i-BEP: A Non-Redundant and High-Concurrency Memory Persistency Model

Yuanchao Xu*†, Zeyi Hou*, Junfeng Yan*, Lu Yang*, Hu Wan*
*College of Information Engineering, Capital Normal University, Beijing, China
†State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
{xuyuanchao, houzeyi, yanjunfeng, yanglu, wanhu}@cnu.edu.cn

***Abstract*—Byte-addressable, non-volatile memory (NVM) technologies enable fast persistent updates but incur potential data inconsistency upon a failure. Recent proposals present several persistency models to guarantee data consistency. However, they fail to express the minimal persist ordering as a result of inducing unnecessary ordering constraints. In this paper, we propose i-BEP, a non-redundant high concurrency memory persistency model, which expresses epoch dependency via persist directed acyclic graph instead of program order. Additionally, we propose two techniques, background persist and deferred eviction, to enhance the performance of i-BEP. We demonstrate that i-BEP can improve the performance by 15% for typical data structures on average over buffered epoch persistency (BEP) model.**

I. Introduction

The emergence of non-volatile memory technologies like PCM and STT-RAM provides a fast, byte-addressable alternative for persistence, which can be achieved via load/store memory interface rather than block-based I/O interface of Flash and disk. In this single level store system, the data may stay at last level cache (LLC) or other volatile store buffers, where all the data will be lost upon a power failure or system crash. In addition, these data may be reordered in cache hierarchy or store buffers, thereby leading to different persist order from the desirable program order. As a result, these data structures cannot be recovered correctly from a failure, and then become inconsistent. Therefore, how to maintain the consistency of data structures in persistent memory becomes an important challenge.

To achieve this goal, programmers need to enforce flushing cache lines out of cache hierarchy to persistent memory by inserting sync instructions [1]. However, this method is inefficient, since it introduces too strict persist ordering. Several memory persistency models are proposed [2]–[4]. However, all these models fail to express the minimal persist ordering constraints because they couple persist order with program order, thereby inducing unnecessary persist order.

The aim of this work is to express the minimal persist ordering constraints for high concurrency. We propose a new implementation approach, i-BEP, which uses directed acyclic graph (DAG) instead of program order to express the minimal data dependency and semantic dependency among epochs. i-BEP decouples persist order with program order, thereby avoiding inducing unnecessary persist ordering constraints. In addition, in order to reduce the impact of long-latency online persists on performance, we present two techniques, *background persist* and *deferred eviction*, to move some online persists out of the critical path of program execution, thereby further improving persist concurrency. We evaluate i-BEP and demonstrate that it can improve performance by 15% for typical data structures on average over BEP model.

II. Background and Motivation

A. Single-level Store Architecture

Modern computers have generally adopted two-level storage model with separate main memory and persistent storage. The advent of NVM is driving a rethink of the fusion of working memory and persistent storage [5] by leveraging its DRAM-like byte-addressability and disk-like non-volatility. By using NVM, we can design a flat single-level store architecture, where working memory and persistent storage are located in the same level. We assume that cache is volatile and memory is non-volatile in our target system. DRAM may be integrated into our system to place hot data temporally in order to mitigate the endurance of NVM [13]. Even so, it does not affect the conclusion of this work.

In single-level store system, data are accessed via load/store instructions. Because memory consistency model only ensures the visibility of data, the durability of data is guaranteed via issuing sync instructions including `clflush`, `sfence`, and `pcommit` explicitly. However, using these instructions alone is hard to exploit the concurrency of persists due to too strict persist ordering. A relaxed persistency model is desired to achieve high persist concurrency besides from data durability and recoverability.

B. Problems and Solutions

As an extension to memory consistency model, memory persistency model is used to define persist ordering constraints, which not only has impact on the program execution but also enables correct recovery after a failure. Existing studies proposed several persistency models ranging from strict ordering constraints to relaxed ordering constraints [1]–[4]. Of these models, epoch persistency model is the most important, which has two rules: (**R1**) *the older epochs must be persisted before a new epoch continues to execute*. (**R2**) *a new epoch may not be persisted until all the older epochs have already been persisted*. Although **R1** is not required by BEP model [4], **R2** is still needed, thereby inducing unnecessary persist ordering. The reason is that the older epochs in program order may be independent with the new epoch from the respective of persist order.

