than 90% of network persistence time is spent on RDMA round trips (further described in Section III). Hence, the intra-thread persistence parallelism is very important for the network request performance, considering the dominated network overhead.

Figure 2 summarizes the contribution of prior work and the superiority of this work. Prior work realizes improvement by leveraging the persistence parallelism in the datapath of cache hierarchy to memory controller [23], [25], [26], [33], [35], [39], [42], leaving the inefficiencies in the other two datapath segments unsolved. To maximize the persistence parallelism for better memory bus and network utilization, we make the following observations for these datapath segments: (1) for the datapath segments through the network and through the cache hierarchy to the memory controller, it is more efficient to record rather than implement the persistent ordering constraints. Hence, we can leverage *intra-thread persistence parallelism* by coalescing more persistent requests in these datapaths; (2) for the datapath from the memory controller to the memory devices, it is required to implement ordering constraints to make sure that the data persisted in NVM device is in order. In this datapath, we leverage *BLP-aware inter-thread parallelism* to ensure persistent ordering with better memory throughput. In summary, our contributions are listed as follows:

- We identify that existing persistence request management schemes significantly under-utilize the bandwidth available at the memory bus and the RDMA network.
- We propose and implement the persistence parallelism management methodology based on the buffered strict persistence model. It enables BLP-aware barrier epoch management, which leverages large amount of inter-thread persistence parallelism to improve BLP of persistent requests across local memory bus.
- We propose the architecture design to support buffered epoch strict model for remote persistent requests through network. Our remote persistent request management scheme exploits the intra-thread persistence parallelism for better performance, while enforcing the ordering of remote persistent requests.

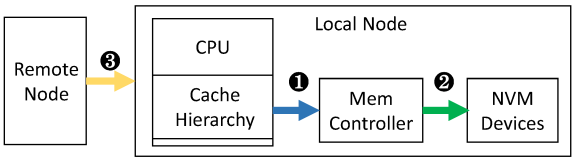

Figure 1. Datapath that persistent requests go through. Two possible datapaths are 1-2 (local) and 3-1-2 (from remote to local).

	❶ Ordering Ctrl Cache hierarchy -> MC		❷ Ordering Ctrl MC->NVM	❸ Ordering Ctrl Remote node-> Local node	
	Intra-thread parallelism	Inter-thread parallelism	Inter-thread parallelism for BLP	Intra-thread parallelism	Inter-thread parallelism
Sync Memory Model [43]	X	X	X		
Buffered Epoch Memory Model [25][42]	√	√	X		
RDMA Solution [17]				X	√
This work	√	√	√	√	√

Figure 2. Persistence parallelism across various ordering approaches.

II. Background

A. Versioning and Ordering Control

To support the persistence property for storage systems, most persistent memory designs borrow the ACID (atomicity, consistency, isolation, and durability) concepts from database and file systems [5], [8], [33], [50], [52], [59]. The durability can be guaranteed by the non-volatile nature of NVM, the atomicity is supported by storing multiple versions of the same piece of data and carefully controlling the order of writes into the persistent memory [59]. Commonly-used methods to maintain multiple versions and ordering include redo logging, undo logging, and shadow updates [14], [33], [50], [52]. Ordering control means that the hardware performs the memory access orders exactly as specified by the software program. For example, to make sure that the log is persisted in the memory before the data is persisted, the program uses memory barrier instructions between the writes to the log and the writes to the persistent data [13], [14], [48], [50]. The hardware ensures that the requests after a barrier will not be persisted ahead of the requests before this barrier.

B. Synchronous ordering and Delegated ordering

Previous work defines synchronous ordering from Intel ISA-based solution [25], [42], [43]. The synchronous ordering prevents instructions ordered after a fence instruction from retiring until prior persist writes complete. It enforces order by stalling core execution, which places persistent memory write latency on the critical path and incurs significant performance overhead. Delegated ordering is proposed to address this issue, where the volatile execution may proceed ahead of properly ordered persistent writes [25]. The delegated ordering is based on buffered strict persistence, in which the persist writes reflect the order that stores become globally visible, but some persistence may be delayed. When failures occur, some writes may be lost, but the data structure consistency is maintained for recovery correctness [25]. Our design is implemented

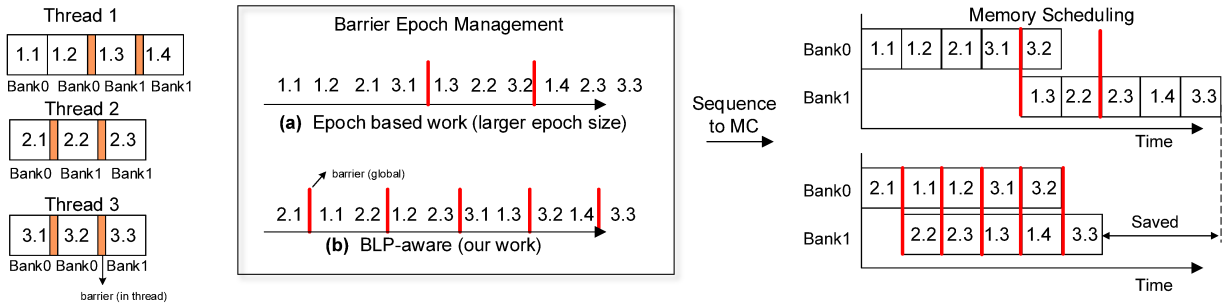

Figure 3. The barrier epoch management strategies. Two observations: 1) Barrier epoch management has large design space without violating ordering constraints. 2) Optimization for large barrier epoch may not lead to better memory scheduling efficiency.

based on delegated ordering with buffered strict persistence model [42].

C. Memory Persistence through Network

The software system community envisions the opportunity of remote persistent memory systems to provide fast write replication for better system availability in storage workloads, such as visualization, enterprise, and data center applications [44], [48]. There are several reasons for it: 1) The write latency of replication is critical for such storage workloads, because all such copies must be made durable before responding [48]. 2) The NVM shows its advantage of write latency which is almost two-scale faster than local SSD. Therefore, industry researchers try to enhance the persistence capability for remote NVM system [17], [47]. The academic researchers explore the high-performance databases, or file-systems [6], [21], [30], [57] , and shared persistent memory systems [34], [44] based on RDMA network-attached NVMs.

III. Motivation

In this section, we illustrate the inefficiencies in the memory bus and memory network during data persistence.

Inefficiencies in the Memory Bus. Prior studies [23], [25], [33], [35] leverage the persistence parallelism between independent threads. These studies employ a barrier epoch management strategy that utilizes larger epoch sizes to alleviate the persistent ordering restrictions. Although they alleviate the barrier restrictions, we observe that optimization for larger epoch size may not lead to a better bank-level parallelism or larger memory scheduling space.

