Efficient Persist Barriers for Multicores

Arpit Joshi
University of Edinburgh
arpit.joshi@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Marcelo Cintra
Intel, Germany
marcelo.cintra@intel.com

Stratis Viglas
University of Edinburgh
sviglas@inf.ed.ac.uk

ABSTRACT

Emerging non-volatile memory technologies enable fast, fine-grained persistence compared to slow block-based devices. In order to ensure consistency of persistent state, dirty cache lines need to be periodically flushed from caches and made persistent in an order specified by the persistency model. A persist barrier is one mechanism for enforcing this ordering.

In this paper, we first show that current persist barrier implementations, owing to certain ordering dependencies, add cache line flushes to the critical path. Our main contribution is an efficient persist barrier, that reduces the number of cache line flushes happening in the critical path. We evaluate our proposed persist barrier by using it to enforce two persistency models: buffered epoch persistency with programmer inserted barriers; and buffered strict persistency in bulk mode with hardware inserted barriers. Experimental evaluations using micro-benchmarks (buffered epoch persistency) and multi-threaded workloads (buffered strict persistency) show that using our persist barrier improves performance by 22% and 20% respectively over the state-of-the-art.

Keywords

non-voaltile memory, data persistence, persist barrier, multicore

1. INTRODUCTION

Recoverability in case of system crashes has been an important problem for programmers and system designers. To recover from a crash, a consistent state of program data has to be maintained in some persistent storage media. Traditionally disks were the only mode of persistence, which led to the development of recoverability techniques designed for slow block based devices. But the emergence of non-volatile memory (NVRAM) technologies like Phase Change Memory (PCM) and Spin-Transfer Torque RAM (STT-MRAM) provide a fast, byte-addressable alternative for persistence.

Since NVRAM sits on the memory bus, existing programs can use NVRAM for persistence using CPU load and store instructions. Therefore, using NVRAM for persistence avoids the hardware (I/O device delay) and software (legacy file system interfaces) overhead for persistence, which can greatly improve application performance. An important challenge in designing NVRAM based persistent memory is maintaining the consistency of data structures in persistent memory. Consistency of data structures is required to ensure correct recovery of program state after a crash. For example, consider a program which, while adding a node to a linked list, first writes to the new node and then updates a pointer to point to the new node. At the time of failure, the cache might have flushed the pointer write to persistent memory, but the write to the node might still be in the cache leading to an inconsistent state of the data structure in persistent memory. Hence, to ensure consistency of memory state, the correct ordering of cache line flushes needs to be enforced. Most modern processors reorder writes to memory at multiple levels (e.g., load-store queue, cache hierarchy, memory controller) to optimize performance. Because of this reordering, a system failure might leave the data structures in persistent memory in an inconsistent state.

To ensure consistency of persisted data, a mechanism like a persist barrier is needed to enforce the ordering of writes to NVRAM. A persist barrier will ensure that stores appearing before the barrier persist before the stores appearing after the barrier. One way to implement a persist barrier is by using existing instructions like *clflush* and *mfence* [1] [3, 4, 5, 6, 7]. However, this implementation tightly couples visibility and persistence. This coupling forces persistence (e.g., cache line flushes) to happen in the critical path of execution, which can

MICRO-48, December 05 - 09, 2015, Waikiki, HI, USA

ACM 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830805

[1] Although these instructions ensure the correct order of cache line flushes, they provide no guarantees on the order in which these cache lines persist (are written to NVRAM by memory controller) [1]. For this, additional instructions like the new *pcommit* [2] need to be used.

lead to significant performance degradation [8].

Condit et al. [9] propose hardware support for realising an improved persist barrier, that enforces persist ordering lazily. We refer to this barrier as *Lazy Barrier* (LB). Their key idea is to decouple visibility from persistence, allowing program execution to continue beyond the persist barrier, without waiting for stores from previous epochs[2] to persist; their memory system ensures that stores persist in the correct order, out of the critical path.

In LB, the memory system delays the flushing of cache lines; a cache line is only flushed, either due to natural eviction (e.g., replacement), or due to a forcible eviction. Forcible evictions are required in case of *epoch conflicts*, where old epochs need to be flushed in the critical path to ensure consistency of data in persistent memory. For example, before a dirty cache line belonging to a newer epoch can be replaced, cache lines written in older epochs need to be flushed first. Thus, in case of an epoch conflict, the conflicting request has to wait until all the relevant epochs have been flushed to memory. Epoch conflicts bring persist ordering constraints, and consequently cache line flushes, back in the critical path. This again couples visibility with persistence for the duration of conflicts.

In this paper, we design and implement a persist barrier (LB++) which improves upon LB. We first categorize epoch conflicts into inter-thread and intra-thread conflicts. We then propose optimizations to reduce the overhead due to the conflicts. We propose an Inter-thread Dependence Tracking (IDT) mechanism for dynamically tracking inter-thread dependencies in hardware, which allows us to reduce the overhead of inter-thread conflicts. We then propose a Proactive Flushing (PF) scheme to flush epochs proactively as opposed to the reactive approach of LB. Once an epoch completes, the values of all its cache lines are final. PF exploits this property and starts flushing cache lines on completion of epochs. A related issue in multi-threaded programs is that deadlocks can occur on inter-thread epoch conflicts if the programmer does not correctly place persist barriers. We present a solution to break persistence deadlocks, by splitting epochs, on detecting scenarios which could potentially lead to deadlocks. Finally, we propose a detailed protocol for flushing epochs in the correct order for a system with multi-banked caches.

We demonstrate the efficacy of LB++ by employing it to enforce 2 *persistency models* [8]. First, we use it to enforce Buffered Epoch Persistency (BEP) [8]. Using micro-benchmarks we show that using LB++ (as opposed to LB) improves performance by 22%. Second, we show how LB++ coupled with logging support can be used to enforce Buffered Strict Persistency (BSP) [8] in bulk mode. Our experiments using a subset of PARSEC, SPLASH and STAMP benchmarks show that using LB++ allows us to support BSP with a 1.3× execution time overhead over a non-persistent execution, which is an improvement of 20% over LB.

[2]Epochs are instruction groups divided by persist barriers.

2. BACKGROUND

In this section, we first provide an overview of persistency models [8] and highlight the limitations of current implementation [9] of those models. We then present the system configuration that we consider for our work.

2.1 Persistency Models

Our focus in this paper is on leveraging NVRAM technologies to enable fast persistence of programs. In order to enable correct recovery, program state that is persistent in NVRAM needs to be in a consistent state. The definition of what constitutes a consistent state depends on the programming model or more specifically, on the *persistency model* [8]. One easy way to understand persistency models is to think about them in relation to memory consistency models. Just as consistency models allow us to reason about visibility of stores, persistency models allow us to reason about durability of stores. Pelley et al. [8] introduce 3 persistency models: Strict, Epoch and Strand persistency. Here we focus only on Strict and Epoch persistency.

