CS736 Project 3 Final Report: Performance and Consistency Analysis of NVM-based B-trees

Suryadev Sahadevan Rajesh*
sahadevanraj@wisc.edu
UW-Madison

Shyam Murthy*
shyamm@cs.wisc.edu
UW-Madison

Guanzhou Hu*
guanzhou.hu@wisc.edu
UW-Madison

ABSTRACT

We present a thorough analysis of B-tree implementations running over local non-volatile RAM device (NVRAM, narrowly refered to as NVM in this report), covering its performance, consistency, and concurrency behavior. We study B-trees with three levels of NVM ordering constraints: *no-ordering*, *full-ordering*, and *with the FAST optimization* [9]. We show that NVM-based B-tree performance is critically sensitive to user workload *monotonicity* and *write ratio*. Interestingly, in contrast to intuition, it is not sensitive to workload *access locality*. On the throughput metric, concurrency helps fill the gaps between levels of ordering constraints down to 1.5x. On the consistency metric, we see that data stays volatile for a much shorter period with ordering constrains, but the average extent to which data remains volatile without ordering varies based on the workload access pattern, where we see shorter average volatility intervals with irregular access patterns (uniformly random data), compared to more regular access patterns (monotonic and reverse-monotonic). Inconsistency probabilities are also tied to the workload, varied contention in the cache and differing access patterns of the workload can change the probability of inconsistency.

1 INTRODUCTION

Non-volatile memory (NVM) devices are now commercially available starting with Intel's Cascade Lake and researchers are starting to understand its performance characteristics [18]. While NVM devices are performant and persistent, ordering needs to be ensured for NVM storage based system implementations to have crash consistency.

Typically, we see ordering lapses when we move data from a volatile storage to non-volatile storage. Similar to the case of hard disks where we move data from in-memory pages to the disk surface to ensure persistency, in the NVM world, we move data from the volatile CPU cache to the persistent NVM storage. To ensure ordering for crash consistency, we use `mfence` and `clflush` instructions which are typically supported in x86 architectures. The `clflush` instruction invalidates a cacheline from all levels of the cache hierarchy and flushes the data to memory if the cacheline has modified data. The `mfence` instruction ensures ordering between load/store instructions and is issued before and after every `clflush` instruction.

Our goal as part of this work is to understand and analyze the performance impact of the primitives used to ensure crash consistency, namely `mfence` and `clflush` operations [13]. In B-Tree implementations for NVM, missing memory ordering constraints poses some risks of losing data in NVM and also has the risk to put our storage in an inconsistent state. Since it is always possible for the developers to introduce redundant or additional memory ordering constraints and miss some of the memory ordering constraints at places that are required [12], it is interesting to study the effect of this on performance or on the correctness.

To study the inconsistency and volatility and understand its behaviour with different access patterns, we quantify the danger of inconsistency w/o ordering using the notion of *inconsistency windows* from OptFS [6]. As we expect, the probability of having an inconsistency is high in the absence of ordering compared to having full ordering. However we observe that the probability is highly dependent on the workload's access pattern, where the probability of an inconsistency could vary from close to 0, to being close to 1 in the absence of ordering. To study the risk of losing data in NVM when there is a crash or power-off, we study the notion of volatility in the NVM. Since CPU cache in our system is volatile and is in the path of NVM write from CPU, we study how long the data stays in CPU cache before getting flushed to memory. Additionally, we also quantify the time the data stays volatile before being persisted, to give us an idea of how much natural persistence the system has to offer.

In our project, we show that workload monotonicity and write ratio greatly affect latency performance of NVM based B-tree, resulting in a 64x latency gap between full ordering constraints and no ordering constraints in the worst case of write heavy, reverse-monotonic workloads. Workload locality, in contrast, does not affect latency much because the CPU cache is constantly bypassed/flushed, hence there is no effective caching mechanism here for write-heavy workloads. We also from our experiments show that how these different metrics that we proposed for consistency and volatility are dependent upon the different access patterns when different mode of NVM based implementation of B-Trees are used.

For the rest of this report, section 2 lists background knowledge on NVM device and NVM-based B-tree data structure. Section 3 defines our metrics for quantifying inconsistency: *volatility interval* and *inconsistency window*. We then describe our methodology of doing NVM-based B-tree experiments on different metrics. Following that, section 5 presents our results. Section 6 lists related work and section 7 concludes this report.

2 BACKGROUND

2.1 NVM Device & Consistency

NVM devices are now available in real production systems with the release of Intel Optane DIMMs [2]. Throughout this report, we use the abbreviation NVM to narrowly refer to non-volatile RAM devices connected directly to the processor through memory bus. The processor communicates with an NVM device through regular memory load/store instructions, treating it as a persistent DRAM.