Considering the case in Figure 3 as an example, there are three transactions from three threads, and every thread is independent from one another. The orange slices represent the barrier instructions in every thread. The bank location information is labeled below every request. The previous work discovers the persistence parallelism among the threads and schedules requests to achieve more memory scheduling freedom with a larger epoch size [25] [23] [33]. In this case, after the barrier region management, the requests sent to the memory controller are as follows: *(1.1, 1.2, 2.1 , 3.1), barrier, (1.3, 2.2, 3.2), barrier, (1.4, 2.3, 3.3)*. Indeed, the epoch size (number of requests between barriers) is larger. However, requests *1.1, 1.2, 2.1, 3.1* are all in bank 0, which will be scheduled sequentially because of the bank conflicts. If the requests from different threads can be scheduled in a BLP-aware manner, as shown in Figure 3(b), the sequence sent to the memory controller has more BLP, which facilitates memory controller scheduling efficiency, meanwhile ensuring the persistent ordering constraints.

We make two observations in this case: 1) There is a large design space for barrier epoch management to exploit the inter-thread persistence parallelism without violating the ordering constraints. 2) Relaxing barrier restriction with larger epoch size does not necessarily lead to better memory scheduling efficiency. Our motivational studies show that 36% of the requests are stalled by bank conflicts. Hence, *the main idea of our work is to optimize the memory scheduling efficiency rather than to reduce barrier restrictions.*

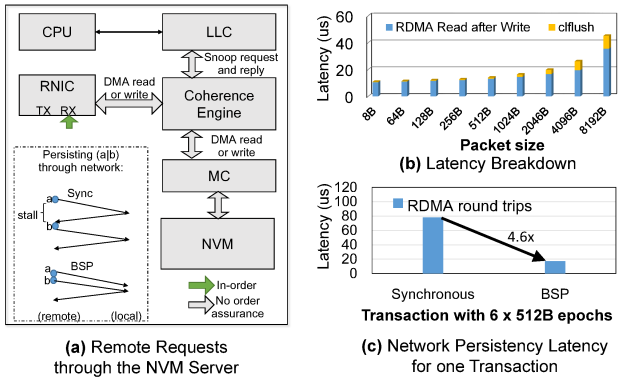

Figure 4. Synchronous vs. BSP network persistence.

Inefficiencies in the RDMA Network. The remote persistent memory can be used as the replacement of SSD for meta data storage in some high performance systems. Figure 4(a) shows how remote requests are being processed in the NVM server node. Although RDMA requests can be transported through network in order, the cache hierarchy and memory controller do not enforce the persist order of the remote requests to NVM devices. Therefore, to implement the ordering control for the remote network requests, we need additional support of synchronous persistent verification operation for every epoch. For example, to persist data block *a* and *b* in target node in order, the RDMA write operations for *b* will not be issued until after verifying that reqeust *a* has been persisted [48]. This method puts the network latency in the critical path of system which deteriorates the system performance significantly. As

shown in Figure 4(b), the RDMA round trips occupy most of the latency overhead for network persistency. If we can assure the persist ordering for the remote requests with buffered persistency model, the system can send the RDMA requests asynchronously and only need to check whether the final request in the transaction is persisted. As shown in Figure 4(c), the asynchronous method with the assistance of buffered strict persistency model (abbreviated as BSP) can reduce the RDMA round trip time by 4.6x, when persisting a transaction with 6 epochs of size 512B.

IV. Architectural Design

In this section, we first introduce the detailed memory persistence model, which is the foundation of the architecture design. Then the details of the architectural design and the system support will be introduced.

A. Memory Persistence Model

Buffered Strict Persistence. Our work adopts the delegated ordering method implemented in buffered strict persistency model, originally proposed by Kolli *et al.* [25]. In buffered strict persistency model, the persist memory order is the same as volatile memory order. Taking the Figure 5 for example, the request sequence of Thread P consists of (*b, barrier, d*) and Thread V consists of (*a, barrier, c*). Because of $Barrier_{p1}$ and $Barrier_{v1}$, request a must be persisted before c (formalized as $PMO_c < PMO_a$) and request b must be persisted before d (formalized as $PMO_d < PMO_b$). There is a write conflict between a and d, and we have $VMO_a < VMO_d$, so request b must be persisted before c (formalized as $PMO_c < PMO_b$). In a summary, the ordering constraints in a system can be classified into two categories: intra-thread and inter-thread. Specifically, *the intra-thread ordering arises from barriers, which divides the instructions within a thread into epochs. Inter-thread persistent ordering arises from coherence order and fence cumulativity* [4], [23], [42]. This rule is also applicable to requests over RDMA, since the RDMA operations are cache-coherent with local accesses [12], [55], as shown in Figure 4(a).

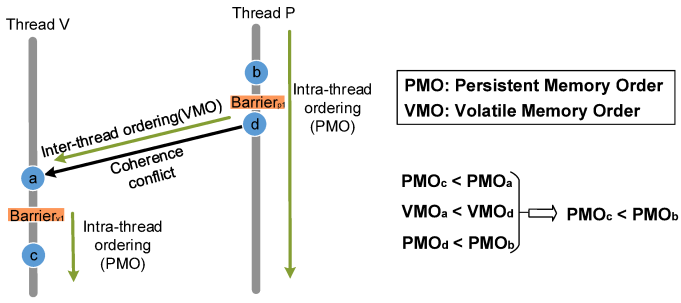

Figure 5. Persistent ordering in systems.

Persistence Parallelism. There are two types of persistence parallelism, ***intra-thread*** and ***inter-thread***, in the persist datapaths. When two threads are independent from each other, the persistent requests between them can be freely scheduled without restrictions, which leads to inter-thread persistence parallelism. If the hardware provides the ability to keep tracking of the dependency, these requests can be processed in the persist datapaths in an overlapped manner, which leads to intra-thread parallelism.

We should distinguish that the inter-thread and intra-thread persistence parallelism play different roles among the persist datapaths. In the memory bus, the BLP-aware inter-thread parallelism is important for scheduling efficiency. In RDMA network, the intra-thread parallelism plays a more important role for system performance because the sequential persist ordering restrictions put the network overhead in the critical paths and incur large overhead. Therefore, we propose barrier region management method that considers BLP for local requests and enable the epoch strict memory model for remote requests.

B. Architecture Overview

The overview of the architecture is shown in Figure 6. The key components consist of the ***persist buffer*** and the ***barrier region of interest (BROI) controller***, which keep track of dependency and make the ordering control. In summary, the persist buffer and BROI controller cooperate to implement the following functions: *barrier epoch identification*, *barrier epoch buffering*, and *BLP-aware barrier epoch management*.

Persist Buffer. We follow the similar persist buffer implementation in prior work to observe, record, and enforce persistent dependencies [25]. There is one persist buffer for every core and an additional persist buffer to process the remote requests. There are several fields in every entry of the persist buffer: operation type (requests or fences), cache block address, data to be persisted, ID that uniquely identifies each in-flight persist request to a particular address, and the array of inter-thread dependencies for this entry. In a summary, the persist buffers identify the barrier epoch region and resolve the inter-thread dependency.