Strict persistency (SP) couples memory persistency with memory consistency. So at the point of failure, whatever updates are visible are guaranteed to have been persisted. For example, Total-Store-Order (TSO) systems under strict persistency operate under the following rules: **S1.)** stores persist in program order and **S2.)** a store cannot be made visible until the previous store (in program order) has persisted. A sequence of stores to different cache lines under SP is shown in Figure 1(a). As shown in the figure, SP creates persist ordering constraints at the level of each store operation. Hence, caches effectively have a write-through behaviour. Essentially, these fine grained persist ordering constraints conflict with 2 key optimizations employed in most modern processors. First, multiple stores to a cache line are coalesced in the caches and only written back to memory on a cache line replacement. Under SP since a store operation cannot be issued until the previous store operation persists (rule S2) multiple stores to a cache line cannot be coalesced (as shown in Figure 1(a) for cache line a). Secondly, processors reorder cache line persists to improve performance by exploiting temporal and spatial locality. This reordering happens in caches as well as in memory controllers. But under SP, cache lines have to be flushed in program order (rule S1), eliminating any possible performance gain from reordering of writes to memory.

Epoch persistency (EP) relaxes persist ordering constraints compared to SP and enforces ordering at the granularity of *epochs* [9]. An epoch is a contiguous group of instructions which are demarcated using a primitive known as a *persist barrier*. A system with EP operates under the following rules, **E1.)** stores belonging to different epochs persist in the order of their respective epochs and **E2.)** a new epoch cannot begin until all stores belonging to the previous epoch have persisted. Thus, EP allows coalescing of stores and reordering of persists for stores belonging to the same epoch. A sequence of stores to different cache lines un-

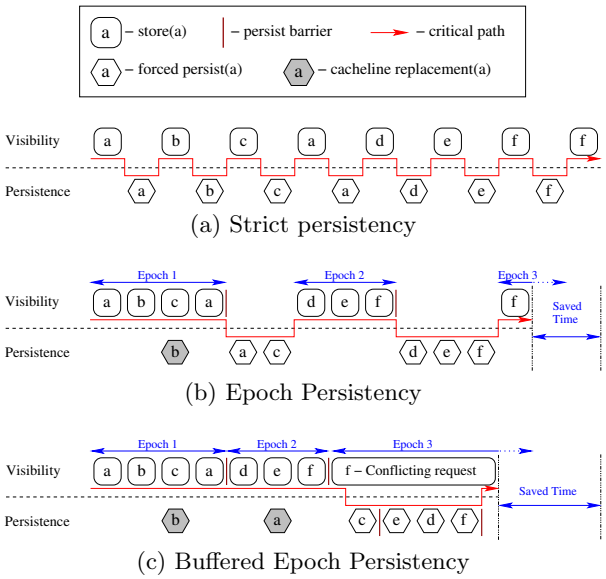


Figure 1: Timeline for completion of memory requests for various persistency models.

der EP is shown in Figure 1(b). As shown in the figure, EP allows coalescing for cache line a which reduces the overall time taken to complete the sequence of accesses compared to SP. Moreover, since cache lines belonging to the same epoch can persist out of order, cache line b can persist before cache line a. In EP, persist operations are in the critical path of execution upon completion of an epoch. Even though EP allows write coalescing and reordering of persists within an epoch, it still has a high performance overhead over volatile execution.

Buffered Epoch persistency (BEP)[3]: The fundamental reason for overhead in EP is that persist operations are in the critical path of execution (because of rule E2). BEP further relaxes constraint E2. Thus, BEP only requires that stores belonging to different epochs persist in the order of their respective epochs. BEP allows program execution to continue across epoch boundaries without waiting for previous epochs to persist. In this case, the cache sub-system has to ensure that epochs are flushed in correct epoch order. Figure 1(c) shows the timeline for a sequence of stores under BEP. Persist barrier after *Epoch*1 does not prevent *Epoch*2 from executing before all the cache lines modified by *Epoch*1 persist. While the program execution continues, modified cache lines can persist naturally because of replacement as shown in the figure for cache lines b and a. In BEP, persist operations are not in the critical path of execution as long as there are no epoch conflicts. An epoch conflict is a scenario where a memory request triggers an epoch flush. In Figure 1(c) a store to cache line f conflicts with *Epoch*2 because cache line f has been modified in *Epoch*2 which has not yet persisted. The store request has to wait until all epochs up to *Epoch*2 are flushed. Only epoch conflicts bring the persist operation in the critical path of execution for BEP.

[3]Pelley et al. [8] do not explicitly differentiate between epoch and buffered epoch persistency.

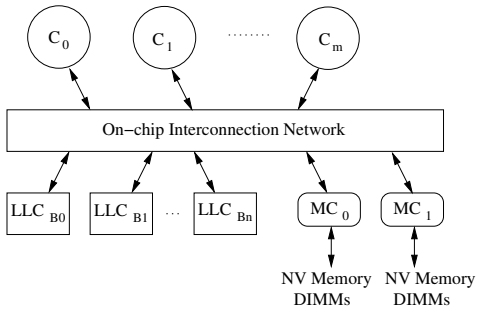


Figure 2: System Configuration: Multiple cores (C), a volatile shared multi-banked last level cache (LLC) and multiple memory controllers (MC) connected by an on-chip interconnection network.

The persist barrier proposed by Condit et al. [9] (LB) is basically an implementation of BEP. To track the current epoch, each core is extended with an *epoch ID counter* which is incremented by one each time a persist barrier is encountered. Whenever a store completes, it is tagged with the value of current epoch ID and core ID[4]. To track the status of cache lines, cache tags are extended to include epoch ID and core ID fields. Core ID identifies the core that last modified the cache line and epoch ID identifies the epoch in which the cache line was modified. Using these hardware extensions, cache controllers can track and enforce persist ordering dependencies between epochs belonging to the same core.

We call persists happening in the critical path as *online persists* and persists happening out of the critical path as *offline persists*. Online persists have a direct performance impact because they delay program execution while waiting for persist operations to complete. Offline persists on the other hand have no direct performance impact since they are not in the critical path. *The current implementation LB delays persist operations by buffering epochs and relies on offline persists in the form of natural cache line replacements. Although this design improves performance, this is not optimal for two reasons. First, conflicts trigger online persists thus delaying program execution. Second, this design does not actively reduce the number of online persist operations.* We elaborate on these limitations and present solutions in Section 3.

Buffered strict persistency (BSP) [8] is a result of relaxing constraint S2 from strict persistency. Although it will remove persistence from the critical path, the problems of not being able to coalesce writes and reorder persists would still remain. These problems in turn would trigger frequent conflicts resulting in a larger percentage of persist operations being online persists. We present an optimized implementation of BSP in bulk mode with logging support in Section 7.2.

[4]Condit et al. [9] partition the epoch ID counter and use the high order bits as core ID and remaining bits as epoch ID. We show them as 2 fields for the sake of simplicity.

2.2 System Configuration

We consider a multicore system as shown in Figure 2. In this system each core (C) has a private cache and all the cores share a multi-banked last level cache (LLC). All the caches are volatile. It has multiple memory controllers (MC) to provide sufficient memory bandwidth for large number of cores. These memory controllers are connected to NVRAM. This system is similar to most modern server processors, with the only difference being that memory in our system is non-volatile memory.

