Fine-Grain Checkpointing with In-Cache-Line Logging

Nachshon Cohen
Amazon
Haifa, Israel
nachshonc@gmail.com

David T. Aksun
EPFL
Lausanne, Switzerland
david.aksun@epfl.ch

Hillel Avni
Huawei
Tel Aviv, Israel
hillel.avni@huawei.com

James R. Larus
EPFL
Lausanne, Switzerland
james.larus@epfl.ch

Abstract

Non-Volatile Memory offers the possibility of implementing high-performance, durable data structures. However, achieving performance comparable to well-designed data structures in non-persistent (transient) memory is difficult, primarily because of the cost of ensuring the order in which memory writes reach NVM. Often, this requires flushing data to NVM and waiting a full memory round-trip time.

In this paper, we introduce two new techniques: *Fine-Grained Checkpointing*, which ensures a consistent, quickly recoverable data structure in NVM after a system failure, and *In-Cache-Line Logging*, an undo-logging technique that enables recovery of earlier state without requiring cache-line flushes in the normal case. We implemented these techniques in the Masstree data structure, making it persistent and demonstrating the ease of applying them to a highly optimized system and their low (5.9-15.4%) runtime overhead cost.

CCS Concepts • **Hardware → Non-volatile memory**;

Keywords non-volatile memory, NVM, durable data structures, in-cache-line logging, InCLL, fine-grain checkpointing

ACM Reference Format:
Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304046

Work done while the first author was a postdoc at EPFL.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6240-5/19/04.
https://doi.org/10.1145/3297858.3304046

1 Introduction

Non-Volatile Memory (NVM) is fast, byte-addressable memory that retains its contents after a power failure or a system crash. New technologies, such as 3D-XPoint [15], PCM [17, 25], STT-RAM [12], and ReRAM [1, 29], promise NVM at low cost, thus blurring the line between durable storage and main memory. One important use of NVM is enabling the rapid restart of a failed system. Restarting an existing machine typically incurs a significant delay due to the need to read data from durable media such as a disk or SSD, parse it, and rebuild internal data structures. NVM can avoid most of these restart costs. Since NVM is byte-addressable, it is possible to store efficient, pointer-based structures, such as B+ trees or hash maps, directly in NVM. After a failure, these structures remain in NVM, enabling the system to resume immediately after rebooting and recovering data [8].

The main challenge in designing durable data structures for NVM is that processor caches are (and are likely to remain) transient. During a power failure, all memory writes that were not propagated from cache to NVM will be lost. The processor memory system also complicates the task of writing cache lines to NVM in a consistent manner. Cache lines are not written back to memory (NVM) in the order in which an application modifies them, but rather according to a memory system's low-level and frequently undocumented cache replacement policy. This creates a well-studied challenge: how to ensure that the durable copy of a data structure is well-formed (consistent) after a crash, even though NVM contains a mixture of stale and new cache lines?

Most NVM systems require programmer-specified transactions to ensure that a group of memory writes either all reach NVM or none of them do. Typically, a change to a structure is first logged (using either a redo or undo log) in NVM and then applied to the structure. The log provides sufficient information for the recovery process to restore a structure to a consistent state, regardless of whether all structure modifications reached NVM.

However, the system must ensure that the log is completely flushed to NVM before the structure itself is modified.

This requires the use of cache flush instructions that transfer dirty cache lines from the (transient) cache to the (durable) NVM. These write backs are only guaranteed to have completed after a fence instruction executes. These instructions are expensive since they require a full round trip to NVM and reduce program performance by a significant amount. This overhead, moreover, can be incurred on an application's critical path, as a structure is being updated.

An alternative to transactions is checkpointing, another widely used technique for ensuring recoverability. At periodic intervals, an application's entire state is saved on durable media. After a failure, the last recorded checkpoint is restored to memory and the computation resumes from this point, requiring the re-execution of the work done between the checkpoint and failure. Since copying the entire state of an application to slow durable media is expensive, most systems take checkpoints at infrequent intervals (minutes to hours) to reduce this cost. The interval between checkpoints is a tradeoff between the overhead of recording checkpoints and the cost of re-executing the lost computation.

In this work, we introduce *Fine-Grained Checkpointing*, which uses frequent checkpoints to NVM to ensure persistence at low cost. Instead of ensuring that every memory write is logged or propagates to NVM, we partition an application's execution into epochs and ensure that after a crash, data structures can be restored to their state at the end of the most recently completed epoch (Nawab calls a similar approach "periodic persistence" [23]).[1] Our system flushes the processors' caches to NVM at the start of an epoch, thus ensuring that at this point NVM contains *all* modified data.

This approach has many advantages. The number of cache lines that must be flushed is bounded by the cache size, and modified cache lines may have been written back during the epoch, so the cost of recording a checkpoint is low. The modified lines are flushed in a batch by hardware, further reducing the cost. Our approach's low cost allows short checkpoint intervals, e.g., tens of milliseconds (we use $64ms$), thereby reducing both the potential data loss and the recovery time. In addition, a software developer need not annotate an application to delimit fine-grained transactions, rather he or she only needs to ensure that the application's state is recoverable at the end of an epoch.

Our approach differs from a traditional checkpoint in that there is no distinct copy of a data structure or a memory image. The in-NVM data structure also serves as the durable checkpoint. The challenge is to keep this checkpoint consistent as the structure is modified. After a crash, NVM state will consist of a mixture of the state at the beginning of the epoch, which must be kept, and modifications during the failed epoch, which must be discarded. The system must be able to distinguish between these two intermixed states and recover the one from the beginning of the epoch. One solution to this problem is to log the old memory value before each write. The log can be applied, in reversed application order, to roll back writes and to revert to the state at the beginning of the epoch. But logging itself requires care to ensure that the log reaches NVM before the structure is modified. Therefore, it again introduces the cost of cache-line flushes on the critical path.

To solve the problem of fine-grained modifications to NVM, we introduce the new concept of an *In-Cache-Line Log (InCLL)*. An InCLL serves the same role as an undo log. But instead of using an external log, the InCLL is placed *in the same cache line* as the data structure field being logged. InCLL relies on the Persistent Cache Store Order (PCSO) memory-ordering model of NVM (two writes to the same cache line reach NVM in program order) to ensure the ordering requirements of the log, without introducing cache flushes and delays.

The main limitation of an InCLL is its limited capacity. Since it resides in the same cache line as the data, an InCLL should be small and cannot handle all modifications. If an object is modified multiple times during an epoch, InCLLs may be insufficient to provide crash recoverability. In this case, our approach falls back on object-level logging. After the entire object is logged, subsequent modifications in the epoch do not require additional actions. Together, the combination of InCLL and object-level logging drastically reduces the number of synchronous writes to NVM.

To validate this approach, we applied Fine-Grained Checkpointing into Masstree [21], a cache-efficient data structure that combines a B+ tree and Trie. We also implemented a durable memory allocator based on this approach. Measurements show that the overhead of our scheme is low and restart time is dramatically reduced.

The main contributions of this paper are:

- Fine-Grained Checkpointing, a technique to ensure a consistent, quickly recoverable data structure in NVM after a system failure.
- In-Cache-Line Logging, a undo-logging technique that enables recovery of the state from the beginning of an epoch without requiring cache-line flushes in the normal case.
- Implementation of these techniques for the Masstree data structure, which made it persistent and demonstrated their application in a highly optimized system and their low (5.9-15.4%) runtime overhead cost.