BROI Controller. The requests will be sent to BROI controller only after the persist buffer ensures that there is no inter-thread dependency. As introduced in the motivational section, the barriers are originally visible inside the thread and its dependent threads if there it is. After sent to the memory controller, all the requests are flattened and the barriers become visible to the all the threads. This procedure is referred to as the barrier region management. The BROI controller makes BLP-aware barrier region management across different epoches of independent threads without conflicts, and meanwhile maintains the intra-thread ordering restrictions.

The BROI controller uses BROI queues to buffer barrier epochs for the ease of leveraging intra-thread and inter-thread persistence parallelism. There are local and remote BROI queues in the BROI controller which store the persistent requests from local and remote processors. Each BROI queue has several BROI entries which store the barrier epochs of every thread. Each BROI entry is equipped with barrier index registers to indicate the barrier locations of the request sequence. The requests in different BROI entries are independent from each other. There are several fields in each entry to

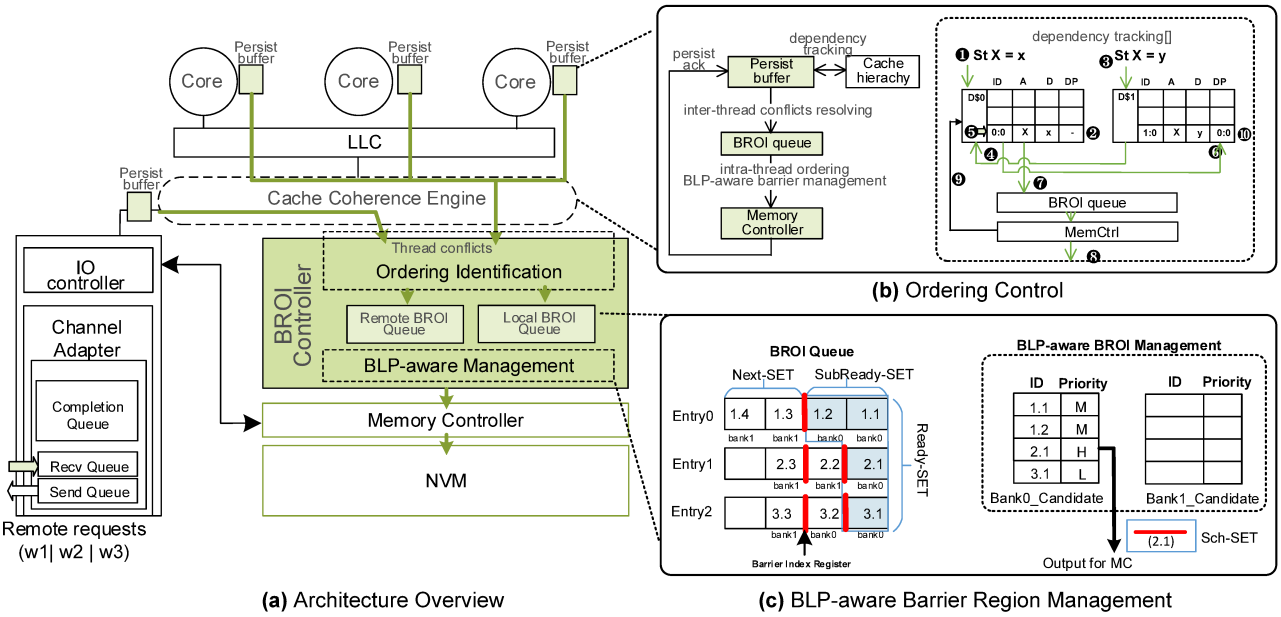

Figure 6. Architecture overview: the persist buffer and BROI (barrier region of interest) controller conduct ordering control, support buffered strict memory persistence through network, and make BLP-Aware BROI management to facilitate the memory scheduling.

store the persistent request information including both request addresses and data. The BROI controller maintains the request sequence out of the BROI queues to the memory controller, leveraging inter-thread persistence parallelism to provide more BLP for memory controller scheduling.

C. Epoch Ordering Identification

In this section, we will introduce the detailed implementation of ordering identification.

Programming Interface. It is normally implemented with fence instructions to define the ordering constraints for local requests [23], [25], [26], [33], [35], [39], [42], as shown in Figure 7(a). In RDMA software stack, a new programming interface for persistent write requests is needed, so that the system can distinguish them from normal write requests. We can use new `rdma_pwrite` semantic, which is similar like `rdma_write` with the only difference that the hardware will treat the data block in `rdma_pwrite` as in one barrier region. There is also an alternative solution that marks the persist feature in the available tag bits of the RDMA write operations, as long as the system can identify them with normal write requests. The overall system support is introduced in the Session. V.

```
/*request a, b
are persistent*/
St a = x
fence
St b = y

Constraint: St a <p St b
```

a. Local requests

```
//Epoch Persistency
/*data block a, b are
persistent*/
rdma_pwrite(ptr_a)
rdma_pwrite(ptr_b)

Constraint: St a <p St b
```

b. Remote requests

Figure 7. Ordering indication in software.

Ordering Control.

1) Dependency Tracking. The persist buffer and cache hierarchy are in the cache coherence region. The cache coherence engine tracks the inter-thread dependency between the persist buffer and cache hierarchy and persist buffer is updated accordingly during execution. A request will be sent to BROI queue only when the persist buffer ensures that this request has no inter-thread dependent requests to be resolved. In this way, the requests sent to BROI controller have no inter-thread conflicts. After the requests drained from the data buffer in memory controller, the memory controller sends back the acknowledgements and the corresponding requests will be deleted from the entries in persist buffer. Then the persist buffer updates the dependent field of every entry accordingly.

i) Inter-thread dependency. This scenario includes both dependency in the local requests and the remote requests. For local requests, previous work proves that either direct dependency (persist-persist dependency) or chain dependency (epoch-persist dependency) could be tracked in persist buffer with the assistance of cache coherence engine [25]. The persist buffer is also able to track the dependency of the remote requests and resolve the dependency, since the RDMA operations are cache-coherent with local accesses [12], [55]. For data services in real applications, most threads are independent from each other and only 0.6% of the requests have conflicts [39], which proves that there is a large amount of inter-thread persistence parallelism. There may be fault positive inter-thread conflicts. We do not take care of that part to keep the hardware overhead small.

ii) Intra-thread dependency. Local requests in persist buffer will be in a first-in-first-out manner. When there is no inter-thread dependency, the BROI controller receives the requests from persist buffer. Then BROI controller allocates the requests in local BROI queue or remote BROI queue. In BROI

queue, there is one BROI entry for every thread, and each entry only maintains the ordering of requests that it contains. The BROI controller implements the intra-thread dependency by blocking a request in a BROI entry until its previous fences are resolved. Since it is guaranteed that the requests from different threads are independent from each other, the requests in different entries of the BROI queue can be flexibly interleaved to the memory controller.