3. PERSIST BARRIER DESIGN

The goal of LB is to decouple persistence from visibility, which allows persist operations to happen out of the critical path. LB achieves this goal as long as there are no epoch conflicts. In this section, we first describe two types of conflicts and propose optimizations to reduce the overheads because of conflicts. We also illustrate the problem of epoch deadlocks and present a solution for the same.

3.1 Resolving Inter-thread Conflicts with IDT

An inter-thread conflict is a scenario where a thread tries to read or write to a cache line which has been modified by some other thread in an epoch which has not yet been flushed. Consider the example shown in Figure 3(a). Thread T_0 consists of epoch E_{00} and E_{01}. Thread T_1 consists of epoch E_{10} and E_{11}. T_0 tries to read address Y in epoch E_{01} after T_1 has written to it in epoch E_{11}. This creates a new persist ordering constraint that epoch E_{11} of thread T_1 should persist before epoch E_{01} of thread T_0. The epoch tracking hardware in LB can only track persist ordering constraints between epochs from the same core. Since this is an inter-thread ordering constraint, before completing *Ld Y* request from T_0, epoch E_{11} needs to be persisted; if not, epoch E_{01} might persist before epoch E_{11}, leaving persistent data in an inconsistent state. It is important to note here that a read request *Ld Y* creates an epoch conflict since it leads to persist ordering constraints which LB cannot track.

Inter-thread conflicts can lead to significant performance degradation as shown in Figure 4(a), which shows memory requests issued by 2 threads T_0 and T_1. T_1 issues request R_B to read cache line B. Since B has been modified by T_0 in epoch E_{00}, it triggers an inter-thread conflict which triggers online persist of E_{00}. This delays the completion of request R_B.

Inter-thread Dependence Tracking (IDT). If hardware support is provided for tracking inter-thread ordering constraints, the impact of inter-thread conflicts can be reduced. We define the epoch from which a request triggered an inter-thread conflict as *dependent epoch* and the epoch which last modified the requested cache line as the *source epoch*.

To avoid online persists of epochs in case of inter-thread conflicts, we propose a mechanism called IDT. On detecting a conflict, instead of waiting for the conflicting epoch to flush, IDT records source and dependent epochs and enforces this dependence offline. Thus a conflicting request does not have to wait for older epochs to persist. Figure 4(b) illustrates the possible performance improvement by using IDT. Request R_B from thread T_1 does not wait for the persist of epoch E_{00} to complete. IDT records the dependence between epochs E_{00} and E_{11} and allows request R_B to complete. When epoch E_{11} completes, cache line E is not allowed to persist until E_{00} has persisted. Thus the overall completion time is reduced while enforcing the correct persist ordering constraints.

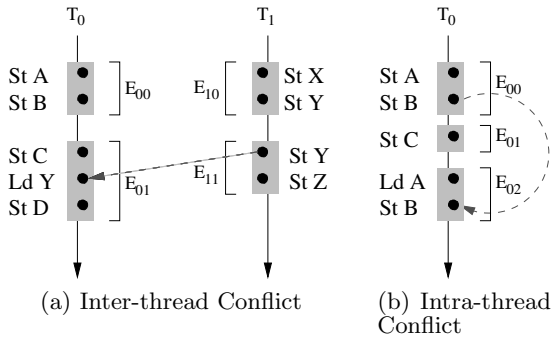


Figure 3: Examples illustrating epoch conflicts. (a) Highlights inter-thread conflict where epoch E_{01} tries to read cache line Y modified in epoch E_{11} (b) Highlights intra-thread conflict where epoch E_{02} tries to modify cache line B modified in epoch E_{00}.

3.2 Resolving Intra-thread Conflict with PF

An intra-thread conflict is a scenario where a thread tries to write to a cache line which it has already modified in some prior epoch and the cache line has not yet been flushed. Consider the example shown in Figure 3(b). Thread T_0 writes to address B in epoch E_{00}. It then again tries to write to the same address in epoch E_{02}. At this point in time the previous value of B has not yet persisted. If operation *St B* in epoch E_{02} completes, then the value of B will be overwritten. Now if the system crashes after persisting epoch E_{00} but before persisting epoch E_{01} then it will lead to an inconsistent state. To prevent this scenario, before completing *St B* in epoch E_{02}, epoch E_{00} needs to be persisted. It is important to note here that a read request (*Ld A*) does not create a conflict, as the persist ordering constraint between epochs within a thread is already being tracked by LB. On an intra-thread conflict, the epoch that last wrote to the conflicting cache line (B in the example) and all the epochs before it need to be flushed. The only way to minimize the performance impact of this type of conflicts is to minimize the number of such conflicts.

Proactive Flushing (PF). To mitigate the problem of intra-thread conflicts, we propose to persist epochs proactively. Proactive flushing would increase the number of epochs persisting offline. An intra-thread conflict happens because a cache line modified by some older epoch has not yet persisted because of natural cache line

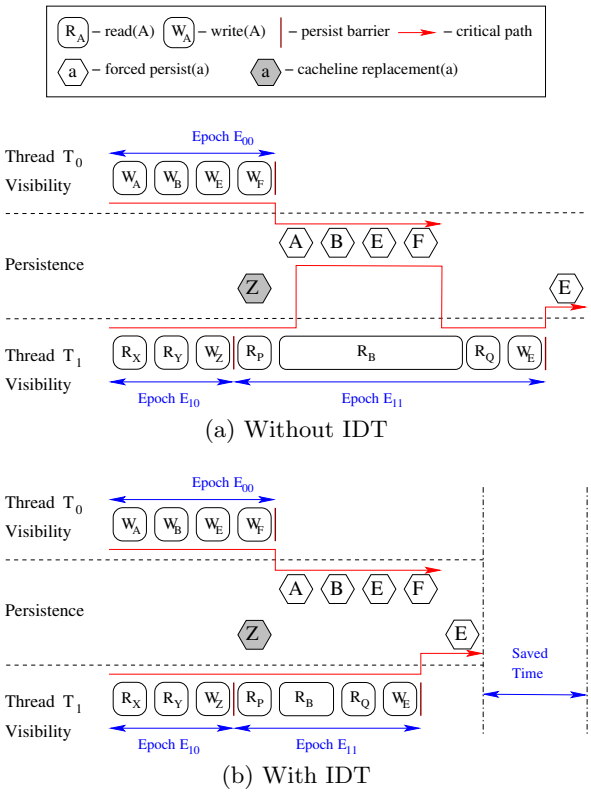


Figure 4: Example showing the benefit of IDT optimization. (a) Shows an example of how completion of conflicting requests is delayed waiting for persist of source epochs to complete. (b) Shows with the same example that by reducing the completion time of conflicting request and allowing the source epoch to persist offline, while enforcing persist ordering constraints, IDT improves performance.

eviction. By flushing a cache line proactively we reduce the probability of a conflict arising out of a subsequent access to it. This decrease in the probability of an intra-thread conflict results in improved performance. *It is worth noting that proactive flush will similarly reduce the probability of inter-thread conflicts too.* For ease of explanation, we have introduced it in relation to intra-thread conflicts.