2 Background

2.1 Persistent Memory Ordering Model

In this paper, we use the Persistent Cache Store Order (PCSO) memory ordering model for NVM [9]. Cache lines are written back to NVM according to a computer's (unspecified)

[1]Unlike most checkpointing, which resumes execution at the point at which a checkpoint occurred, our goal is to restore the persistent data structures to their state at the checkpoint, so that a program can *restart* its execution.

cache replacement policy, so we cannot assume any specific write-back behavior. An application may explicitly force specific cache lines to be written to NVM, by using cache-line write-back instruction, such as the x64's `clflushopt` or `clwb`. These instructions are asynchronous, they only initiate a memory transfer but do not wait until data actually reaches NVM. To ensure that a write-back completes, the application must issue a fence instruction, such as `sfence`, which delays CPU execution until the outstanding write-back instructions finish. Since this instruction waits until the data reaches NVM, it is far more expensive than a normal (cached) memory reference.

While ensuring the order in which writes to different cache lines reach NVM is expensive, ordering writes to the same cache line is essentially free. If two writes target the same cache line, the order in which they reach *the cache* corresponds to the order in which they reach NVM. Preserving the order of cache writes can be done with release memory ordering in C++11, which introduces a *happens-before* relation between the writes [4, 20]. On the x64 architecture, the release memory fence incurs *no* runtime overhead and only limits the ability of a compiler to reorder writes.

Formally, given two writes X and W, we say that $X <_p W$ if X is written to persistent memory no later than W. $X <_{hb} W$ is the standard *happens before* relationship. $c(X)$ represents the cache line address X writes to. The following holds [9]:

- $W <_{hb} \mathit{writeback}(c(W)) <_{hb} \mathit{fence} <_{hb} X \Rightarrow W <_p X$ (explicit flush).
- $W <_{hb} X \wedge c(W) = c(X) \Rightarrow W <_p X$ (granularity).

Our InCLL technique relies on the second ordering guarantee, that if two writes target the same cache line, a happens-before relation is sufficient to ensure persistence ordering.

2.2 Masstree Data Structure

Masstree [21] is a production-quality ordered-set data structure that has been used to build in-memory databases such as Silo [27]. Masstree is a combination of a Trie and a B+ tree, implemented to carefully exploit caching, prefetching, optimistic navigation, and fine-grained locking. Below we sketch some details of Masstree that are necessary to understand our changes that make Masstree durable.

Masstree uses two types of nodes: internal nodes and leaf nodes. There are roughly an order of magnitude more leaf nodes then internal nodes and leaf nodes are modified much more frequently, so our checkpointing focuses on the leaf nodes. The number of items in an leaf node is a parameter of Masstree's implementation; the default implementation uses 15 keys and 15 pointers to values. The keys reside in the `keys` array and the value pointers resides in the `vals` array. Listing 1 illustrates some details of a leaf node.

The `permutation` field records valid key-value pairs, in other words which entries in the arrays are occupied. We can consider the `permutation` field to be a bitmap specifying whether an index is in use, although, in practice, it also orders entries as well. Deleting a key-value pair from a node modifies only the `permutation` field. Inserting a new key-value pair modifies both the `permutation` field and an unused entry pair in the `keys/vals` arrays. Updating an existing key modifies only the `vals` entry.

Listing 1. Masstree's node structure

```
class basenode; // lock, version, meta information
template <int width=15>
class leafnode : public basenode{
    basenode *parent, *prev, *next;
    uint64_t permutation; // which key/vals are active
    keytype keys[width];
    valuetype *vals[width];
    void remove(keytype key){
        int idx = find_idx(key);
        remove_idx(&permutation, idx);
    }
    void insert(keytype key, valuetype *val){
        int idx = insert_idx(&permutation);
        keys[idx] = key;
        vals[idx] = val;
    }
    void update(int idx, valuetype *val){
        vals[idx] = val;
    }
};
```

Masstree support splitting and merging of nodes, but these happen far less frequently than modification of leaf nodes. The full details of the Masstree algorithm are quite involved [21], but are not necessary to understand this paper.

3 Overview

The main contributions of this paper are Fine-Grained Checkpointing and In-Cache-Line Logging, which we illustrate by show how to make Masstree durable.[2] The approach uses a combination of three techniques: fine-grained checkpoints, in-cache-line log, and external logging. This section sketches these three mechanisms, and § 4 provides more detail.

Fine-Grained Checkpointing Execution is broken into *epochs* — our implementation uses 64*ms*, the Masstree memory reclamation epoch, though longer or shorter intervals are feasible. During an epoch, NVM contains a mixture of the memory state from the previous epoch and some — but not all — memory writes executed during the current epoch. At the start of an epoch, the entire cache is flushed to NVM, ensuring that all modifications from the previous epoch are safely stored in NVM. The cost of flushing the cache is low

[2]Our code is online: https://github.com/epfl-vlsc/Incll

as its size is bounded and some modifications may have been written back to NVM during the previous epoch (§ 6.2).

External Logging The external log is a standard undo log. Under certain circumstances, when a Masstree node is modified during an epoch, the entire node is stored in the log so that subsequent modifications can be reversed. To ensure persistence ordering, the log is written to NVM and an `sfence` operation is issued before the node is modified.

An external log is the standard tool for ensuring the consistency of durable data structures in NVM [5, 9, 10, 13, 14, 16, 19, 28]. It is possible to log at different granularities: a word, an object, or a page. We choose object-level granularity, so that when a single word in a Masstree node is modified, the entire node is recorded in the external log. External logging's primary benefit is simplicity. It ensures durability without requiring pervasive changes to the Masstree algorithm. However, it may also have performance benefits since a node is only logged once, even if it is modified many times, as during merges or splits.

In our approach, we always use the external log for infrequent, these complex modifications. In addition, changes to internal (non-leaf) nodes are infrequent, so they are also handled by the external log (§ 6.1). Leaf nodes are logged only when required by the InCLL algorithm described below.

The external log is discarded after the cache is flushed at the start of an epoch since all of the logged changes will have been stored in NVM.

InCLL In-Cache-Line Logging (InCLL) is a technique for logging modifications to a node without waiting on NVM. InCLL embeds an undo log inside the same cache lines as a Masstree leaf node. When a node is modified for the first time in an epoch, InCLL stores the old value of the modified field. Since the log resides in the same cache line as the data, no write backs or fences are required to ensure that the log reaches NVM with the modification.

The primary benefit of InCLL is the low cost of logging. But, the capacity of each node's log is limited since an InCLL resides in the same cache line as the data. Each log entry also reduces the number of Masstree array entries, which degrades the cache efficiency of the Masstree structure. Our durable Masstree algorithm logs — in a typical case — only one or two modifications per node per epoch. If a leaf node is modified repeated, external logging is likely to be used.

We find that the combination of a limited InCLL and an external log works extremely well. If Masstree updates during an epoch are random, most will access different nodes, and the external log will be infrequently used. If, on the other hand, modifications are ordered, there may be many writes to the same node, in which case the external log, which only records the node once, will perform well.

4 Durable Masstree

In our scheme, execution is partitioned into $64ms$ epochs. We use the `wbinvd` instruction to flush the entire cache at the start of an epoch. Since Masstree uses epoch-based reclamation for allocating and de-allocating nodes, we reuse its mechanism and interval for our epochs as well. Shorter intervals would raise the overhead cost of cache flushing (currently about 2% (§ 6.2)) but reduce the number of updates that might be lost or need to be re-executed after a failure.

Epochs are assigned a monotonically increasing index, which is stored durably. With $64ms$ epochs, a 32-bit index wraps after more than eight years.[3] We also keep track of failed epochs. During recovery, an epoch in which a crash occurred is added to the set of *failed epoch*. Modification made during a failed epoch will not be visible after recovery.