The remote requests could be transferred through network controller in order. Then, the requests will be allocated in remote persist buffer. The remote persist buffer communicates with NIC to get the length of data block in this operation, then it identifies the address range of the requests in order to mark the barrier region of requests and record the fence instruction in persist entry. After resolving the inter-thread dependency, the requests will be sent to BROI controller which maintains the intra-thread ordering control.

2) A Detailed Example. In this subsection, we walk through the detailed example shown in the Figure 6(b) to illustrate how the architecture design deals with dependency.

❶ The L1 data cache of core 0, D$0, receives a persistent request $StX = x$ from core 0. ❷ After checking the coherence engine, it has no dependency and one entry of persist buffer is allocated for this request. ❸ Another request, $StX = y$, is issued from core 1 to persist data to the same address. ❹ D$1 sends a read-exclusive request to D$0. ❺ D$0 snoops both cache and persist buffer and replies there is a request with ID "0:0" in dependency region. ❻ D$1 allocates an entry for request $Stx = y$ and marks the dependency request ID in the DP field. ❼ request "0:0" is sent to BROI queue and BROI queue schedules this request to memory controller. ❽ The memory controller sends this request to NVM devices and returns an acknowledgement to persist buffer (❾). ❿ The completed entry is deleted from persist buffer and the corresponding DP field of request "1:0" is updated as no dependency. Then the request "1:0" will be scheduled to BROI queue. This case shows how to resolve a direct dependency (persist-persist dependency). This design also can resolve the epoch-persist dependency [25]. To note, the buffered strict persistence model does not change the cache coherence protocol. The persist buffer is in the cache coherent region and records the dependency information provided by the cache coherence engine for inter-thread conflict avoidance.

D. BLP-aware Barrier Epoch Management

The guidelines for barrier epoch management are as follows: 1) Persistent ordering of requests in one BROI entry must be obeyed, which can be achieved by forcing the requests after a barrier to stay in the BROI queues until all the requests before the barrier have been executed; 2) for the requests in different BROI entries with inter-thread persistence parallelism, the barrier epoch management should provide more BLP in the memory request sequence that being sent to the memory controller. The detailed BLP-aware barrier epoch management method is introduced as follows.

Terminology and Problem Definition.

First, we define the terminology for the simplicity of reference. There are t BROI entries. The barriers (number n) in BROI entry i divide requests into several sets: s_i^0, s_i^1,..., s_i^n. The scheduling for this entry must follow this order: $s_i^n < ... < s_i^1 < s_i^0$, which means no request in a later set can be scheduled until the previous set has been processed completely. There are several special sets we want to emphasize, listed in Table I.

Table I
THE REQUEST SET DESCRIPTIONS.

SubReady-SET: R_i	The first request set in BROI entry i, $R_i = s_i^0, 0 \le i \le t$
Ready-SET: R	The first request sets in all BROI entries, $R = s_0^0 \cup ... \cup s_i^0 ... \cup s_t^0$
Next-SET: N_i	The second request set in BROI entry i, $N_i = s_i^1, 0 \le i \le t$
Sch-SET: Sh	The requests to be scheduled, $Sh \in R$

Set BLP: We use the $bank_i(SET)$ function to represent whether there are requests of this SET in the bank i. Then we quantify the $BLP(SET)$ of a set as follows,

$$BLP(SET) = \sum_{i=0}^{b} bank_i(SET)), \tag{1}$$

where the bank number is b.

Problem Formulation: Having the R_i, N_i, and the bank location of every request in these sets as inputs, we want to find the *Sch-SET* with maximum *BLP(Sch-SET)*.

Scheduling Algorithm.

To maximize the utilization of memory bus, we intend to schedule the requests with maximum bank-level parallelism from every *subReady-SET*. Since the *subReady-SET* will be updated if all the requests in it are scheduled completely, we also expect the *Next-SET*, which will become the new *subReady-SET*, has more bank-level parallelism. Hence, the key principle of the BLP-aware barrier region scheduling technique is to give the higher priority to the requests in *subReady-SET* of the ith BROI entry, if it delivers larger BLP and its *Next-SET* will also bring more bank-level parallelism at the soonest. The algorithm consists of four main steps:

i) Priority calculation. Calculate the priority of requests in *Ready-SET*. The requests belonging to the same *subReady-SET* would have the same priority. The priority calculation method is shown by the following equation.

$$Priority(R_i) = BLP(R - R_i^0 + R_i^1) - \sigma \cdot size(R_i^0), \tag{2}$$

where σ a weighted parameter which indicates that the BLP is more important than the size during priority calculation.

Taking the Figure 6(c) as an example, the initial *Ready-SET* is *(1.1, 1.2, 2.1, 3.1)*. We give the highest priority to request *2.1*, because 1) completing *2.1* request can bring the request *2.2* to Ready-SET which will introduce additional bank parallelism by adding new request from $Bank_1$, and 2) the *2.1* can bring additional bank parallelism sooner than set *(1.1, 1.2)*.

ii) Enqueue requests in bank-candidate queues. Put the updated requests of *Ready-SET* to the Bank-Candidate Queue according to the request location. In the example shown in

Figure 6, the requests (*1.1, 1.2, 2.1, 3.1*) will be enqueued into Bank0-candidate queue.

iii) Output Sch-SET. Output the request with the highest priority in every bank-candidate queue to form the *Sch-SET*. In this example, the Sch-SET of current iteration is *(2.1)* under our scheduling algorithm.

iv) Update Ready-SET. When one *SubReady-SET* completes, BROI controller schedules a barrier to memory controller, and then updates the *Next-SET* next to the completed *SubReady-SET* as the new *SubReady-SET*. *Ready-SET* is updated meanwhile, formalized as follows.

$$R_i = N_i;\ N_i = s_i^2, R = R_0 \cup ... \cup R_i \cup ... \cup R_t \tag{3}$$

In the example, the request *(2.2)* will become the new *SubReady-SET* of BROI Entry 1. The *Ready-SET* will be updated as *(1.1, 1.2, 2.2, 3.1)* accordingly.

Discussion.

1) Interference between Remote and Local requests. The strategy to schedule between local and remote requests is based on the following two observations: 1) The response time of remote requests are about 10x us, which is inherently much larger than the local memory access. Hence we can give the higher priority to local requests during scheduling because they are more latency-sensitive. 2) The remote memory accesses will devour the memory bandwidth if the scheduling strategy is throughput-oriented. Because the remote persistent requests are accessing contiguous memory addresses, which easily leverage row buffer locality for better memory throughput, so the memory controller is likely to schedule the remote requests with higher priority. However, it is unwise to give higher priority to the latency-nonsensitive requests. Based on these observations, the BROI controller will schedule the local requests with higher priority and keep the remote requests waiting until the memory controller queue is in low utilization. To avoid starvation, the remote requests will be flushed to memory devices if the blocked time of the remote requests exceeds the threshold.