While persisting epochs proactively, care also needs to be taken to ensure that we do not increase the number of flushes to memory. In other words, a cache line should be persisted only when its value is final. This will avoid multiple writes to memory for persisting the same cache line. Therefore, we propose persisting epochs proactively after epochs complete. Naturally, we cannot start proactive persist of an epoch if the previous epoch is not yet fully persisted.

An epoch persist operation consists of durably writing all the modified cache lines, belonging to the epoch being persisted, to memory. An important aspect when persisting epochs proactively should be to ensure that epoch persist operation does not invalidate cache lines.

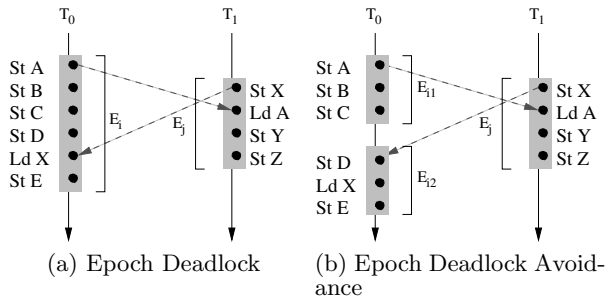


Figure 5: (a) Shows an example of persistent epoch deadlock. Epoch E_i and E_j belonging to threads T_0 and T_1 respectively have a circular dependence. E_j reads cache line A modified by E_i and E_i reads cache line X modified by E_j. (b) Possible epoch deadlock between epochs E_i and E_j is avoided by splitting the ongoing epoch E_i into epochs E_{i1} and E_{i2} on detecting conflict with epoch E_j.

If hardware employs mechanisms similar to *clflush* instruction, cache lines are also invalidated while being written back to memory. This will have a negative impact on the overall system performance as the persist operation will start evicting working sets from the cache. We implement a non-invalidating flush operation similar to the new *clwb* instruction [2]. This mechanism will not invalidate the cache line being persisted, thus avoiding any negative impact on performance.

3.3 Epoch Deadlocks and their Avoidance

In multi-threaded applications, epochs belonging to different threads are independent and can persist in parallel, as long as there are no inter-thread dependencies. In the presence of dependencies, epochs need to persist in the order specified by the dependencies. For the example shown in Figure 3(a), epochs E_{00} and E_{10} are independent and can persist in parallel. Whereas, epoch E_{01} is dependent on epoch E_{11}, so E_{01} cannot persist before E_{11} to ensure consistent state of memory. This dependence is enforced by the epoch persistence mechanism by either flushing epoch E_{11} before completing *Ld Y* request of epoch E_{01} or by tracking the inter-thread dependence relation between epochs E_{01} and E_{11} and ensuring that E_{11} persists before E_{01}.

The epoch persistence mechanism can enforce epoch ordering constraints when the dependence relation between epochs is linear. Consider the example shown in Figure 5(a). Epochs E_i and E_j have a circular dependence between them. On encountering *Ld A* request by T_1 LB identifies E_i to E_j dependence and will try to flush E_i before completing the request. But a flush for epoch E_i cannot complete because the epoch is ongoing. Then on encountering *Ld X* request by T_0 the epoch persistence mechanism tries to flush E_j before completing the request. This leads to a deadlock.

To prevent deadlocks, a scenario where a circular dependence relation arises needs to be avoided. We pro-

pose a solution to epoch deadlocks by conservatively preventing a scenario which can lead to circular dependence. This solution is based on the observation that circular dependence can only occur if a request triggers an inter-thread dependence with an ongoing epoch. By ongoing epoch we mean an epoch whose persist barrier has not yet occurred – in other words, an epoch which has not yet completed. If a request triggers an inter-thread dependence with a completed epoch, then there is no chance of having an inverse dependence since no memory operations are pending in the completed epoch. On detecting an inter-thread dependence with an ongoing epoch, our proposal is to divide the source epoch into two parts: the first part includes all the operations completed at the time of detection and the second part is the remaining portion of the epoch. Without IDT we would have had to flush the first part of the epoch, whereas with IDT it suffices to register the inter-thread dependence in hardware, before completing the request. It is worth noting that, by breaking the source epoch, we have ensured that there is no chance of having an inverse dependence. Figure 5(b) shows the solution with an example. When the first dependence is detected with respect to ongoing epoch E_i, E_i is split into epoch E_{i1} which consists of the part of the epoch that has already completed (until *St C*) and the remaining epoch which is called E_{i2}.

Discussion. To prevent epoch deadlock scenarios from happening, persist barriers need to be placed appropriately. One way to prevent epoch deadlocks from happening is to place persist barriers at the start and end of each critical section [10]. However, there can be programs where it is difficult to identify where to place epoch barriers to prevent deadlock (e.g., lock-free programs, programs with user-defined synchronisation, etc.) The deadlock avoidance scheme described above can be useful in such programs.

4. PERSIST BARRIER IMPLEMENTATION

In this section, we describe the implementation of our persist barrier. We first present a detailed epoch flush protocol that enforces persist ordering correctly in systems with multi-banked caches. We then describe how IDT and PF are implemented. We summarize by highlighting the additional hardware required for our persist barrier implementation.

4.1 Epoch Flush Protocol

To ensure consistency of data in NVRAM, epochs need to be persisted in the correct order. The union of the intra-thread program order and inter-thread shared memory dependencies define this epoch happens-before order. The goal of the epoch flush protocol is to ensure that order in which epochs are persisted is consistent with this happens-before order.

In the system presented in Section 2.2 caches are volatile, so persistence happens only when epochs have been written back to NVRAM. Hence, it is sufficient to ensure that for any two epochs E_1 and E_2 such that E_1 happens-before E_2, the last level cache (LLC) will not flush a cache line belonging to epoch E_2 until all the cache lines belonging to epoch E_1 have persisted.

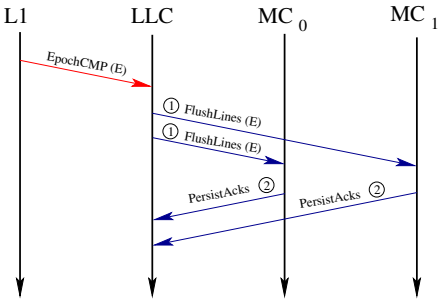
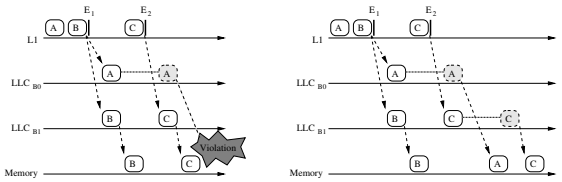


Figure 6: Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with monolithic last level cache.

To satisfy the above constraint LLC has to identify two pieces of information: first, the set of all the cache lines belonging to each epoch; second, it has to know when a cache line has persisted. These two pieces of information will help in identifying when an epoch has persisted, based on which LLC can potentially start persisting its successor(s). LLC needs to identify when the L1 cache has written back all the cache lines belonging to an epoch. To convey this information the L1 controller sends an *epoch completion* message (*EpochCMP*) to LLC after writing back all the cache lines belonging to an epoch. This informs the LLC that it has seen all the cache lines belonging to that epoch. It is important to note here that receiving an *EpochCMP* message for a given epoch is a prerequisite for completing the flush of that epoch. If LLC has to flush an epoch but has not received *EpochCMP* message for the same, it can request L1 to flush all the cache lines belonging to that epoch.