For example, we assume there is a thread with six epochs, which can be depicted as a directed acyclic graph as illustrated in Fig. 1(left) according to epoch dependency. Each node denotes an epoch where each directed edge represents happens-before relationship between two epochs. If we use BEP model to describe epoch dependency of this thread, there are three possible scenarios, which have not all the same program order as shown in Fig. 1(right). They differ in the order of three epochs including E20, E21, and E22 and the location of persist barriers. Unfortunately, all of them induce unnecessary persist ordering constraints. The program order in Fig. 1(a) incurs extra ordering constraint between E20 and E22. The program order in Fig. 1(b) incurs extra ordering constraint between E21 and E20. The program order in Fig. 1(c) incurs extra ordering constraints among E20, E21, and E22.

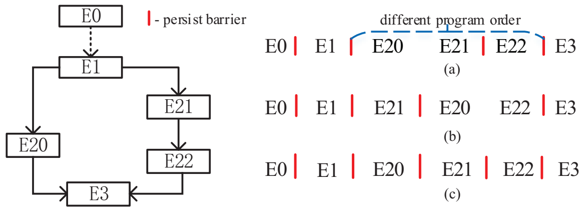

Fig. 1. An example that existing model introduces unnecessary persist ordering constraints for three different program orders with persist barrier.

It indicates that BEP model fails to express the minimal persist ordering constraints. The fundamental reason is that it uses **program order** rather than **persist order** to reason about persist dependency among epochs. To address this issue, we propose *i-BEP*, a non-redundant high concurrency buffered epoch persistency model, where we rely on persist directed acyclic graph (**pDAG**) instead of program order to express epoch dependency. pDAG unifies inter-thread and intra-thread dependency. Different from previous work, our main goal is to exploit persist concurrency and also explore mechanisms to mitigate the overhead of online persists.

III. i-BEP Design and Implementation

In this section, we first give the key points of i-BEP and then describe other techniques including background persist and deferred eviction.

A. Programming Model

Store and Persist. Recent research proposals [2], [4], [5] do not differentiate *store* from *persist*. Actually, not all the data need to be persisted. In single-level store systems, store is by default considered as volatile execution. Persist is required to adopt further operations, either passive eviction or proactive flush, thus persist is higher in latency than store. Generally, each persist implies a store, except for those cases using non-temporal instructions or uncacheable caching mode. Since persist takes longer latency than store and also shortens the lifetime of NVM, we should try to reduce the number of persists as many as possible.

Epoch partitioning. The foundation of epoch persistency is epoch partitioning (i.e., where to place the persist barrier). The number of epochs and the locations of persist barriers have direct impact on persist concurrency [1], [5]. All persists within an epoch may be reordered and even coalesced, whereas persists across two epochs must be ordered. The newer dependent epochs may not be persisted until all the data within current epoch are persisted completely.

Epoch dependency derives from data dependency and semantic dependency at epoch granularity. Data dependency means that two accesses (at least one persist) to the same address must be ordered. If these two accesses come from different epochs, it indicates that these two epochs are dependent. Semantic dependency means that two persists to different addresses are also required to be ordered if they are related to the same semantic, e.g., a transaction or a persistent data structure (e.g., hash table). Both data dependency and semantic dependency indicate that these two epochs have happens-before relationship from the perspective of persist.

Consider the example shown in Fig. 2. We assume that all stores are required to be persisted. There are two data dependencies between E01 and E10, E01 and E02. Semantic dependency exists between E11 and E12.

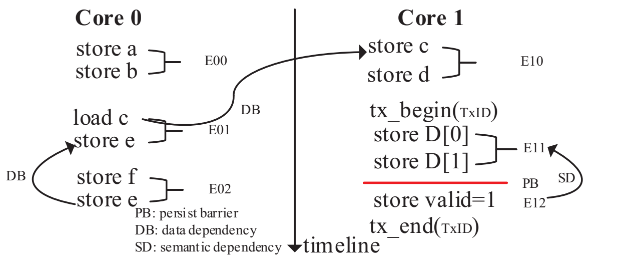

Fig. 2. An example of illustrating data dependency and semantic dependency.