2) Address mapping strategy. The address mapping strategies exert a great impact on the intrinsic bank-level parallelism of requests. We use the similar address mapping strategy in [58] which strides the consecutive groups of row-buffer-sized persistent write operations to different memory banks, while contiguous persistent writes of less than or equal to the row-buffer size are still mapped to contiguous data buffer space to achieve high row buffer locality. This method optimizes both the BLP and row buffer locality for memory scheduling. In the following evaluation, all the experiments are conducted using such a stride address mapping strategy.

E. Hardware Implementation and Overhead

The storage overhead for each persist buffer entry is 72B and each persist buffer requires 8 persist buffer entries. We also need additional 320B storage for dependency tracking and 8B for address range recording.

The BROI controller consists of BROI queues and the scheduling logic. For local BROI queues, there are eight BROI entries, which is equal to the number of cores. Every BROI entry has 8 units to store the requests information including the index in the persist buffer (4 bits each). Every BROI entry is equipped with two Barrier Index Registers to indicate the barrier location. Hence, the Barrier Index Register has three bits to indicate the barrier location in the BROI entry. Such implementation is sufficient for two reasons: 1) the barrier epoch scheduling only concerns the first two sets, the *SubReady-SET* and the *Next-SET* in BROI entries; 2) most epochs are very small in applications. As shown in the statistics of previous work, most epochs are singular epoch with only one request [39]. The scheduling logic first calculates the priority of request according Equation 2 and then schedules the requests with highest priority in *Ready-SET*. There is no iterative computing logic in scheduling implementation, which is fairly simple.

For remote BROI queues, there are two BROI entries, equal to the number of RDMA channels. There is a length counter for Remote BROI queue to identify the epoch boundary. Since we use address range to identify the BROI, and the remote requests are sequentially accesses to a block of memory region, we only reserve eight units in the remote entry with one Barrier Index Register.

The BROI controller was implemented in Verilog and built with the commercial logic synthesis tool Synopsys Design Compiler (DC) to evaluate the area and power overhead in a 65nm process. The experiment results show that the latency is about 0.4ns and we count the extra scheduling cycle in the McSimA+ for performance evaluation. It is not on the critical path and would not affect the working frequency. The area overhead is 247 um^2 and the power overhead is 0.609 mW, which is negligible for data center server design. The overall hardware overhead is summarized in Table. II.

Table II
HARDWARE OVERHEAD.

Dependency Tracking	320B
Persist Buffer Entry	72B
Local BROI queues	32B per core 2 Index Register: 2x3bit
Remote BROI queues	4B overall 2 Index Register: 2x3bit
Control Logic	$247um^2, 0.609mW$

V. SYSTEM DESIGN

A. Support for Network Persistence

Our system can be built based on the similar programming model that proposed by the Microsoft [49]. The applications use native file API or load/store instructions to call the service of NVM file system or do the MMU mapping. The NVM file system or libraries call the specialized RDMA write semantic for persistent requests. The difference is that the logging engine of file system or NVM libraries can send the asynchronous RDMA pwrite verb instead of the sequentially blocked way which issues the second RDMA pwrite only after the first one is completely durable.

RDMA Stack Support. To apply the buffered strict persistency in remote persistent memory, we need the support from

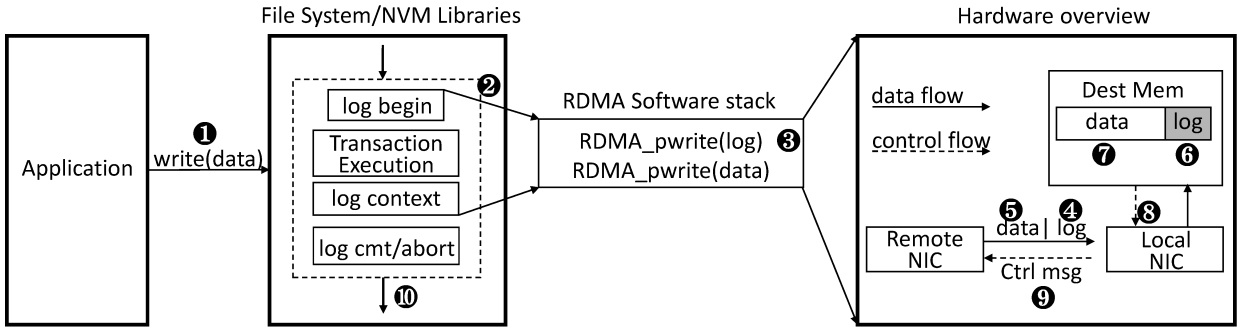

Figure 8. An example for a transaction through network with BSP network persistence.

RDMA stack [49]: 1) RDMA semantic: additional persistent write semantic is needed to distinguish between the normal write operations and the persistent operations. The basic software functionality of `RDMA_pwrite` and `RDMA_write` are similar, while the hardware will only conduct the ordering control on `RDMA_pwrite` operations. We can also implement it as a tag bit in the available payload of RDMA write semantic. 2) Advanced RDMA NIC (network interface card): it is practical to implement the persistent acknowledgement in network controller rather than using RDMA read after RDMA write. After the memory controller drains the persistent requests out of write buffer, it sends an persist acknowledgement to the network controller. Then, the NIC sends back the persist validation signal to the remote node.

Usage Example. We show an example to explain how can the system utilize the buffered epoch network persistence, as shown in Figure 8. First, applications call the file system API (such as `write`, `fsync`, `mmap`, or `msync`) to write an element (❶). Then the file system or NVM library tries to persist this element with a transaction (*log — data*) (❷). The file system or the NVM-library call the `RDMA_pwrite` verbs for log and data sequentially in a non-blocked way (❸). From the hardware perspective, the local network controller will send data to the target side in order (❹ and ❺). The persist buffer and BROI controller identify the barrier region of remote memory requests and allocates the requests in remote BROI queues which ensures the data to be persisted in order (❻ and ❼). After the data is persisted in NVM devices, the memory controller sends a signal to the local NIC (❽) and the local NIC sends the persist ACK signal to the remote NIC (❾). The file-system or NVM system library verify whether the operation is successfully persisted (❿). If the persist acknowledgement is received, then the file system will commit this transaction. Otherwise, the system will perform log abort and try to persist the transaction again (❷).