Monolithic LLC. If the LLC is monolithic, it can flush epochs independently after receiving *EpochCMP* messages for the epochs being flushed. Figure 6 illustrates the protocol. After flushing the epoch preceding epoch E, LLC in step ① starts flushing cache lines belonging to epoch E. Memory controllers respond with a *PersistACK* message after durably writing cache lines to NVRAM in step ②. On receiving *PersistACK* messages for all the cache lines flushed, LLC registers epoch E as having persisted and can start persisting the subsequent epoch. It is important to note that LLC has already received *EpochCMP* message for epoch E and hence it can consider the flush of E as having been completed.

Multi-banked LLC. The two step protocol presented above works for monolithic caches, but when extended to a system with multi-banked caches it might not work. Consider the example shown in Figure 7(a). The system consists of an L1 cache and two banks of *LLC*. Epoch E_1 consist of two cache lines A and B mapping to LLC_{B0} and LLC_{B1} respectively. Epoch E_2 consists of cache line C mapping to LLC_{B1}. L1 first flushes



(a) Violation. (b) Correct ordering.

Figure 7: (a) Shows an example of how epoch ordering constraint is violated. Cache line C belonging to epoch E_2 persists before cache line B belonging to the previous epoch E_1. (b) Shows the correct enforcement of epoch ordering constraints. LLC_{B1} delays persisting cache line C belonging to epoch E_2 until all the cache lines belonging to the previous epoch E_1 have persisted.

epoch E_1 and then epoch E_2. LLC_{B1} decides to flush epoch E_1 and hence flushes cache line B. Meanwhile LLC_{B0} delays flushing cache line A. When LLC_{B1} receives cache line C belonging to epoch E_2, it flushes the cache line. LLC_{B1} is allowed to flush epoch E_2, because all its cache lines belonging to previous epoch E_1 have already been flushed. This leads to a violation of epoch ordering constraints since a cache line belonging to epoch E_2 persists before the previous epoch E_1 is flushed completely. If the system crashes at this point, persistent memory will be left in an inconsistent state. The violation shown in Figure 7(a) happened because, in a multi-banked cache organisation, each bank only handles a range of addresses and has no information about the status of cache lines outside that range. In the example, LLC_{B1} had no information about the pending cache line belonging to epoch E_1 in LLC_{B0}. To avoid this scenario, a bank of LLC should not start flushing a cache line until all the banks have completed persisting the previous epoch. With this constraint, LLC_{B1} will not flush cache line C until LLC_{B0} has also flushed all the cache lines belonging to epoch E_1. This scenario where epoch ordering constraint is correctly enforced is shown in Figure 7(b), where LLC_{B1} does not flush cache line C belonging to epoch E_2 until LLC_{B0} has flushed cache line A belonging to epoch E_1.

To persist epochs in correct order all the banks of the LLC need to communicate with each other to coordinate flushing of every epoch. If all the banks send messages informing epoch completion to each other directly, it will require $\mathcal{O}(n^2)$ messages, where n is the number of banks. This can be a prohibitively large overhead, especially considering the fact that this will have to be incurred for each epoch that is being flushed. Instead we propose using an arbiter module to control flushing of epochs. All the banks will inform the arbiter when they have completed persisting an epoch. The arbiter in turn on receiving messages from all the banks will broadcast a message indicating that the relevant epoch has persisted. After receiving this message,

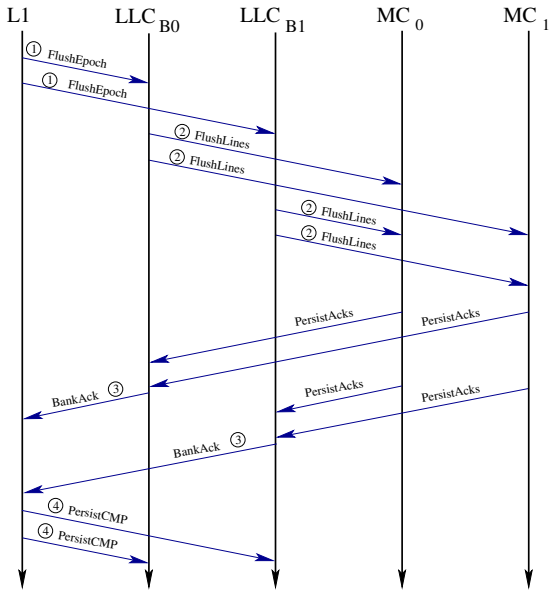


Figure 8: Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with multi-banked last level cache.

LLC banks can start flushing the next epoch. Using an arbiter in this way requires $\mathcal{O}(n)$ messages only. In a multicore, the arbiter can become a bottleneck if there is a single arbiter for persisting epochs belonging to all the threads. Instead we propose using a per thread arbiter, which is responsible for coordinating the persist operations belonging to a single thread. This per thread arbiter is placed along with private L1 caches in all the cores and is responsible for coordinating the persist of epochs belonging to the thread executing on that core.

We propose a handshaking protocol by using an arbiter module sitting in the L1 cache to orchestrate epoch flush. The protocol is shown in Figure 8. The arbiter in the L1 cache will start epoch flush by first flushing all the cache lines, belonging to the epoch being flushed, from L1. In step ①, L1 will flush all the cache lines belonging to that epoch to LLC and also send an epoch flush message to all LLC banks. In step ② each LLC bank will start flushing all the cache lines belonging to the epoch being flushed. On receiving *PersistAcks* for all the cache lines flushed, each LLC bank will send a *BankAck* message to the arbiter in the L1 controller in step ③. Finally in step ④, after receiving *BankAck* from all LLC banks, the arbiter will signal flush completion (*PersistCMP*) to all the LLC banks. This final step will update the state corresponding to last flushed epoch in all the banks.

4.2 IDT and PF Implementation

Enforcing inter-thread persist ordering constraints out of the critical path requires two things. First, preventing the dependent epoch from persisting before the source epoch persists. For this, an entry called dependence register is created in the arbiter corresponding to the dependent epoch. The arbiter before persisting an

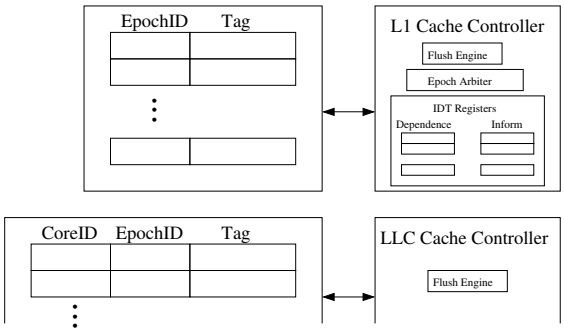
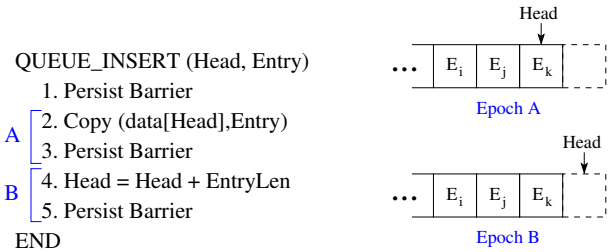


Figure 9: Hardware extensions.