4.1 InCLL Algorithm

In our durable Masstree algorithm, each node consists of 14 keys, 14 pointers to values, and two InCLLs.[4] Each InCLL requires 8 bytes, like a pointer. The InCLL entries are carefully cache aligned. The first resides immediately before the 14-pointer array and the second resides immediately after the 14-pointer array. Hence, the first InCLL resides in the same cache line as pointers 0–6 and the second InCLL resides in the same cache line as pointers 7–14 (Figure 1). Each of these InCLL can record a single value modification per epoch.

We use an additional InCLL for the `permutation` field. The `permutation` field records whether a location in the pointer array is occupied. This field is modified when a new key-value pair is inserted or removed.

The InCLL used to log the `permutation` field is denoted InCLL$_p$, and the two InCLLs used to log values are denoted InCLL$_1$ and InCLL$_2$ for the first and second cache lines, respectively. The operation of InCLL$_p$ differs from InCLL$_{1,2}$, as described below.

4.1.1 InCLL Structure

To understand the operation of the InCLL$_p$ log, we describe how insert and delete operate. When a new key-value pair is inserted into a leaf node, an unoccupied location is found using the `permutation` field. The appropriate entry in the `keys` array is set to the key and the corresponding entry in the `vals` array is set to the value. Furthermore, the `permutation` field is updated to record that the entry is now occupied. If there is no free entry, the node must be split, a case that is handled by external logging.

After a crash, a leaf node must be returned to the same state as the start of the current epoch. There are a number of cases to consider. If an entry was unoccupied at the beginning

[3]If the data lasts longer, a background thread could run once every 8 years and reset all indices to zero. Since the duration of graduate studies is less than 8 years, this feature is not in our current implementation.

[4]One fewer key and pointer than the standard implementation.

of the current epoch, there is no need to restore its `keys` or `vals` fields. Hence, the only field that must be logged is the Masstree `permutation` field (in `permutationInCLL`). Even if multiple key-value pairs are inserted during an epoch, only the `permutation` field needs to be logged. Since this field is logged in InCLL$_p$, there is no need to use the external log for multiple consecutive writes.

Deletion in Masstree is similar to insertion. Only the `permutation` field is modified to indicate that the entry for the key is now unoccupied, so no external logging is necessary in this case either. Moreover, if a node is modified by inserting new key-value pairs and subsequently removing key-value pairs, then InCLL$_p$ logging still suffices since restoring `permutation` leaves the node in its original state.

But it is not possible to simply log the `permutation` field in a mixed sequence of insertions and deletions. An entry that is deleted might be overwritten by a subsequent insertion, which destroys the original key-value pair that should be restored after a crash. Thus, if a key-value pair is removed and, in the same epoch, a key-value pair is inserted into the same entry, then the entire node must be externally logged. InCLL$_p$ contains a boolean indicator (`insAllowed`) that insertions do not require external logging. The indicator is initially `true` and is set to `false` during a delete.

Since a data structure might be large, it is impractical to clear all InCLL entries when an epoch advances. The InCLL$_p$ also records the epoch number in which the InCLL was used. During recovery, the log is applied (i.e., the `permutation` field is restored to its old value) only if the epoch number corresponds to a failed epoch.

The last field of InCLL$_p$ is a boolean (`logged`) indicating that a node was logged to the external log. If the node was logged in the current epoch, no further logging is required.

Overall, InCLL$_p$ consists of four fields:

- `nodeEpoch` stores the epoch number for the InCLL$_p$.
- `permutationInCLL` stores the value of the `permutation` field at the beginning of the `nodeEpoch` epoch. If the system crashes during this epoch, `permutation` is recovered from `permutationInCLL`.
- `insAllowed` controls if insertions are permitted to use the InCLL.
- `logged` records if the node was logged in the external log during epoch `nodeEpoch`.

Code depicting the structure of a leaf node appears in Listing 2. It is also illustrated in Figure 1.

4.1.2 InCLL Inserts and Deletes

Next we consider inserting or deleting a key-value pair to a leaf node and show how InCLL$_p$ is used (Listing 3). When a key-value pair is inserted or removed from a leaf node, the thread first checks whether `nodeEpoch` is equal to the current epoch (`curEpoch`). If not equal, the current modification is the first time the node is modified in the current epoch.

Listing 2. Durable Masstree's node structure

```
class basenode; // lock, version, meta information
struct ValInCLL{
    long idx:4;
    static const INVALIDIDX=−1;
    long ptr:44; // 48 bits minus 4 least significant bits
    long lowNodeEpoch:16; // last 16 bits of the epoch;
    ValInCLL(ptr, idx);
    ValInCLL():ptr(nullptr),idx(INVALIDIDX);
};
template <const int width=14>
class leafnode : public basenode{
    basenode *parent, *prev, *next;
    uint62_t nodeEpoch; // InCLL_p field
    bool logged, InsAllowed; // InCLL_p fields
    uint64_t permutationInCLL; // InCLL_p field
    uint64_t permutation; // which key/val are active
    keytype keys[width];
    alignas(64) struct {} ALIGN; // align to cache line
    ValInCLL InCLL1; // same cache line as vals[0..6]
    valuetype *vals[width];
    ValInCLL InCLL2; // same cache line as vals[7..13]
};
```

Then, the old (pre-modified) `permutation` value is saved in `permutationInCLL`, effectively logging its old value. Afterward, `nodeEpoch` is set to the current epoch, `insAllowed` is set to `true`, and `logged` to `false`.

The persistence ordering of writing `permutationInCLL` and `nodeEpoch` is important. If setting `nodeEpoch` reaches NVM before `permutationInCLL`, recovery might fail. The problem is that if a failure occurs after the new epoch reaches NVM but before `permutationInCLL` reaches NVM, the recovery procedure (discussed below) will assume that the node was modified in the most recent epoch and must be recovered. Thus, it would incorrectly recover the node using the very old value of `permutationInCLL` (belonging to a previous epoch).

We use this persistence ordering: Set `permutationInCLL` to the old (pre-modified) `permutation` value. Second, set `nodeEpoch` to the current epoch. Third, modify `permutation` to reflect the insertion or deletion of a key-value pair. The fields `insAllowed` and `logged` are semantically transient and do not require persistence ordering. This ordering ensures that the node can always be recovered. If only the first modification reaches NVM, the node is not recovered as `nodeEpoch` does not record a failed epoch. If the first and second modifications reach NVM, the node is recovered; in this case, both `permutation` and `permutationInCLL` represents the value at the beginning of the failed epoch. Recovery is not required, but also not harmful. If all three modifications reach NVM, the node is recovered correctly using `permutationInCLL`. These three fields reside in the same

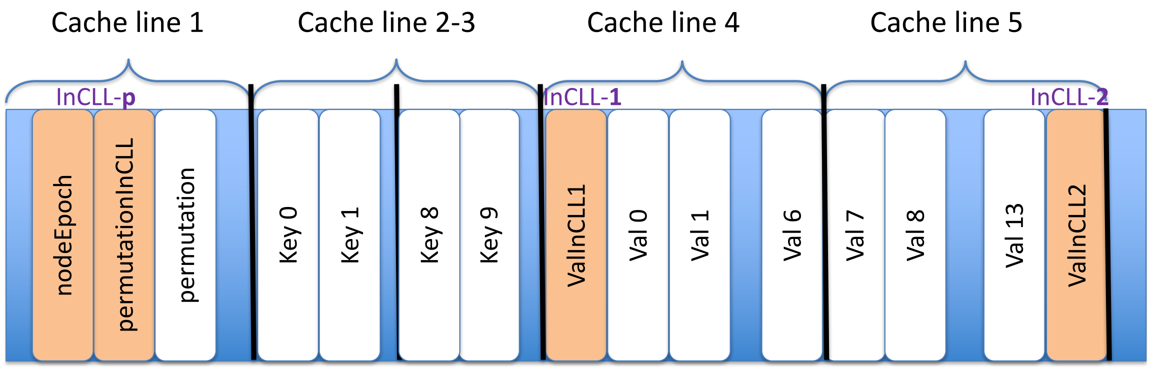


Figure 1. Durable Masstree leaf node. InCLL are orange. `nodeEpoch` and `permutationInCLL` resides in the same cache line as `permutation`. Values have two InCLL entries, $\text{InCLL}_{1,2}$, located in their cache lines.

cache line, so ordering is ensured by a release memory fence, but not writing back or waiting for a full round-trip to NVM.