It is the technique trend to adopt the remote persistent memory system [21], [44], [48], which enables fast and instant data replica for better system availability. Buffered strict network persistence is an improvement for fundamental capability in distributed persistent shared memory system or for the systems that adopt remote NVM as the replacement of disk. In addition to reducing the response time for better system performance, it can also enable the advanced software design, such like the RDMA-friendly B+ tree [47] and other persistent objects [24], [53]. In this work, we conduct performance evaluation under the scenario where the remote NVM is adopted as the replacement of disk for replica storage. For example, when there is an update or insertion to an element in the hashtable, the log and data will be stored in the remote NVM memory for backup replication. Such a design has been widely used in many system [6], [57] to significantly improve performance versus traditional SSD-based systems. In a summary, we will compare the application performance under synchronous and buffered strict network persistence in Section VII.

B. Discussion

Persistent Domain. The persistent domain is defined as the physical domain where the data is persisted successfully only after being stored there.

For local data, the persistent domain starts at the NVM device in the experimental evaluation. Although there are Asynchronous DRAM Self-Refresh (ADR) techniques which may be used to protect the memory controller, it needs to attach the capacitor to the memory controller and also the write logic of NVDIMM to help flush the requests. Adopting the ADR is turning the memory controller to the persistent domain. The BROI scheduling is still able to do the BLP-aware barrier region management for the write pending queue in memory controller.

For cases that data waiting to be persisted in NVM server, the design scope includes client-DRAM with server-NVM and client-NVM with server-NVM [1]. For the former design, the client side has no NVM equipped and persistent data services are supported by NVM servers. This is a cost-driven design which reduces the deployment cost and makes the NVM server more efficient. The latter is a system availability-driven design where both the client and server are equipped with NVM storage, but this design may have high cost and worse performance due to the limited DIMM slots and longer access latency of NVM. Our work is applicable to both scenarios. In this work, we use the first design as the case study for performance evaluation.

DDIO-on or DDIO-off. The Direct Data I/O (DDIO) technique supports direct communication between the network and the last level cache [2], [20], which may impact the RDMA persistent memory solution. Previous work observes that DDIO-on will greatly improve the performance of data

services [20], [31]. Therefore, we use the DDIO-on solution for performance evaluation in our work. The RDMA read after write cannot assure that the data has been persisted when Direct Data I/O (DDIO) technique is enabled in target node [17]. Because the DDIO technique supports direct communication between the network and the last level cache [2], [20]. When the DDIO is on, the RDMA requests go to the cache and the RDMA read operation cannot make sure the data is from cache hierarchy (still volatile) or memory devices (persisting completed). In this work, the system adopts the advanced network controller to send the persist ACK back to the remote NIC, instead of using RDMA read after write approach. The advanced NIC approach is used in both baseline and our work.

VI. Experiment Setup

This section introduces the experimental setup and benchmark configurations for performance evaluation.

A. Simulation Framework

We evaluate the system performance from both the client-side and NVM server-side perspective. The NVM server processes requests from both the local and remote nodes. The memory throughput and local application operational throughput of the NVM server are evaluated in our work. The client sends persistent requests to NVM server and waits the data to be persisted. We evaluate the application throughput of the client with different network persistence strategy.

To evaluate the NVM server performance, we first conduct experiments with McSimA+ [3] to evaluate the local memory system performance with the input of both local and remote requests. McSimA+ is a Pin-based [36] cycle-level multi-core simulator. The configurations of the processor and memory system used in our experiments are listed in Table III. Each processor core is similar to an Intel Core i7 core. The processor incorporates SRAM-based volatile private and shared caches and employs a two-level hierarchical directory-based MESI protocol to maintain cache coherence. The cores and LLC cache banks communicate with each other through a crossbar interconnect. The byte-addressable NVM (BA-NVM) is modeled as off-chip DIMMs compatible with DDR3. The timing parameters of BA-NVM is generated by NVSim [15], as shown in Table III.

To evaluate the application performance of client system, we emulate persistence latency by inserting delays into the source code of applications or NVM libraries, similar as the prior work [18], [19], [51], [57]. The persistence latency consists of RDMA round trips and persisting procedure in the NVM server. The RDMA round trip latency is derived from network model which correlates the average time with epoch size based on thousand times of running. The persisting procedure latency in NVM server side is derived from McSimA+ simulation. We take the Whisper benchmarks [39] as the client-side benchmarks and emulate the replication scenario by inserting remote persistence latency in Whisper logging engine. The client-side nodes run with Xeon E5-2680 (Sandy Bridge) processors at the frequency of 2.5GHz.

Table III
Processor and memory configurations.

Processor	Similar to Intel Core i7 / 22 nm
Cores	4 cores, 2.5GHz, 2 threads/core
IL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
DL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
L2 Cache	8MB, 16-way set-associative, 64B cache lines, 4.4ns latency
Memory Controller	64-/64-entry read/write queues
NVRAM DIMM	8GB, 8 banks, 2KB row 36ns row-buffer hit, 100/300ns read/write row-buffer conflict [18], [27].

Table IV
A list of evaluated benchmarks.

u-bench	**Footprint**	**Description**
Hash [13]	256 MB	Searches for a value in an open-chain hash table. Insert if absent, remove if found.
RBTree [59]	256 MB	Searches for a value in a red-black tree. Insert if absent, remove if found
SPS [59]	1 GB	Random swaps between entries in a 1 GB vector of values.
BTree [9]	256 MB	Searches for a value in a B+ tree. Insert if absent, remove if found
SSCA2 [7]	16 MB	A transactional implementation of SSCA 2.2, performing several analyses of large, scale-free graph.
Whisper [39]	**Workload Configuration**	
tpcc	4 clients, 400K transactions, 20%– 40% writes	
ycsb	4 clients, 8M transactions, 50%– 80% writes	
C-tree	4 clients, 100 INSERT transactions	
Hashmap	4 clients, 100 INSERT transactions Insert if absent, remove if found	
Memcached	memslap/4 clients,100K ops, 5% SET	

B. Benchmarks

Both microbenchmarks and the Whisper persistent benchmark [39] are evaluated in our experiments. To verify the effectiveness of barrier epoch management strategy, we use microbenchmarks to evaluate the persistent memory system throughput in local node (NVM server). Specifically, we repeatedly update persistent memory through various data structures including hash table, red-black tree, array, B+tree, and graph. These data structures are widely used in related applications such as databases and file systems. Table IV describes the details of these benchmarks. These benchmarks are compiled in native x86 and run on the McSimA+ simulator. Then we gather the remote memory access traces of these benchmarks and feed them into McSimA+ simulator to simulate the memory performance with local and remote requests as inputs.

To evaluate the overall system performance with network persistent requests, we implement experiments based on Whisper [39], the configuration of which is shown in Table IV.

VII. Results

The effectiveness of the proposed architecture with BROI controller is evaluated in this section. The system performance evaluation consists both of the local and remote applications. For the local applications (in the NVM Server), our solution increases the memory throughput by providing more BLP for memory controller, which leads to better application throughput. For the remote applications (in the Client node), our solution reduces the response latency of the persistent requests through network, thus improving the remote application throughput.