B. Dynamic pDAG

The partial order over all persists across threads at epoch granularity can be expressed as a pDAG. If the indegree of a node is zero, it indicates that those older dependent epochs all have already been persisted or this is the first epoch after reboot, thus current epoch may be persisted. If all stores from two epochs are globally visible and meanwhile their indegrees are both zero, they can be persisted concurrently. In short, persist order depends on pDAG rather than program order.

We make a comparison between i-BEP and BEP as illustrated in Fig. 3. We can observe that, after epoch E0 and epoch E1 are completely globally visible, there is no dependency between them. Hence they may persist concurrently (e.g., *b* and *c* may be persisted in parallel). However, under BEP as shown in Fig. 3(a), they must be persisted in order in terms of program order, thereby incurring unnecessary ordering constraints.

pDAG is maintained dynamically as a volatile data structure by LLC controller. It is driven by two events including store and persist (either passive eviction or proactive flush). When the first memory request of an epoch is issued, a new node is created in the pDAG. When the last persist of this epoch completes, the node is deleted from the pDAG. Correspondingly, all edges associated with this node are also deleted, thereby making other epochs possible to be persisted due to

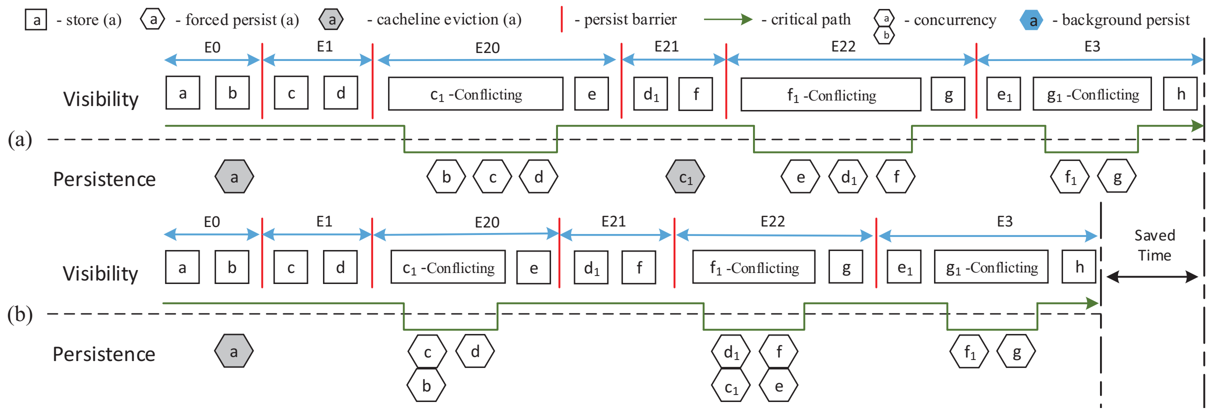
Fig. 3. An example of i-BEP compared with BEP. (a) Timeline of visibility and durability under BEP. (b) Expressing epoch dependency via pDAG.

the elimination of epoch dependency. Consider the example shown in Fig. 4. The pDAG goes through eight statuses (coarse granularity) with the arrival of memory requests according to Fig. 3(b). For instance, when CPU issues a new store *c* from E20, a conflict occurs, thus pDAG is transferred from status (2) to status (3).

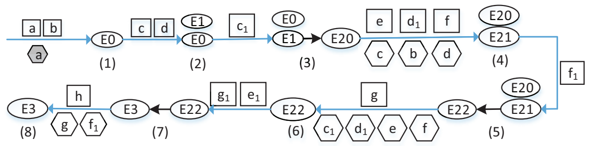
Fig. 4. An example of dynamic pDAG.

C. Background Persist

The execution time of a program is mainly subject to the length of critical path, which consists of online stores, online persists, and computing. Compared with volatile store, persist is a time-consuming process. Hence, we should move some online persists out of the critical path of program execution by making them overlap with other operations. We propose *background persist*, a technique enforcing dirty cachelines flushing back to NVM when memory bus is relatively idle. As shown in Fig. 5, *b* may be persisted in background ahead of time, thereby moving it out of the critical path. Cache controller will periodically check those dirty cache lines which do not have any dependency with other epochs. Once memory bus is idle, LLC controller will initiate background persist.