If `nodeEpoch` is equal to `curEpoch`, the node was already modified during the current epoch. As mentioned above, consecutive insertions or removals can use the InCLL. The only case in which further action is need is when `logged` is `false` (that is, the node was not logged in the external log), the current operation is an insertion of a new key-value pair, and `insAllowed` is `false`. In this case, the node is logged in the external log before it is modified. Code for insertions and deletion appear in Listing 3.

4.1.3 InCLL Update

Updating the value of a key already in a leaf is slightly more complex than insertion or deletion. The two InCLLs embedded in the value array of a node, denoted $\text{InCLL}_{1,2}$, are used to log the old value when it is updated. These InCLLs, unlike InCLL_p, require careful encoding to reduce their size. $\text{InCLL}_{1,2}$ can be used to log one of seven possible fields, so it contains an additional field that records the entry that was modified. Using two 64-bit words for $\text{InCLL}_{1,2}$ would reduce the number of values in the array to 12 in two cache lines, incurring a performance penalty. Therefore, it is desirable to use a single word for $\text{InCLL}_{1,2}$.

To compact $\text{InCLL}_{1,2}$, we observe that the values stored by Masstree are pointers to the actual values. In current x64 architecture, pointers are represented in a canonical form in which only the lower 48 bits are used.[5] In a valid memory address, the upper 16 bits must equal the value of the 47th bit. In addition, all memory allocations are aligned on 16-byte boundaries, so the least significant four bits must be zero. We pack $\text{InCLL}_{1,2}$ as follows. Bits 0–3 represent the index of the pointer that is logged; this field can represent seven values, so with 4 bits we can indicate all array entries (0–6 for InCLL_1 and 7–13 for InCLL_2). Bits 4–47 hold the logged pointer. Bits 48–63 represent the lower 16 bits of the epoch in which the node was modified, denoted `lowNodeEpoch`. We assume that the lower 16 bits of the current epoch can be combined with the higher 16 bits of InCLL_p's `nodeEpoch` to produce the full epoch number in which the InCLL was used. During updates, we check if the difference between the current epoch and the InCLL_p's `nodeEpoch`. If 16 bits are insufficient to correctly encode the epoch, we fall back on the external log. This happens approximately once an hour (2^{16} epochs of $64ms$ each).

When the value of an existing key is updated, the thread first checks if InCLL_p's `nodeEpoch` is equal to `curEpoch`. If it is not, this is the first time the node is modified in the current epoch. The thread computes which InCLL must be used, depending on whether the modified entry's index is 6 or lower. The old value, the index, and the lower 16 bits of the epoch are encoded into a single word and stored in the appropriate InCLL.

If InCLL_p's `nodeEpoch` is equal to `curEpoch`, the node has been modified during the current epoch. But if the InCLL of the other cache line was used, it is still possible to use the unused InCLL. In addition, if the pointer being modified was previously logged in the InCLL, there is no need to record it again. The latter is valuable when the keys are drawn from a skewed distribution. So, if some keys are popular and modified multiple times during an epoch, there is no need to use the external log. The external log is likely to be necessary if *two* (or more) popular keys reside in the same cache line of a leaf node. Code for updates appear in Listing 3.

4.2 External Logging

We use the external log for modifications that are more complex or less common than these leaf updates. This has the benefit of requiring minimal changes to the Masstree code, while maintaining a relatively low overhead cost. Splitting and merging of leaf nodes are infrequent and are handled by logging the affected nodes. Also, all modifications to internal tree nodes are logged.

[5]With future 5-level paging, we can fallback to external logging if the stored address is higher than 2^{48} or exploit stricter alignment restrictions.

Listing 3. Durable Masstree operations

```
void leafnode::InCLL(bool InCLLallowed, permInCLL,
                         valInCLL[2]){
    if(globalEpoch != nodeEpoch){
        isInsertionsAllowed = true; isLogged = false;
        if(higher(globalEpoch) != higher(nodeEpoch))
            isLogged = logNode();
        if(!isLogged){
            permutationInCLL = permInCLL;
            InCLL1 = valInCLL[1];
            InCLL2 = valInCLL[2];
            // order writes
            atomic_thread_fence(memory_order_release);
        }
        nodeEpoch = globalEpoch;
        InCLL[1,2].lowNodeEpoch = lower(nodeEpoch);
    }
    else if(!isLogged && !InCLLallowed)
        isLogged = logNode();
    atomic_thread_fence(memory_order_release);
}
void leafnode::remove(keytype key){
    int idx = find_idx(key);
    InCLL(true, permutation, ValInCLL(), ValInCLL());
    InsAllowed=false;
    remove_idx(&permutation,idx);
}
void leafnode::insert(typetype key, valuetype *val){
    int idx = insert_idx(&permutation);
    InCLL(InsAllowed, permutation, ValInCLL(), ValInCLL());
    keys[idx] = key;
    vals[idx] = val;
}
void leafnode::update(int idx, valuetype *val){
    ValInCLL InCLL = (idx<=6) ? InCLL1 : InCLL2;
    InCLLallowed = (InCLL.idx == idx
                     || InCLL.idx == INVALIDIDX);
    ValInCLL vc1 = ValInCLL (vals[idx], idx);
    ValInCLL vc2 = ValInCLL(nullptr, INVALIDIDX);
    if(idx>=7) swap(vc1, vc2);
    InCLL(InCLLallowed, permutation, vc1, vc2);
    vals[idx] = val;
}
```

The external log is also used whenever a modification cannot be handled by the InCLL. If two values in the same cache line are modified in the same epoch, the external log is used. Similarly, if a key-value pair is removed and inserted at the same epoch, we also fall back on the external log.

To reduce the cost of checking if an internal node was logged, we introduce an epoch number in each internal node that indicates that the node was log in a specific epoch. A simple comparison against the current epoch number prevents multiple logging. Our algorithm locks a node before logging to avoid races. This provides an additional benefit during recovery; since a node appears at most once in the external log, there are no dependencies among log entries and they can be restored in parallel. In contrast, standard undo logging has to be applied in a reversed application order, limiting concurrency in the recovery procedure.

4.3 Recovery

After an abrupt crash, the durable Masstree is recovered as follows. First, the external log is applied before execution resumes. This is done by copying the contents of each node from the log to its corresponding node. As mentioned above, there are no dependencies among log entries, so it can be applied concurrently with minimal or no synchronization. Pseudo-code illustrating recovery appear in Listing 4.