A. Local Application Performance

In this section, we evaluate the performance of NVM servers for the scenarios with local persistent requests and hybrid persistent requests (from both server and client). We compare the performance between the following two design philosophies: *Epoch* and BROI-enhanced management methodology (abbreviated as *BROI-mem*). A detailed description is listed in the following.

- ***Epoch***: using delegated ordering and adopting buffered persisting method to optimize for relaxed epoch size [25].
- ***BROI-mem***: using BROI-enhanced delegated ordering to utilize inter-thread persistence parallelism with consideration for bank-level parallelism.

Memory System Throughput. We first compare the memory system throughput (data volume per second in memory bus) using the pre-mentioned two strategies. The comparison result is shown in Figure 9, where the *local* refers to the scenarios with only local requests and *hybrid* refers to the scenarios with both local and remote requests. The x-axis lists the benchmarks for evaluation. The y-axis is the throughput ratio normalized to the *Epoch-local* scenario. There are following observations: 1) Compared to the *Epoch* solution, the *BROI-mem* achieves an improvement of 16% and 18% for *local* and *hybrid* scenarios respectively. 2) The *Hybrid* scenarios have larger memory throughput due to larger epoch size and sequential access patterns of remote requests.

Since the memory throughput is the average data volume transmitted through memory bus during the whole benchmarks execution, including the abundance of bus idle time, it might be imprecise to indicate the overall performance with memory throughput. Therefore, we further evaluate the application operational throughput to validate the effectiveness of BROI controller.

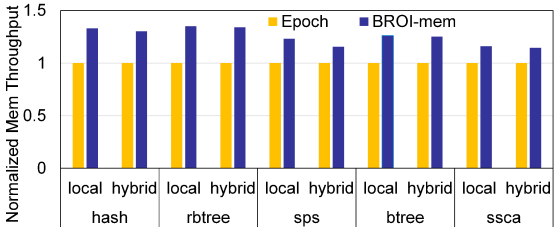

Figure 9. Memory system throughput.

Application Operational Throughput. The evaluation of the application operational throughput is as shown in Figure 10. The x-axis lists the evaluated benchmarks. The y-axis shows the operational throughput (Mops, millions of operations per second). The *ssca* has much higher operational throughput than the others because it is less memory-intensive. There are following observations. First, *BROI-mem* improves application operational throughput significantly. Under scenarios with *local* and *hybrid* persistent operations, *BROI-mem* improves performance of local applications by 28% and 30% compared to the *Epoch* method. The results show the effectiveness of BLP-aware barrier epoch management that improves application performance significantly, in both memory-intensive and non-memory-intensive cases. The operational throughput is calculated as the transactions numbers divided by the execution time, hence the execution latency and throughput evaluation are comparable here. We conclude that BROI controller will significantly improve local application performance by implementing the BLP-aware barrier epoch management.

We conduct the scalability experimental study on hash, the results of which is shown in Figure 11. The configuration of processor core number and BROI queue size are shown in the table. Every core supports two-way SMT (simultaneous multi-threading). The results show that our approach can support good scalability of performance with affordable hardware overhead.

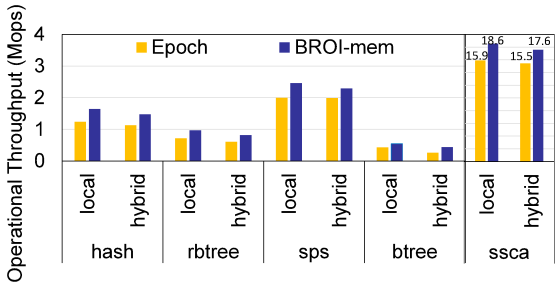

Figure 10. Local application operational throughput.

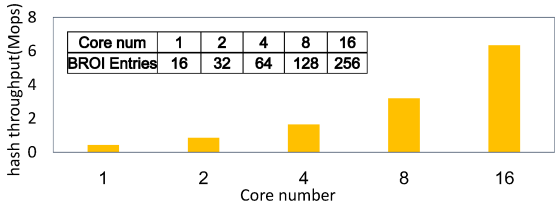

Figure 11. Scalability.

B. Remote Application Performance

In this section, we evaluate the effectiveness of BSP to improve the network persistence performance of Whisper benchmarks.

We analyze the performance of *ycsb*, *tpcc*, *memcached*, *hashmap*, and *ctree* from whisper benchmarks, under the following two persisting strategies: 1) persisting every network packet in synchronous way, which is referred to as *Sync*; 2) persisting multiple network packet in order with assist of buffered strict persistence design (BSP). The system throughput comparison between *Sync* and *BSP* is shown in Figure 12. For *tpcc* and *ycsb*, *BSP* improves by 2.5x. These results prove

that network persistent latency has great impact on application performance.

For *memcached*, *BSP* only performs 15% better then the *Sync*, because most of the operations in *memcached* are read rather than write operations. For *hashmap*, *ctree*, *ycsb*, and *tpcc*, *BSP* method achieves 2x times improvement over baseline, with both small or large input dataset of persistent requests. The results show that the BSP improves performance of remote applications with assistance of buffered epoch persistence support. When there are less persistent requests, the application is latency sensitive, *BSP* decreases the valid transportation latency for transactions. When there is heavy pressure for persisting, *BSP* increases the bandwidth utilization of the network and reduce the response latency of the remote requests, leading to better performance.

We also make sensitive study with the varying size of data elements. Taking *hashmap* as an example, the throughput with varying element size is shown in Figure 13. The BSP is effective with varying data element size from 128B to 4096B. When the data size is continue increasing, the bottleneck would be network bandwidth and BSP is less effective. However, according to the technique trend, there would be large amount of small epochs in the remote persistent memory [39], [47].

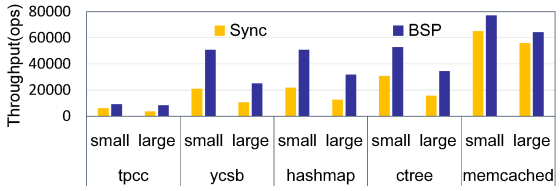

Figure 12. Remote application operational throughput.

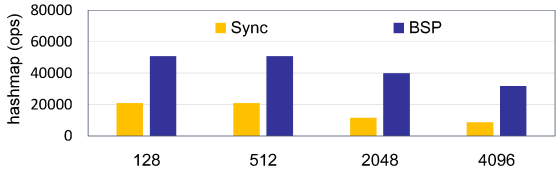

Figure 13. hashmap throughput with varying data element size in one epoch.

VIII. Related Work

In this session, we briefly discuss related work beyond those covered in details in Section I and Section II:

Memory Scheduling. Previous memory scheduling work trY to alleviate the performance degradation of persistent ordering. NVMDuet [32] decouples the memory scheduling between normal requests and non-volatile requests to reduce the performance degradation caused by persistent data. FIRM [58] classifies the memory requests into four categories in terms of access pattern, such as row buffer locality, streaming and bank parallelism. This work recognizes that the normal requests have no need to follow persistent ordering, but ignores the parallelism in persistent requests. NVM server would likely be used for high-performance data service for handling a large volume of persistent requests. Hence, the abundant persistence parallelism in NVM is important for performance and should be carefully managed.