(a) Queue insert pseudo-code (b) Example

Figure 10: (a) Pseudo-code for a queue insert operation using persist barriers for recovery in case of a system crash. (b) Example illustrating the status of the queue on completion of different epochs within the insert function.

epoch will check (in addition to its predecessor epochs in program order) the dependence register to see if that epoch is dependent on an epoch belonging to some other thread. If so, the arbiter will not flush until the source epoch has been flushed. The second aspect is to inform the dependent epoch when source epoch has been flushed. For this, an entry called inform register is created in the arbiter corresponding to the source epoch. On completing the persist for an epoch, the arbiter will send an epoch persist completion message to the dependent epochs listed in the inform registers.

To implement a proactive flushing scheme, once an epoch completes, a request is sent to the corresponding arbiter to start flushing the epoch. The arbiter starts flushing the completed epoch after ensuring that all its predecessor epochs have persisted.

4.3 Hardware Extensions

Hardware extensions required to implement a persist barrier are shown in Figure 9. To track the epoch status of each cache line, cache tags in both L1 and LLC are extended with EpochID. Cache tags in shared LLC need to be extended with CoreID information to detect inter-thread conflicts. Apart from the cache tags we add a *flush engine* in each cache controller to flush epochs as and when required. Flush engine will trigger a flush for the dirty cache lines of the epoch being flushed. An epoch arbiter is added in the L1 cache controller to coordinate epoch flush operation in the presence of multi-banked caches. To track inter-thread dependencies as proposed in IDT, we add a pair of registers called dependence and inform registers in L1 cache controller to identify the source and dependent epochs (using a combination of EpochID and CoreID). These registers are added per in-flight epoch.

In our implementation we support 8 in-flight epochs in a 32-core machine. So EpochID is 3 bits wide and CoreID is 5 bits wide. The overhead of tagging cache lines is 5 bits and 8 bits per cache line in L1 and LLC respectively. We add 4 pairs of IDT registers per epoch to allow tracking of as many inter-thread dependencies per epoch. The overhead of adding these registers is 64 bytes in each L1 cache. The per core arbiter contains a 5-bit counter to track *BankAck* messages received from all the banks. Even though multiple epochs can be in flight, only one of them will be flushed at a time; so one counter is sufficient. Our flush engine maintains bookkeeping information similar to [9], in order to reduce the overhead of searching. In our implementation, we maintain a bitmap per epoch, where each bit corresponds to 64 sets in the cache, amounting to an overhead of 512 bytes for 16-way 1MB LLC bank.

5. ENFORCING PERSISTENCY MODELS

In this section, we illustrate how our efficient persist barrier can be used to implement two models of persistency: Buffered Epoch Persistency (BEP) and Buffered Strict Persistency (BSP) in bulk mode.

5.1 Buffered Epoch Persistency

In BEP [8], programmer inserted persist barriers divide the program into epochs and persist ordering is enforced at epoch granularity. Consider the sample queue insert function (similar to [8]) shown in Figure 10(a). Queue insert operations consist of two epochs. In *Epoch A* from lines 2 to 3, a new entry is copied in the queue at the location pointed to by *Head* pointer. In *Epoch B* from lines 4 to 5, *Head* pointer is updated to point to the next empty location. Figure 10(b) shows the status of the queue on completion of Epoch A and Epoch B. If the system crashes after Epoch A persists but before Epoch B persists then the new entry E_k is ignored on recovery. If the system crashes after persisting Epoch B then on recovery the program will see successful completion of insert operation.

5.2 Buffered Strict Persistency in Bulk Mode

We enforce BSP in bulk mode, to minimize the overheads of strict persistency. Instead of enforcing persist ordering constraints at memory operation granularity, we enforce them at an epoch granularity. This is similar in spirit to the way Sequential Consistency is enforced by BulkSC [11].

BSP is implemented completely in hardware. Hence, no programmer annotations in the form of persist barriers are required. A hardware persistence engine divides the sequence of stores from a program execution into epochs. Persistency is enforced at the granularity

of epochs using the optimized persist barrier presented in Section 3. At the end of each epoch, along with the modified cache lines, processor state is also saved to persistent memory. This state can be used to restart the process, similar to the way it is done in [12]. It is worth noting that epoch boundaries are the points at which BSP holds. At the time of a crash though, some epochs might have persisted partially and therefore BSP might be violated. To overcome this problem we propose to use logging to undo any partially persisted epochs, in such situations.

Another issue that can arise in the proposed implementation is the possibility of epoch deadlocks. Since hardware dynamically creates epochs, it is oblivious to dependencies between threads. This could lead to epoch deadlocks. We use the solution presented in Section 3.3 to overcome this problem.

5.2.1 Logging

In order to enforce BSP, epoch updates to persistent memory need to be atomic. The granularity (atomic unit) at which memory can be updated by the hardware is typically much smaller than the size of an epoch (which spans multiple cache lines). To ensure atomic updates of an epoch, logging support is needed. This logging support can be provided by the hardware or can be explicitly managed in software through system libraries or by the application itself. In this work, we make use of a hardware undo logging mechanism, described below, to enforce BSP.

Undo logging requires that before modifying a cache line its old value should be written to the log area. Therefore, whenever a cache line is being modified for the first time in an epoch, the old value of the cache line (which is either already in the cache or has been brought into the cache on a cache miss) is written back from the cache to the log in NVRAM, before the modified cache line is written back. The first modification to a cache line can be identified by looking at the epoch identifier: if the epoch identifier of the cache line is the same as the epoch in which it is being modified, it means that this is not the first modification to the cache line in this epoch.

It is important to note however that our key contribution is persist barrier implementation and our work is orthogonal to the logging mechanism. In other words, our work can be equally applied with any kind of underlying logging infrastructure provided by the hardware or implemented in software.

6. EXPERIMENTAL METHODOLOGY

We evaluate our proposed persist barrier (LB++) using gem5 [13] with Ruby in full system simulation mode. The on-chip interconnect is modelled using Garnet [14]. We evaluate a 32-core multicore (1 thread per core) with multi-banked LLC and 4 memory controllers placed on 4 corners of the chip. Table 1 shows the parameters of the system.

Our aim is to evaluate BEP for applications that maintain persistent data structures (using micro-benchmarks) and BSP for long running applications that require checkpointing (using real workloads).

Cores	32 OoO cores @ 2GHz
ROB Size	192 Entry
Write Buffer	32 Entry
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×32 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
Memory Controllers	4
NVRAM Access Latency	360 (240) cycles write (read)
On-chip network	2D Mesh, 4 rows, 16B flits

Table 1: System Parameters

Hash	Insert/delete entries in a hash table
Queue	Insert/delete entries in a queue
RBTree	Insert/delete nodes in a red-black tree
SDG	Insert/delete edges in a scalable graph
SPS	Random swaps between entries in an array

Table 2: Micro-benchmarks used in our experiments

Workloads: We use micro-benchmarks listed in Table 2 to evaluate the proposal of using LB++ to implement BEP. These micro-benchmarks implement data structures that are similar to those in the benchmark suite used by NVHeaps [4], except for the queue micro-benchmark which is similar to the copy-while-locked queue presented in [8]. The size of data entry (table entries, tree nodes, queue entries etc.) for each micro-benchmark is 512 bytes. Each benchmark performs search, delete and insert operations on the corresponding data structure. We inserted persist barriers at appropriate points to ensure persistency (as illustrated in Figure 10(a)).