InCLLs must also be applied to recover nodes. But unlike the external log, the InCLLs are embedded inside the durable Masstree nodes. Applying all of them before the execution resumes would require a traversal of the entire tree, which would cause a long delay. To avoid this, the InCLL restores are applied lazily, during tree traversals.

When a thread attempts to access a node, it first checks if the node's `nodeEpoch` is less than the epoch number of the current execution. If it is, recovery is applied to the node before continuing with the access. To avoid concurrency races when multiple threads attempt to recover a node simultaneously, we use locking. However, it is not possible to use the leaf's lock. The problem is that the state of the lock is not preserved, so after a failure, it might be in a failed state. Therefore, attempting to lock the node could result in a deadlock, even if only a single thread is attempting to lock the node. Instead, the system defines an array of (transient) locks for recovery. When a thread attempts to recover a node, it hashes the leaf address to find an appropriate recovery lock. After acquiring the lock, the thread checks if the node's epoch is still lower than the first epoch in the current execution. If it is, the thread attempts to recover the node from the InCLL. First, it checks if `nodeEpoch` is a failed epoch. If so, the `permutation` field is recovered by copying the `permutationInCLL` field into it. Second, the thread reconstructs the epoch of $\text{InCLL}_{1,2}$ by combining the lower 16 bits with the higher bits from `nodeEpoch`. If the resulting epoch number is a failed epoch, the index and value pointer are retrieved from the $\text{InCLL}_{1,2}$ field and are applied to the appropriate location in the `vals` array. Lastly, the node's InCLL is initialized to the first epoch in the current execution to indicate that the node does not need further recovery. Code illustrating InCLL lazy recovery appears in Listing 4.

At the point when a leaf node is externally logged, it may have been modified and the changes logged in its InCLL. Thus, the contents of the external log will not equal to the state of the node at the beginning of the failed epoch, and simply copying the log to the node is insufficient for correct recovery. After recovery using the external log, the InCLLs in the nodes must also be applied. To reduce recovery time,

Listing 4. Durable Masstree recovery

```
uint64_t currExecEpoch; // first epoch in current execution
lock recoveryLocks[K];
// before first access to durable Masstree
void durableMasstree::recovery(){
# parallel for
    for each node L in external log do:
        memcpy(L−>addr, L−>content, L−>size);
}
// before first access to a leaf node
void leafnode::lazyNodeRecovery(){
    if(unlikely(nodeEpoch<currExecEpoch)){
        int idx = hash(this);
        recoveryLocks[idx].acquire();
        if(nodeEpoch<currExecEpoch){
            nodeRecovery();
        }
        recoveryLocks[idx].release();
    }
}
void leafnode::nodeRecovery(){
    // InCLLp
    if(failedEpoch.find(nodeEpoch))
        permutation = permutationInCLL;
    nodeEpoch = currExecEpoch;
    // InCLL1
    uint64_t epoch = higher(nodeEpoch) | InCLL1.epoch;
    if(failedEpoch.find(epoch))
        vals[InCLL1.idx] = InCLL.ptr;
    // InCLL2
    epoch = higher(nodeEpoch) | InCLL2.epoch;
    if(failedEpoch.find(epoch))
        vals[InCLL2.idx] = InCLL.ptr;
    InCLL[1,2] = ValInCLL(nullptr, INVALIDIDX);
    InCLL[1,2].epoch = lower(currExecEpoch);
    basenode::initlock(); // might be in bad state after crash
}
```

these restores are done lazily, on the first access to a node, similarly to non-externally logged nodes.

There is no need to flush cache lines during recovery. If the system crashes before recovery is complete, it can be applied again.

5 Durable Memory Allocation

The Masstree data structure does not store actual values inside the tree. Rather, Masstree holds a pointer to value buffers. Clearly, these data buffer must be allocated in NVM so their contents remain after a crash. Allocating and deallocating these buffers are not strictly part of the Masstree algorithm. Still, allocating a data buffer is typically required for every update operation on Masstree, so it is highly desirable to reduce the cost of such allocation by avoiding costly write backs and fences. As a second demonstration of Fine-Grained Checkpointing and In-Cache-Line Logging, we describe an allocation algorithm that uses them to avoid write backs and flushes during the critical path of allocation.

Our main observation is that an allocator is essentially a data structure (a set) that records free chunks of memory. Therefore, we can again use checkpointing for this structure, recovering the state of the allocator to the beginning of a failed epoch. The InCLL technique can reduce the overhead of logging writes to the allocator's data structures.

Our durable allocator uses a linked list of free objects, with a different list for each size class. When an object is allocated, it is popped from the appropriate list of free objects and inserted to Masstree. When an object is deallocated, it is added to the appropriate list of free objects and can be reused later (Masstree use epoch-based reclamation, so the memory becomes available only in the next epoch).

To implement the list of free objects, it is sufficient to use a single next pointer per node. Thus, to implement a durable allocator, we protect this free pointer with an InCLL, denoted InCLL$_n$. The basic persistent allocator is simple, each object has a header of three words that fit into a cache line: the next pointer, the InCLL copy of the next pointer, and the epoch number. This design ensures that allocation never requires writing back to NVM or memory fences.

Our allocator is based on Epoch-Based Reclamation (EBR), which allows a node to be allocated only if it was free at the start of an epoch (otherwise, concurrent threads might access the new data improperly). This property implies that there is no need to log the actual content of a buffer and no write backs are necessary. On recovery, the buffer is returned to its state at the beginning of a failed epoch, when the buffer is free and its contents are irrelevant.

Our checkpointing technique is much simpler to use than other NVM systems' techniques for durable allocation. In these systems, a programmer has to specify explicitly that the buffer's contents must be written back to NVM before the structure operation is invoked. This is not the case for our system. The programmer can simply write data into a buffer and insert it into the durable Masstree. At the start of the epoch, the contents of the buffer, together with the durable Masstree, will be written back to NVM when the entire cache is flushed.

The drawback of our approach is that the allocator requires 24 bytes in the header of each object. Due to the 16 bytes alignment constraint, this results in a 32-byte header. But, we can reduce this overhead to only 16 bytes.

5.1 Compacting InCLL for Memory Allocation

To ensure that the *free list* (linked list of free nodes) return to the same state as the beginning of the epoch, each object needs a header with three fields: the current `next` pointer; the value of the `next` pointer at the beginning of the epoch, denoted `nextInCLL`, used for logging; and a 32-bit epoch number. Since both `next` and `nextInCLL` are pointers in canonical form, the upper 16 bits of each can be computed from their 47th bit. Thus, the 32-bit epoch can be broken into two parts and encoded into *both* next pointers, requiring only 16 bytes for all three fields.

However, this idea is insufficient by itself. The problem is that a crash may happen after half of the epoch number reaches NVM but before the other half, leading to an incorrect recovery. Our solution is to encode a small counter at the two least significant bits of both `next` and `nextInCLL`. This counter is incremented when the pointers are written in a new epoch and allows the recovery procedure to distinguish whether both pointers were fully written. If both pointers have the same counter value, the correct epoch number can be reconstructed by combining the 16 most significant bits of `next` with the 16 most significant bits of `nextInCLL`. In this case, recovery proceeds similar to the `permutation` case (Listing 4). If, on the other hand, the pointers have different counter values, we know that a crash happen while the pointers were being modified, and the `next` pointer must be recovered from the `nextInCLL` logged pointer.

5.2 Correctness

We tested the modified system by intentionally crashing it at random points, launching a new process, and checking that system's state matched the state at the beginning of the failed epoch. We also used many unit tests to ensure that a cache line was left in its correct state. We are currently developing a tool to help reason about the correctness of this type of system.