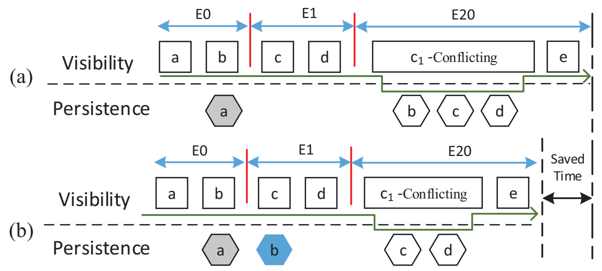
Fig. 5. Comparison of the timeline of visibility and durability. (a) Without background persist. (b) With background persist.

D. Deferred Eviction

Due to persist ordering constraint, a single event of cacheline eviction can trigger a series of proactive flush operations from the older dependent epochs. As illustrated in Fig. 6(a), if cacheline *d* is required to be evicted, cachelines *b*, *f*, and *c* need to be flushed to NVM first because a newer epoch cannot be persisted until all stores belonging to the older dependent epochs have been persisted. To address this issue, we propose *deferred eviction*, a viable method to reduce the number of proactive flushes by choosing a suitable cacheline as a candidate to be evicted. As illustrated in Fig. 6(b), we choose cacheline *b* instead of *d* as a candidate cacheline to be evicted, thereby making the critical path shorter.

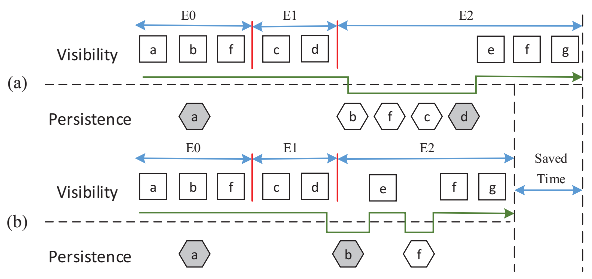
Fig. 6. An example of deferred eviction. (a) Under BEP, *d* is evicted. (b) Under i-BEP with deferred eviction, *b* is evicted.

The cache controller chooses a cacheline victim according to the following order. First, the cache controller checks whether there is an invalid cache line. If yes, return the cache line directly. Or else, continue. Then, it checks whether there is a clean cache line which belongs to an epoch whose indegree is zero. Next, it checks whether there is a dirty cache line which belongs to an epoch whose indegree is zero. Finally, it checks a dirty cache line which belongs to an epoch whose indegree is the smallest.

IV. Evaluation

A. Platform and Workloads

We evaluate i-BEP using gem5 [11] with Ruby in full system simulation mode. We evaluate a 16-core multicore with multi-banked LLC and 2 memory controllers. Each core has 32KB L1 D/I cache, 4-way, and 3 cycles. L2 cache is shared, 32MB, 16-way, and 30 cycles. NVM read/write latency is 240/320 cycles. We use several typical persistent data structures, such as Hash(insert/delete entries in a hash table), Queue(enqueue and dequeue), RBTree(insert/delete nodes in a red-black tree), and Arraylist (dynamic arrays that can grow as needed). We insert persist barriers at appropriate points to indicate a new epoch.

B. Performance Comparison

We choose BEP model as the baseline. We evaluate the performance improvement of i-BEP over BEP by using our proposed three techniques including pDAG (+pDAG), background persist (+BP), and deferred eviction (+DE) optimizations individually over the top of BEP. i-BEP is a result of combining all three techniques.

Fig. 7 shows the normalized transaction throughput of the aforementioned systems using different workloads. On average, BEP+pDAG improves throughput by 9%. BEP+BP improves throughput by 13%. BEP+DE does not gain any performance speedup over BEP, since all the data should be persisted sooner or later no matter whether deferred eviction is adopted. But, deferred eviction reduces the number of online persists. Only if these persists are then done by background persist, the critical path of program execution will become shorter. Therefore, deferred eviction should be used together with background persist. By putting these techniques together, i-BEP can achieve an improvement of 15% over BEP in throughput. Of these workloads, Queue is the most sensitive structure to our proposal since enqueue and dequeue operate at different addresses.