To evaluate buffered strict persistency (BSP) we use benchmarks from PARSEC [15], SPLASH-2 [16] and STAMP [17] benchmark suites. The benchmarks were unmodified; persist barriers are inserted transparently by the hardware to ensure BSP. In our experiments, we model the overhead of checkpointing all general purpose, special registers, privilege registers and floating point registers (non-AVX) as part of the processor state. We ran all the workloads to completion.

7. RESULTS

In this section we evaluate our key contribution, which is our proposed persist barrier (LB++). More specifically, we evaluate the additional speedup provided by LB++, over the state-of-the-art persist barrier LB [4], in enforcing BEP (Section 5.1) and BSP (Section 5.2). We also present performance improvements provided by inter-thread dependence tracking (LB+IDT) and proactive flush (LB+PF) optimizations individually on top of the unoptimized barrier. Recall that LB++ is a result of combining both IDT and PF optimizations.

Both optimized and unoptimized barrier implementations involve cache line flushes. Should the cache line

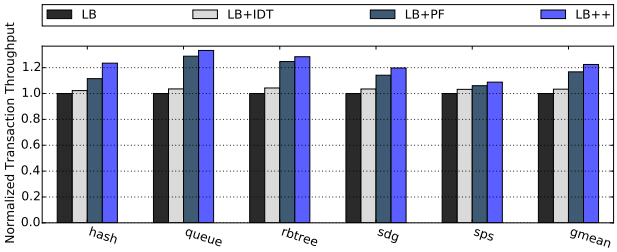


Figure 11: Transaction throughput normalized to LB.

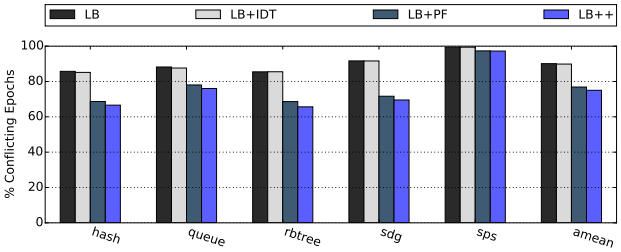


Figure 12: Percentage of conflicting epochs (out of the total number of epochs).

flush be an invalidating (similar to *clflush* instruction) or a non-invalidating flush (similar to recently introduced *clwb* instruction) [5]? We analyzed the performance impact and found that using a non-invalidating flush is significantly faster (around 30% faster). This is not surprising, since an invalidating flush would disrupt locality by evicting lines from cache, which on subsequent accesses need to be fetched again from NVRAM. We do not present detailed results owing to space constraints. *For the remainder of the section we only consider using non-invalidating cache line flushes to implement all persist barriers.*

7.1 Buffered Epoch Persistency

Impact of Optimizations. We study the performance improvement due to the two optimizations, first individually, and then in combination. Figure 11 shows the transaction throughput for micro benchmarks, normalized to throughput of LB. On average, LB+IDT improves throughput by only 3%. Recall that IDT improves performance by reducing the latency of memory requests that trigger inter-thread conflict. The primary reason why LB+IDT does not have a high performance improvement is because the performance is dominated by intra-thread conflicts for these microbenchmarks (it should be noted that IDT provides significant performance improvement for BSP as shown in Section 7.2). LB+PF on the other hand improves transaction throughput by 17%. This improvement is because LB+PF reduces the number of conflicts, thereby reducing the overall latency of memory requests. LB++, which is obtained by combining IDT and PF optimizations, achieves an improvement in throughput of 22% over LB.

Epoch Conflicts. In the presence of epoch conflicts persist operations happen in the critical path. This is contrary to the objective of LB, which is to perform offline persists. Figure 12 shows the percentage of epochs that are flushed because of a conflict. On an average 90% of epochs are flushed because of a conflict in LB. LB+IDT has a similar percentage of epoch conflicts, because IDT optimization does not directly impact the percentage of conflicting epochs but only reduces the latency of conflicting requests. PF optimization, on the other hand, decreases the probability of an epoch conflicting by persisting epochs proactively. Therefore, we can see that, on average LB+PF reduces the percentage of epoch conflicts from 90% to 77%. LB++ (LB+IDT+PF) reduces the epoch conflict percentage further down to 75%, as IDT can help PF. Recall that PF will start flushing an epoch only after the epoch completes. Since IDT allows epochs to complete faster, the scope of flushing epochs proactively increases.

[5]The reason for making this comparison is that many processors today only offer flush instructions that invalidate the cache line on completion.

7.2 Buffered Strict Persistency in Bulk Mode

In this section, we evaluate the performance overhead of achieving BSP in bulk mode when using LB++, in comparison to the overhead when using LB. Recall that we target the x86 architecture which supports a variant of TSO, hence the resultant persistency model is also the same. We compare the performance of achieving BSP relative to a baseline that provides no guarantees: No Persistency (NP). It is worth noting that NP also uses NVRAM as the memory and incurs NVRAM latencies. We used the NP baseline as it is interesting to know the overhead of enforcing BSP. A naive approach to implement BSP will require caches to be write through. We analyzed the performance of such a design and found it to be about 8× slower than NP. This is a prohibitively large overhead, therefore we do not consider (optimizing) this design further, and only present the bulk BSP results.

Epoch Size. In BSP, store operations are divided into epochs dynamically by hardware. How large should the epoch size be? Smaller epochs are desirable as that would mean lesser work lost, whereas larger epochs are expected to be more efficient. Therefore, we analyze the performance impact of varying epoch size; we consider sizes (dynamic stores) of 300 (LB300), 1000 (LB1K) and 10000 (LB10K). We use the unoptimized persist barrier (LB) for this study. Figure 13 shows the execution time of benchmarks for designs with varying epoch sizes normalized to NP.

We observe that, on average, performance improves with increasing epoch size. LB300 has an execution time overhead of 1.9×, whereas LB1K has a significantly lower overhead (40% reduction with respect to the baseline). This is because increasing the epoch sizes provides the opportunity for multiple writes to same cache line (belonging to the same epoch) to be coalesced, thereby decreasing the number of persists. For the same rea-

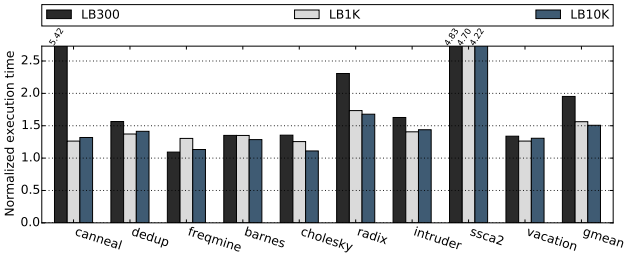


Figure 13: Execution time with varying epoch sizes normalized to NP.