6 Performance

We implemented fine-grained checkpointing and InCLL as described above (INCLL) and measured it against the unmodified, transient Masstree (MT) and against an improved version of Masstree (MT+) that adopted two enhancements from INCLL, using a global barrier at each epoch and `mmaping` memory space for Masstree's pool allocator, rather than obtaining it through `jemalloc`. Without these two enhancements, INCLL performed slightly better than MT (Figure 2).

All experiments were run on a server containing two Intel Xeon Gold 6132 (Skylake) processors, each with 14 cores and 28 hyperthreads, running at 2.6 GHz, with a 19.25 MB L3 cache. The system contained 1.5 TB of DDR4-2666 RAM. The operating system was Ubuntu Linux 16.04.5 LTS. The code was compiled by g++ version 5.4.0. with the baseline makefile optimization level -O3. Each experiment was executed 10 times, and we report the average time. The standard deviation of the experiments ranged from 0.03% to 0.08%.

Since NVM is not available, we allocate a file in `/dev/shm` and mapped it to the application address space (DRAM). By default, we do not introduce artificial latencies for the cache flush or fence operations. However, we also measure the effect of higher NVM latencies by introducing artificial latency

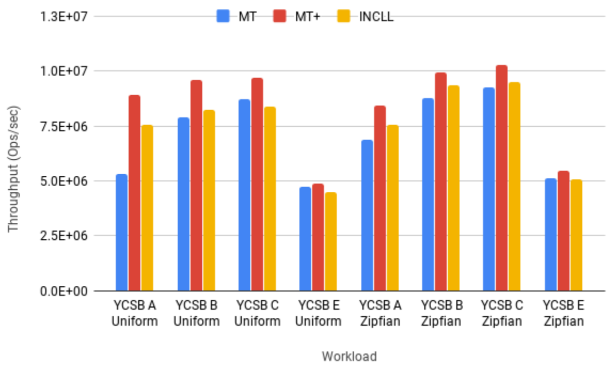


Figure 2. Throughput of baseline Masstree (MT), optimized Masstree (MT+), and our durable Masstree (INCC).

after the `sfence` instructions since `clflushopt` instructions are asynchronous.

Figure 2 reports the throughput of the three Masstree versions on different workloads. Unless otherwise noted, the tree was initialized with 20 million entries and we ran with 8 threads. Keys and values are 8-bytes long[6]. The workload was generated by driver threads on the same machine to avoid network interference. For YCSB_A (write heavy), the operation distribution was 50% puts and 50% reads, for YCSB_B (read heavy) 5% puts and 95% reads, for YCSB_C (read only) 100% reads, and for YCSB_E a read-only scan of 10 keys. We employ two key distributions. In *uniform*, the keys are generated uniformly at random in the range between zero and 20M. In *zipfian*, the keys are generated according to a zipfian distribution with a skew parameter of 0.99. Keys are scrambled by computing a hash of their values, so that frequent keys do not (necessarily) appear in close proximity. We report the overall throughput (operations per second) of executing 1 million operations on each thread.

The optimized version (MT+) performed 2.4–68.5% better than unmodified Masstree (MT). We use MT+ as the baseline for comparisons with the durable version (INCLL). Durable Masstree performed 5.9–15.4% slower than MT+, which reflects the cost of InCLL and periodic cache flushes. As expected, the write-intensive workload's (YCSB_A) performance is reduced by a larger amount (10.3–15.4%) than the read-light (5.9–13.9%) or read-only workloads (7.9–13.5%). The scan workload (YCSB_E) is least affected by InCLL. The Zipfian workload performed better than the uniform workload in both systems, and it is less affected by InCLL (5.9–10.3% vs. 7.8–15.4%) because its skewed distribution means fewer nodes are accessed and consequently processor caching is more effective and more writes are logged (see discussion of Figure 6).

[6]Values are allocated in a 32-byte buffer containing additional Masstree fields.

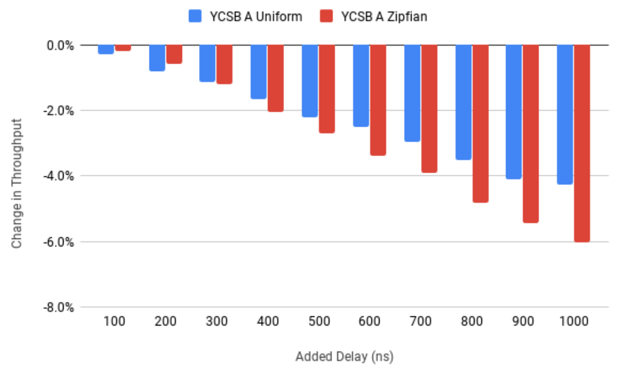


Figure 3. Effect of emulated latencies for cache write back on INCLL (baseline is INCLL with zero emulated latency).

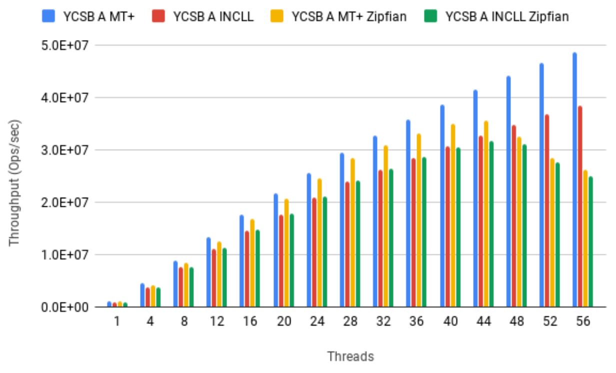


Figure 4. Throughput of INCLL and MT+ (INCL_A benchmark) for different numbers of threads.

INCLL increased the number of instructions executed 0.0–14.5% (uniform) and 0.1–7.6% (Zipfian). It also increased the number of L1 load and store references by a similar amount, but had less effect on the L1 cache miss rate. The number of LLC load references also increased by 7.9–16.3%, but the number of L3 store references was less consistent (-28.7–27.5%). For MT+, the LLC cache miss rate was high (25.8–39.9% load, 25.1–99.6% store), but the absolute number of misses is low (1.3–2.9M load, 31-666.8K store). INCLL had little effect on the L1 load miss rate (-5.1–14.2%), but it reduced the LLC load miss rate by 42.0–95.2%.

Figure 3 shows the effect of increasing the latency of flushing modified locations to NVM on the YCSB_A write-heavy workload. We introduced an additional delay of 100*ns*–1000*ns* after the `sfence` operations. The effect on NVM latency is small. Even with an added latency of 1µs, the performance of INCLL only decreased by 4.3% for the uniform workload and 6.0% for the Zipfian, compared to no emulated latencies. This small difference demonstrates that InCLL is able to avoid the full cost of flushing writing to NVM for most memory references.

Figure 4 depicts the performance of MT+ and INCLL over 1 to 56 threads. The workload is again YCSB_A. The performance loss due to InCLL seemed unrelated to the number of threads, ranging from 14.6–21.3% for uniform and 3.0–19.3% for Zipfian. For the Zipfian workload, overall benchmark performance decreased at 44 threads in both system. In a larger tree (100M entries), however, the performance of all benchmarks increased monotonically with the number of threads and the performance loss due to InCLL was similar (10.7–15.3% and 6.7–22.2%, respectively).

Figure 5 depicts the performance of MT+ and INCLL for increasing the number of entries in the tree. The workload is again YCSB_A. Both MT+ and INCLL were affected similarly by the increased tree size. The performance on the uniform workload decreased 69% for both MT+ and InCLL as the tree grew from 10K to 100M nodes, and the Zipfian workload performance decreased by approximate 50%.