*All authors contributed equally to this research.

The empirical guide paper [18] has a detailed explanation of NVM devcie internals and the semantics of its interface.

Consistency rises as an interesting issue for applications running over NVM directly. Because of the existence of the write-back CPU cache, updates may be cached as dirty copies in the cache and not be persisted to the NVM device. Those dirty data will not survive across a system crash. Similar to `fsync`'s on a disk-based file system, NVM applications need to insert `clflush` instructions at proper points in the code, surrounded by `mfence` instructions, to force flushing the dirty data back to the device. Otherwise, data may appear in an inconsistent state on persistent media. For example, suppose a queue data structure, the application might append a new element to the queue and then update a queue size counter in the header. Without a `clflush` in between these two updates, it is possible that the counter update gets evicted first by the CPU cache, while the new element is still volatile. This exposes the danger of losing the new element data if a crash happens, and reading out garbage bytes at that index after recovery.

2.2 NVM-based B-trees

We study standalone B-tree implementations running on a local NVM device, without the interference of an intermediate storage system layer. Such B-tree implementations assume their memory access all go to the NVM device sitting on the memory bus. In particular, we focus on the strength of *ordering constraints* it deploys to guarantee NVM storage consistency.

The most essential internal operation of NVM B-trees on a user insertion request is to insert a key into a sorted B-tree node key array. Figure 1 demonstrates the element swapping operations that could happend during an insertion request. The user is inserting key $k = 5$. In a B-tree node at some level, this key will be inserted into the sorted B-tree array. The key k will first be written to the end of the array. Then, it compares with the element on its left, k_l, and if $k_l > k$, we swap k with k_l. This procedure repeats until $k_l \leq k$.

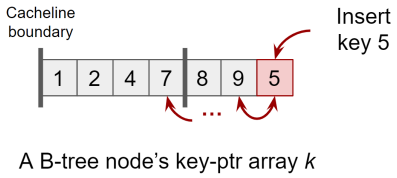

Figure 1: The Element Swapping Operation. The user is inserting key 5 into the B-tree, and in a node at some level, this key will be inserted into the sorted B-tree node array.

There are three different levels of ordering constraints a B-tree implementation could deploy: 1) *no-ordering*, does not do any `clflush` and `mfence`; 2) *full-ordering*, issues a cacheline flush after every swapping of two adjacent elements; 3) *fast-fair*, introduced by the FAST_FAIR paper [9] and described below.

The FAST_FAIR B-tree introduces a neat technique named FAST to reduce the number of necessary cacheline flushes, with the knowledge of cacheline size of the underlying architecture. When inserting an element into a sorted array (in this case, a B-tree node key-pointer array) by swapping elements starting at array tail, for any array chunk residing in the same cacheline, the swaps introduce natural dependency: `swap(a[i], a[i+1])` depends on `swap(a[i+1], a[i+2]` to have finished. Hence, the processor will internally enforce the ordering of these two swaps in order to deliver the correct result. Therefore, we only need to issue a `clflush` followed by an `mfence` every time we cross the cacheline boundary. In the example given in figure 1, FAST_FAIR will only trigger one cacheline flush on this node.

3 CONSISTENCY METRICS

We propose two metrics for quantifying memory inconsistency: *volatility interval*, the time that a piece of data resides in cache before getting written to memory, and *inconsistency window*, the interval at which the persistent state is semantically inconsistent.

3.1 Volatility Interval

Volatility interval gives us an idea how long the data is volatile in the system. Since for the NVM setup, the CPU cache is the volatile storage, we measure the duration for which the data resides in the CPU cache.

To measure the volatility interval we get the traces of the data cache writes and the memory writes from the memory controller logs. The data cache write contains the physical address range at which the data is written. These addresses are then converted to associated cache block addresses, since the memory controller logs contain only the addresses in the form of cache block.

$$volatility\_interval = memctrl_t - dcache_t$$

$dcache_t$ - refers to the time at which an address range is written in the data cache of CPU. $memctrl_t$ - refers to the time at which the corresponding data is written back to memory.

Our tool reports the mean, and 5th, 50th, 99th percentiles of the volatility interval.

To analyze the trace in the memory controller, we take two events into account: *Write Back* and *Write Clean*. The *Write Back* event happens when the cache blocks are written back from the CPU cache to the memory when the cache space need to be freed for the new cache block to get allocated. It usually happens when we are running the B-tree in no-ordering mode, or in some instances of the other modes (full ordering or fast-fair). When the clflush commands are used i.e. in full ordering and fast-fair modes, we could see *Write Clean* events in the memory controller logs. So our tool takes account into these two events.