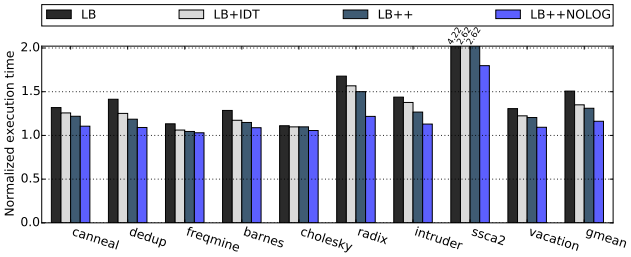


Figure 14: Execution time normalized to NP.

son, we observe that LB10K performs marginally better than LB1K – although interestingly on some benchmarks like canneal, dedup, intruder and vacation LB1K outperforms LB10K. We believe this is because epoch size increases the number of epoch conflicts, so with persist coalescing providing diminishing returns, epoch conflicts starts to dominate. This also explains why performance improvement saturates beyond epoch size greater than 10000 stores (not shown).

Impact of Proposed Persist Barrier LB++. The overhead of ensuring strict persistency using the unoptimized barrier LB is quite significant even with large epoch size (1.5×). This is the gap we are seeking to close with our optimized barrier. We consider an epoch size of 10000 for this study, as this is what gave the best results.

We observe (Figure 14) that LB+IDT reduces the overhead of strict persistency from 1.5× in LB to 1.35×. LB+IDT is able to achieve a 15% improvement because a large number (86%) of conflicts are inter-thread conflicts, which IDT is able to optimize on. LB++ further reduces the overhead to 1.3×, an improvement of 20% with respect to the baseline. The performance improvement is much more pronounced for some benchmarks. For instance, ssca2 sees a reduction from 4.22× to 2.62×; ssca2 is a write intensive benchmark with fine grained interaction between threads and the number of epochs that need to persist for it is very high.

Although using our optimized persist barrier provided a significant improvement, there is still a residual overhead of 30% over NP. Since our implementation of BSP requires logging, we wanted to understand how much of the residual overhead is due to logging. To this end, we also present execution time for an implementation of bulk persistency using the optimized persist barrier without logging (LB++NOLOG). We can see that it has an overhead of about 16% over NP. From this we can conclude that about half of the residual overhead of 30% is owing to logging.

8. RELATED WORK

There have been many proposals for enabling fast persistence with NVRAM [3, 4, 6, 7, 9]. All these techniques provide a programming framework to expose NVRAM to programmers. BPFS and NVHeaps [4, 9] rely on LB for ensuring correct order of persists, whereas others [3, 6, 7] rely on instructions like *clflush* and *mfence* provided by existing processors. It is important to note that these instructions are neither optimal nor sufficient to enforce correct order of persists. Newly proposed *clwb* instruction is optimal because it does not invalidate a cache line while writing it back to memory and another instruction *pcommit* is required to avoid reordering of persists at memory controller level. All of these techniques can seamlessly benefit from our efficient persist barrier implementation.

LOC [18] provides hardware logging support to reduce the overhead of persistence. It would be interesting to use LOC in conjunction with LB++ for enforcing BSP in bulk mode. Kiln [19] proposes a technique to reduce persist latency by using a non-volatile last level cache (NVLLC) along with NVRAM. Using a non-volatile cache also eliminates the requirement of logging by allowing NVLLC and NVRAM to store two versions of a cache line, and one of the versions can be conceptually considered as a log entry. NVM Duet [20], FIRM [21] and DP2 [22] propose optimizations in memory controller to improve the performance of persistent applications using NVRAM. All these proposals broadly help in reducing persist latency which is complimentary to our proposal of efficient persist barrier, in which we reduce conflicts and online persists.

Techniques like WSP [12] have been proposed, which save the entire execution state in NVRAM on a power failure. They rely on a small battery backup to flush caches and store processor state. Although this technique works in case of power failure, it is not clear as to how it can be used in case of other failures such as software crashes. In contrast, BSP guarantees persistence and recovery for any kind of failure. TSP [23], on the other hand, discusses tradeoffs in fault tolerance mechanisms, depending on the failure model (software crashes, power failures etc.). Pelley et al. [24] present designs for implementing ACID transactions for a system with NVRAM. Central to their design is the notion of persisting a batch of transactions together to amortize cost. However, their persists happen in the critical path; in contrast, we seek to move the persists out of the critical path using hardware support. In *memory persistency* [8] various models for persistency including epoch and strict persistency have been proposed. They also identify the possibility of inter-thread dependence tracking and enforcing strict persistency in bulk mode. However, they do not discuss how these can be realized.

Our work presents a detailed design and implementation of the same.

9. CONCLUSION

Many techniques and programming models have been proposed for achieving persistence using NVRAM. All of them require persists to be ordered to ensure consistency of persisted data, i.e. they will benefit from a persist barrier mechanism. We first illustrate the performance bottlenecks in the state-of-the-art persist barrier (LB). We then propose solutions to these bottlenecks and incorporate these in the design and implementation of an efficient persist barrier (LB++) for multicores with multi-banked caches. We evaluate LB++ by using it to enforce two recently proposed persistency models. We show that enforcing buffered epoch persistency (BEP) using LB++ (as opposed to LB) improves the performance by 22%. We show that enforcing buffered strict persistency (BSP) using LB++ (as opposed to LB) improves the performance by 20%, allowing us to support BSP at 1.3× execution time overhead.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by the Intel University Research Office and by EPSRC grants EP/M001202/1 and EP/M027317/1 to the University of Edinburgh.

11. REFERENCES

[1] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the 9th European Conference on Computer Systems*, ACM, 2014.

[2] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*. No. 319433-022, 2014.

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2011.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2011.

[5] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, ACM, 2014.

[7] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of VLDB Endowment*, vol. 8, no. 5, 2015.

[8] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st Annual International Symposium on Computer Architecture*, IEEE, 2014.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the 22nd Symposium on Operating Systems Principles*, ACM, 2009.

[10] D. R. Chakrabarti and H.-J. Boehm, "Durability semantics for lock-based multithreaded programs," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism*, USENIX, 2013.

[11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM, 2007.

[12] D. Narayanan and O. Hodson, "Whole-system persistence with non-volatile memories," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2012.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[14] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, IEEE, 2009.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ACM, 2008.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ACM, 1995.

[17] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multiprocessing," in *Proceedings of the 4th International Symposium on Workload Characterization*, IEEE, 2008.

[18] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proceedings of the 32nd International Conference on Computer Design*, IEEE, 2014.

[19] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual International Symposium on Microarchitecture*, ACM, 2013.

[20] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. 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*, ACM, 2014.

[21] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th Annual International Symposium on Microarchitecture*, IEEE Computer Society, 2014.

[22] L. Sun, Y. Lu, and J. Shu, "Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory," in *Proceedings of the 12th International Conference on Computing Frontiers*, ACM, 2015.

[23] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. M. III, "Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience," in *Proceedings of the 18th International Conference on Extending Database Technology*, 2015.

[24] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proceedings of VLDB Endowment*, vol. 7, no. 2, 2013.