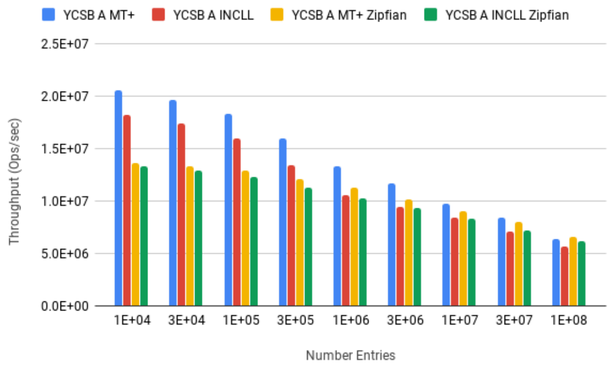


Figure 5. Throughput of INCLL and MT+ (INCL_A benchmark) for varying tree size.

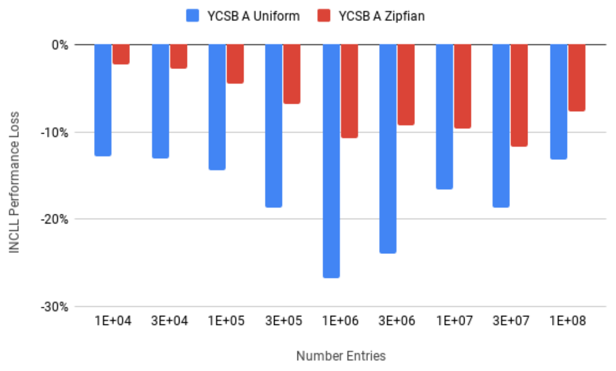


Figure 6. Overhead of INCLL over MT+ (INCL_A benchmark) for varying tree size.

The overhead for the uniform workload forms a parabola (Figure 6), with a lower overheads for small and large trees. To understand this phenomena, we measured the effectiveness of the external log and InCLL. Figure 7 show the number of nodes logged for the YCSB_A workload when InCLL logging is disabled (LOGGING) and in the normal operating mode (INCLL). As can be seen, for both uniform and zipfian

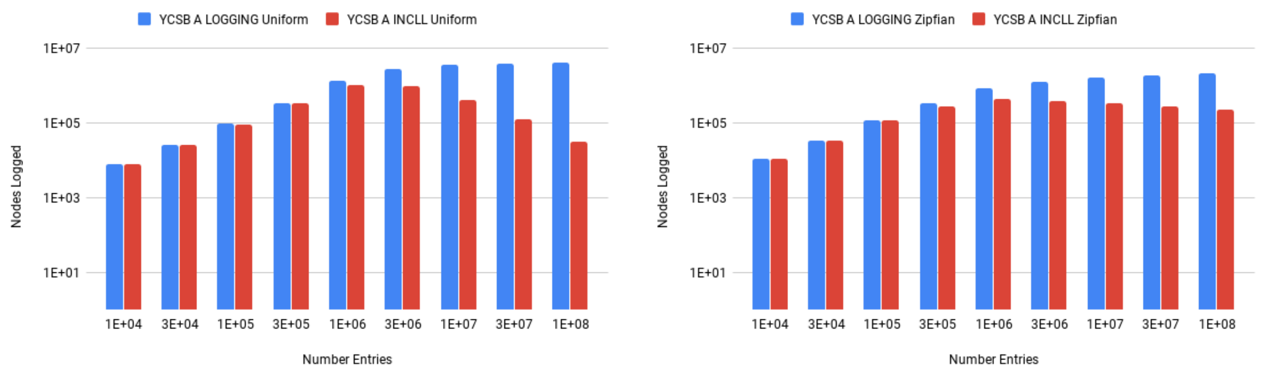


Figure 7. Number of logged nodes when InCLL logging is disabled (LOGGING) and with InCLL (INCLL).

distributions, the number of logged nodes increases sharply (logarithmic plots) until the tree reaches 1–3M nodes. After that point, the uniform distribution levels off without InCLL and declines rapidly with InCLL, and the zipfian distribution grows slowly without InCLL and declines slowly with InCLL.

A tree of 10K keys contains approximately 1K nodes, many of which are modified frequently by the approximately 80K operations during a typical epoch. A node modified a second time is usually recorded in the external log (§ 4.1.1). However, no logging is needed for the rest of the epoch, so subsequent modifications incur no overhead. For a tree with 100M keys, however, overhead is low for a different reason. When the tree is very large and keys are chosen uniformly at random, a node has a low probability of being modified and a lower probability of being modified twice. Thus, most of its modifications are logged by InCLL, not external logging. The Zipfian distribution, which has a higher locality of reference and more nodes access twice or more, does not benefit as much from the InCLL and its number of nodes logged continues to increase as the tree grows. In the middle, when the tree contains 1M–3M entries, the probability that a node is modified twice or more is relatively large. This entails external logging, but since the number of operations on a given node is likely to be low, the overhead of this logging is unlikely be amortized over a series of operations, as in smaller trees. Despite the relatively high overhead for trees in this range, the heavy-write benchmark ran at most 27% slower than MT+.

Figure 8 show how performance changes for both InCLL (INCLL) and only logging (LOGGING) as the latency of flushing modified locations to NVM increases on the YCSB_A workload. With InCLL, the performance decreases only 4.1% (uniform) and 5.7% (zipfian) as the latency of this operation increases to 1μs. With only logging, the performance decreases 42.5% (uniform) and 28.5% (zipfian) over this range. InCLL greatly reduces the number of cache lines that must be flushed, which becomes increasingly beneficial as the latency of this operation increases.

6.1 InCLL for Internal Nodes

An initial version of our system applied InCLL to internal nodes as well. This significantly reduced the number of internal nodes that were logged. However, it did not improve performance appreciable since many more leaf nodes are logged. Furthermore, InCLL reduced the width of internal nodes, resulting in slower tree traversals. Overall, applying InCLL to all nodes resulted in lower performance.

6.2 Global Flush

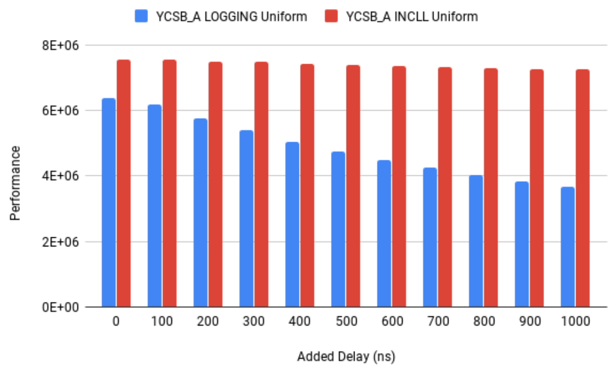
We measured the cost of a global cache flush using the three workloads. The instruction that flushes the entire cache, `wbinvd`, is a privileged instruction that can only be executed by the kernel. We measured user-space overhead: the time from the syscall until the operation returned to user space. In all measured cases, the cost was 1.38–1.39ms with a variance of 6-12%. Since the flushes are executed once every 64ms, the total cost of this operation is 2.2%.

6.3 Recovery Time

To measure recovery, we intentionally crashed the system immediately before starting a new epoch. This is a worst-case scenario for the number of nodes recorded in the external log. The workload is write-heavy (50% writes) and the tree was 1M entries (worst-case scenario for InCLL). We found that 84K nodes were recorded during the epoch. Applying these log entries required approximately 15ms. As expected, recovery is fast, even in a worst-case scenarios, primarily because of the short epoch duration.