Remote Memory System. Driving by the demand for high reliability, availability, and serviceability of data services, the enterprise, data-center, and visualization applications make multiple versions of data replica. To provide faster data replica, which is in the critical path of system, many academic and industry researchers contribute to the remote NVM system designs in observing the opportunity of the remote NVM devices which has much faster speed than the local SSD storage [6], [18], [21], [34], [44], [48]. These work can be classified into two categories: 1) the architecture and system support for remote memory persistence. The Storage Networking Industry Association (SNIA) has proposed remote persistent memory programming model [49]. The system companies discussed about the system solution for the memory persistence through RDMA [17], [43], [47], [48]. 2) The database and distributed persistent memory designs with the support of remote memory persistence. Previous work proposes distributed persistent memory system [44], the file system [21], and database [6], [46] based on remote NVM. Our work improves the fundamental capability of network persistence which is important for the applications and systems based on remote persistent memory.

IX. Conclusion

NVM technologies incorporate both the features of high-speed byte-addressability and disk-like data persistence, which are beneficial for data services such as file systems and databases. To support data persistence, a persistent memory system requires sophisticated data duplication and ordering control for write requests. We observe that the memory bus and networks are significantly underutilized during the data persistence. In response to the inefficiencies across the memory bus and network, we establish the importance of managing the Barrier Region of Interest (BROI) to improve memory and network parallelism. We propose an architecture design with a BROI controller to assist persistence. The BROI controller schedules persistent requests based on BROI and data location to maximize Bank Level Parallelism (BLP) in the memory controller without sacrificing system correctness. It also supports fine-grained memory persistence through the network with multiple epochs in a single network packet to boost the network utilization and reduce the response time. Our results demonstrate that with the BROI controller, the system can achieve 1.3× performance improvement in memory persistence compared to baseline systems and 1.93× the performance in network persistency.

X. Acknowledgement

We thank Yiying Zhang and Xiaoyi Lu for their insightful comments and suggestions. We thank the anonymous reviewers for their valuable feedback. This work is supported in part by NSF 1730309, 1719160, 1500848, 1652328, 1817077, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

REFERENCES

[1] NRCIO: NVM-aware RDMA-based Communication and I/O Schemes for Big Data. 2017 Nonvolatile Memories Workshop.

[2] Intel Data Direct I/O Technology (Intel DDIO). http://www.intel.com/content/dam/www/public/us/en/documents/technologybriefs/data-direct-i-o-technology-brief.pdf.

[3] J. H. Ahn, S. Li, S. O, and N. P. Jouppi. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85, April 2013.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modeling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Program Language Systems*, 36(2):7:1–7:74, July 2014.

[5] Joy Arulraj and Andrew Pavlo. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 1753–1758. ACM, 2017.

[6] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 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 (SIGMOD)*, pages 707–722. ACM, 2015.

[7] David A. Bader and Kamesh Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HPC)*, pages 465–476, 2005.

[8] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HOTOS)*, pages 2–2, 2011.

[9] Timo Bingmann. STX B+ Tree, Sept. 2008, http://panthema.net/2007/stx-btree.

[10] D. Castro, P. Romano, and J. Barreto. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[11] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMOVE: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2016.

[12] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, pages 26:1–26:17, 2016.

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

[14] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

[15] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

[16] K. Doshi, E. Giles, and P. Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89, 2016.

[17] Chet Douglas. RDMA with PMEM, software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2015.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, pages 15:1–15:15, 2014.

[19] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *in Proceedings of VLDB Endowment*, 8(4):389–400, 2014.

[20] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 50–59, 2005.

[21] Nusrat Sharmin Islam, Md. Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing (ICS)*, pages 8:1–8:14, 2016.

[22] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–465, 2018.

[23] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 660–671, 2015.

[24] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 13:1–13:16, 2016.

[25] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *in proceedings of th 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[26] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 481–493, 2017.

[27] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2009.

[28] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 327–336, 2009.

[29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2017.

[30] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 355–370, 2016.

[31] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 476–488, 2015.

[32] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–470, 2014.

[33] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223, 2014.

[34] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 773–785, 2017.

[35] Youyou Lu, Jiwu Shu, and Long Sun. Blurred Persistence: Efficient Transactions in Persistent Memory. *ACM Transactions on Storage*, 12(1):3:1–3:29, 2016.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[37] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[38] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *2008 International Symposium on Computer Architecture (ISCA)*, pages 63–74, 2008.

[39] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 135–148, 2017.

[40] Jiaxin Ou and J. Shu. Fast and failure-consistent updates of application data in non-volatile main memory file system. In *the 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–15, 2016.

[41] J. T. Pawlowski. Memory as we approach a new horizon. In *the 28th IEEE Hot Chips Symposium (HCS)*, pages 1–23, 2016.

[42] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, ISCA '14, pages 265–276, 2014.

[43] Andy Rudoff. Processor Support for NVM Programming. http://www.snia.org/sites/default/files/AndyRudoff_Processor_Support_NVM.pdf, 2015.

[44] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the ACM Symposium on Cloud Computing*, 2017.

[45] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 175–186, 2017.

[46] Swaminathan Sundararaman, Nisha Talagala, Dhananjoy Das, Amar Mudrankit, and Dulcardo Arteaga. Towards Software Defined Persistent Memory: Rethinking Software Support for Heterogenous Memory Architectures. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, pages 6:1–6:10, 2015.

[47] Tom Talpey. Low latency remote storage: a full-stack view. http://www.snia.org/sites/default/files/SDC/2016/presentations/persistent_memory/Tom_Talpey_Low_Latency_Remote_Storage_A_Full-stack_View.pdf, 2016.

[48] Tom Talpey. RDMA extensions for remote persistent memory access. https://www.openfabrics.org/images/eventpresos/2016presentations/215RDMAforRemPerMem.pdf, 2016.

[49] Tom Talpey. Persistent Memory Programming, the Remote Access Perspective. https://www.openfabrics.org/images/2018workshop/presentations/109_TTalpey_RemotePersistentMemory.pdf, 2018.

[50] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*, pages 5–5, 2011.

[51] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference (Middleware)*, pages 37–49, 2015.

[52] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.

[53] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware Supported Persistent Object Address Translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 800–812, 2017.

[54] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent Transactional Memory. *IEEE Computer Architecture Letters*, 14(1):58–61, 2015.

[55] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104. ACM, 2015.

[56] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong. Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 519–531, 2016.

[57] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, 2015.

[58] J. Zhao, O. Mutlu, and Y. Xie. FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 153–165, 2014.

[59] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 421–432, 2013.