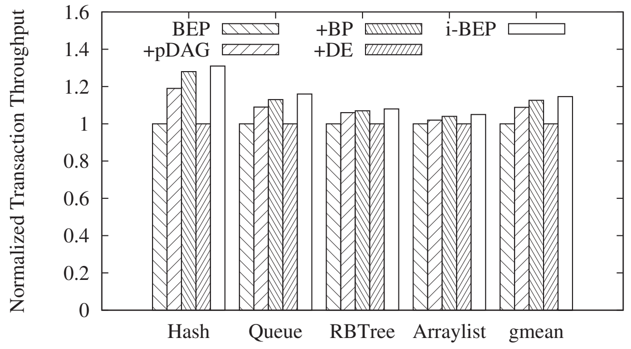

Fig. 7. Transaction throughput normalized to BEP for workloads.

C. Hardware Overheads

pDAG is used to express and track the dependency among epochs, thus pDAG itself needs not to be persisted. Though the size of pDAG is related to the number of concurrent running applications, pDAG is dynamically maintained. Because store and persist are constrained mutually due to epoch dependency, the size of pDAG is not likely too large. In short, the cost of i-BEP is comparable with BEP.

V. Related Work

Persistency model. Condit [6] proposed epoch barrier in order to improve persist concurrency within an epoch. Pelley [2] explored several persistency models. However, hardware implementation is not discussed. Joshi [4] proposed an improved epoch persistency model and discussed its implementation. But this model still incurs unnecessary persist ordering. Kolli [3] proposed delegated ordering, wherein ordering requirements are communicated explicitly to the PM controller. All of them do not study how to express the minimal persist ordering constraints, which is a central theme of our work.

Other work. Kolli [1] enabled fast persistence via sync instructions including `clwb` and `pcommit`. LOC [8] satisfied persist ordering constraints at lower performance degradation. Kiln [9] reduced persist latency by eliminating logging using a non-volatile LLC. Lu [14] proposed to blur the volatility-persistence boundary to achieve fast transaction processing. NVM Duet [5] optimized memory scheduling algorithms to improve the performance of persistent applications. Sun [12] proposed differential persistency and dual persistency. Most of these techniques are orthogonal to ours.

VI. Conclusion

Persist concurrency is closely related to persist ordering constraints. We propose pDAG to track data dependency as well as semantic dependency among epochs, thereby eliminating unnecessary persist ordering constraints. We also present two techniques including background persist and deferred eviction to improve persist concurrency by moving some online persists out of the critical path of program execution. We evaluate and demonstrate that, compared to BEP, i-BEP can improve the performance by 15% for typical data structures on average.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by State Key Laboratory of Computer Architecture under Grant No. CARCH201503.

References

[1] A. Kolli, S. Pelley, A. Saidi, *et al.*, "High-performance transactions for persistent memories," in *ASPLOS*. ACM, 2016, pp. 399–411.
[2] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*. IEEE, 2014, pp. 265–276.
[3] Kolli, A., Rosen, J., Diestelhorst, *et al.*, "Delegated persist ordering," in *MICRO*. ACM, 2016.
[4] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *MICRO*. ACM, 2015, pp. 660–671.
[5] R.-S. Liu, D.-Y. Shen, C.-L. Yang, *et al.*, "NVM Duet: Unified working memory and persistent store architecture," in *ASPLOS*. ACM, 2014, pp. 455–470.
[6] J. Condit, E. B. Nightingale, C. Frost, *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*. ACM, 2009, pp. 133–146.
[7] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency: Semantics for byte-addressable nonvolatile memory technologies," *IEEE Micro*, no. 3, pp. 125–131, 2015.
[8] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *ICCD*. IEEE, 2014, pp. 216–223.
[9] J. Zhao, S. Li, D. H. Yoon, *et al.*, "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO*. ACM, 2013, pp. 421–432.
[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, *et al.*, "System software for persistent memory," in *EuroSys*. ACM, 2014, pp. 1–15.
[11] N. Binkert, B. Beckmann *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
[12] L. Sun, Y. Lu, and J. Shu, "DP^2: reducing transaction overhead with differential and dual persistency in persistent memory," in *CF*. ACM, 2015, p. 24.
[13] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: persistent virtual memory for efficient capacity scaling and object storage," in *EuroSys*. ACM, 2016, Article: 13.
[14] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *MSST*. IEEE, 2015, pp. 1–13.