7 Related Work

There have been many attempts to create efficient durable indexes in NVM. wB+-Trees [6] are a tree designed to reduce the number of writes to NVM by using unoccupied leaf entries for insertions, in a manner somewhat similar to the permutation field in Masstree. However, wB+-Trees still require at least two write backs and fences per update. NV-Tree [30] uses an append-only strategy to reduce the



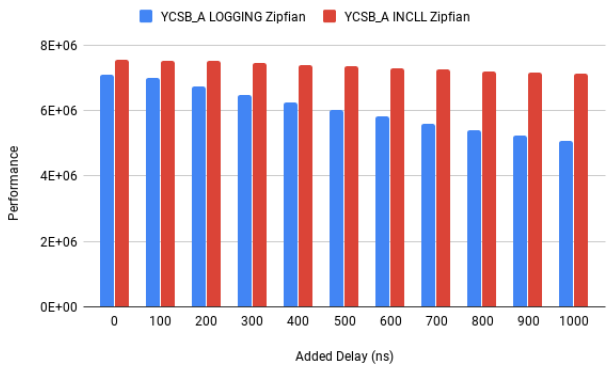


Figure 8. Performance with emulated latencies for cache write back when InCLL logging is disabled (LOGGING) and with InCLL (INCLL).

overhead of writing to NVM. However, every split requires reconstructing the path of internal nodes, increasing overhead. Update operations still require two write backs and fences. FPTree [24] reduces the overhead of NVM writes by storing only leaf nodes in NVM. Internal nodes are stored in DRAM and rebuilt during recovery. However, rebuilding the tree increases restart time significantly. In addition, it requires three write backs and fences per update operation. WORT [18] is a radix tree designed for NV, which attempts to reduce the number of writes to NVM. However, it still requires two write backs and fences per update operation. BzTree [2] uses a lock-free persistent multi-word CAS operation (PMwCAS) to implement a durable B+ tree. They do not report the number of write backs and fences (which might vary due to concurrency races), but at least two are needed for each PMwCAS. Insertions require at least two PMwCAS, so at least four persistent fences are necessary. Dalí [23] is a durable, nonblocking hash map based on globally flushing the cache. Each bucket follows an append-only strategy, so updating an existing value allocates a new node and appends it to the corresponding hash bucket. Therefore, Dalí makes less efficient use of the cache than Masstree. Memory reclamation relies on a garbage collection-like algorithm during recovery, which makes recovery very expensive.

Most programming interfaces to NVM are either transactional memory or locks. Mnemosyne [28] was the first system that used a software transactional memory system for NVM. It is based on a durable redo log whose implementation is from TinySTM [11]. PMDK [10] is a library, provided by Intel, that uses an undo log to provide durable transactions as an NVM interface. Atlas [5] uses locks to delimit uninterruptible durable regions. It uses a durable undo log to roll back unfinished atomic regions after a failure. The InCLL programming model is more complex, as it tailors logging to the semantics of a data structure, but it achieves far better performance than general-purpose approaches.

Numerous papers have tried to improve transaction performance without changing the programming model. None come close to the low overhead of InCLL. Kolli et al. [16] pipelined multiple stages to reduce the number of write backs and fences. NVThread [13] improved the lock-based durable section with a redo-log. Different threads are spawned as different processes, giving each thread a fast, hardware mediated view of its local modification. LSNVMM [14] improved the performance of durable transactions by avoiding replication of data in a log and the original location. Kamino-TX [22] avoids slow lookups in the redo log by applying modifications to a DRAM copy. These modifications are propagated lazily to NVM. DudeTM [19] decouples transactional concurrency control from the durability mechanism. Concurrency control is performed on a DRAM copy and produces a redo log, which is applied to NVM in the background. The latter two systems avoid write backs and fences on the critical path but suffer from a long restart delay due to the cost of copying the NVM structure to DRAM.

The main challenge that needs to be solved by a persistent memory allocator [3, 7, 23, 26] is inconsistency between the allocator's metadata and the application's data structures. If a buffer was allocated but the system crashed before it was linked to the persistent structure, it is a persistent memory leak. Schwalb et al. [26] broke allocation into two steps, reserve and activate, where each flushes data to persistent memory. A crash after the reserve step is rolled back by the system. The NV-heap [7] system implements allocation, automatic garbage collection, reference counting, and pointer assignments as simple, fixed-size ACID transactions using a persistent redo log. Makalu [3] uses a conservative garbage collector to recover unreachable pointers. Thus, no writes back are required during allocation. It has a slow restart time, due to its need to traverse a potentially unbounded amount of memory before an application restarts.

8 Discussion

Fine-Grain Checkpointing is not an exact replacement for transactions and InCLLs are not a general-purpose substitute for logs. Using InCLLs require detailed understanding of data access patterns and cache-line boundaries. Checkpointing requires less program annotation than transactions since a developer only needs to ensure data structure consistency at infrequent epoch boundaries, but this does not alleviate the complexity introduced by InCLL and does not guarantee immediate durability. Their combination provides a powerful tool for experts, such as library developer, for reducing the cost of durability, albeit at the cost of additional programming complexity.

Analogous to cache-efficient or concurrent data structures, we believe that efficient, recoverable structures cannot be created with one-size-fits-all techniques. Achieving high performance require code that is data-structure and architecture specific. In this paper, we use Masstree as an example to demonstrate how InCLLs make a highly optimized structure durable. The amount of effort was approximately 2 person months without prior knowledge of Masstree.

Other pointer-based structures could benefit from these techniques. Currently, achieving good results requires careful reasoning about the characteristic of the specific structures (e.g. § 4). We are aware there are still many open questions about how to apply this technique in other contexts — e.g., values that span a cache line, objects with less sharply defined update patterns, or more complex data structure manipulations — and are investigating more general solutions.

9 Conclusion

In this paper, we present Fine-Grained Checkpointing and In-Cache-Line Logging. The former periodically flushes the cache to NVM, thereby persisting everything. The latter embeds an undo log inside the cache lines in a data structure, thus enabling fast logging to undo writes from partially executed epochs. We transformed the Masstree data structure to be durable using these techniques and an external object-granularity log to handle complex and infrequently written structure operations. The combination of these techniques guarantees durability while introducing only a moderate performance overhead.

References

[1] Hiroyuki Akinaga and Hisashi Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE*, 98(12):2237–2251, December 2010.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: a high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, 2018.

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 677–694, New York, NY, USA, 2016. ACM.

[4] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[6] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[7] 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 XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[8] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang.*, 2(OOPSLA):153:1–153:22, October 2018.

[9] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):67:1–67:24, October 2017.

[10] Krzysztof Czurylo and Andy Rudoff. NVML: NVM Library, 2014.

[11] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[12] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEEE Int. Devices Meet. 2005. IEDM Tech. Dig.*, pages 459–462. IEEE, 2005.

[13] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. ACM.

[14] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 703–717. USENIX Association, 2017.

[15] Intel. Intel and micron produce breakthrough memory technology, 2015.

[16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. ACM.

[17] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[18] 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 *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 257–270. USENIX Association, 2017.

[19] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM.

[20] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[21] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th*

ACM European Conference on Computer Systems, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[22] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[23] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st Int. Symp. Distrib. Comput. - DISC 2017*, volume 91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[24] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[25] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[26] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, pages 61–72, 2015.

[27] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[28] 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 XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[29] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal-Oxide RRAM. *Proc. IEEE*, 100(6):1951–1970, June 2012.

[30] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.