In our algorithm, we match the corresponding cache block write in the memory controller that comes immediately after the corresponding dcache write based on the timestamps.

3.2 Inconsistency Window

Inconsistency window metric helps us in observing how far the persistent state has been inconsistent. So if a crash happens in this window then we will not be able to recover the data. We used the metric that is discussed in the OptFS[6]. We have illustrated with an example in the figure 2.

To measure the inconsistency window, we considered two static points in the program and used the traces to see if the memory

Algorithm 1: Volatility Interval Computation

```
Result: Inconsistency window probability
Input: map<cache_blk, memctrl_write_times>
  memctrl_map;
volatile_intervals = [];
while ((cache_blk, dcache_t) = read(dcache trace) != EOF) do
    if memctrl_map[cache_blk] != NULL then
        memctrl_times = memctrl_map[cache_blk];
        memctrl_t = memctrl_times.upper_bound(dcache_t);
        volatile_intervals.append(memctrl_t - dcache_t);
    end
end
mean_vinterval = mean(volatile_intervals);
99th_vinterval = percentile(volatile_intervals, 99);
```

updates are getting reordered when it is propagated from cache to memory. The three traces that we considered are: (a) logging virtual address of hdr and sibling (two static points in the program), (b) Virtual addresses to physical address translation, (c) Memory controller traces

We consider two static points in the program and collect the list of virtual address for these two memory references on its dynamic occurrences. Using the virtual page number to physical page number translations, we build the *pmap* in our tool and convert these virtual address to physical address and then to the corresponding cache blocks. We then use the cache blocks to see if the immediate next occurrence of these cache blocks in the memory controller logs are getting reordered based on the timestamps.

Algorithm 2: Inconsistency Window Computation

```
Result: Probability of occurrence of inconsistency window
Input: // Precomputed from trace
map<cache_blk, memctrl_write_times> memctrl_map;
// Precomputed from trace
map<vpn, ppn> pmap;
violation_count = 0;
total_count = 0;
// Consider two memory references a and b
while ((time, a_vaddr, b_vaddr) = read(dcache trace) != EOF)
  do
    a_cache_blk =
      compute_cache_blk(pmap(getPage(a_vaddr) +
      getOffset(a_vaddr)));
    b_cache_blk =
      compute_cache_blk(pmap(getPage(b_vaddr) +
      getOffset(b_vaddr)));
    if memctrl_map[a_cache_blk] != NULL and
      memctrl_map[b_cache_blk] != NULL then
        total_count += 1 ;
        a_memctrl_t =
          memctrl_map[a_cache_blk].upper_bound(time);
        b_memctrl_t =
          memctrl_map[a_cache_blk].upper_bound(time);
        if a_memctrl_t > b_memctrl_t then
            violation_count += 1;
        end
    end
end
inconsistency_window_probability = violation_count /
  total_count;
```

4 METHODOLOGY

In this section, we describe the tools we used, our environment setup, and our methodology for doing NVM-based B-tree analysis.

4.1 Performance Emulation

We use Quartz [16], a DRAM-based NVM emulator, to carry out performance analysis of NVM B-trees. Quartz is an epoch-based emulator composed of two parts: 1) a kernel module for checking hardware performance monitoring counters (PMCs) and handling epoch timing; 2) a user-level dynamic library for communicating with the kernel module and periodically injecting delays. In the basic operating mode, when a shell environment has been set up with Quartz, it treats any memory access from user processes running in that shell as NVM device access, and injects proper delay cycles according to the latency configuration.

We use the following NVM performance configuration according to the experiment results given by the empirical guide paper [18]:

- Target average latency: read 350 ns, write 1000 ns;
- Throughput upper bound: write 2400 MB/s.

Memory accesses happen frequently even with the presence of the CPU cache. Typically, an in-memory B-tree implementation without any NVM ordering constraints will introduce at least 1 memory bus access every 100 CPU cycles. The number of memory accesses will get significantly higher if the implementation tries to set ordering constraints by doing cacheline flushes. It is infeasible for an intermediate emulator to interfere with every memory bus load/store request. Quartz solves this problem by injecting delays only once per time window, called an *epoch*. The length of an epoch is typically set as $1 \sim 10$ milliseconds. At the end of an epoch, the kernel module notifies the user-level library, who in turn sends a pause signal to the user process. Once the pause is successful, the library queries the kernel module for PMC counter values, in particular, number of L3 cache load/store misses, then subtracts them with the PMC values at the end of last epoch. The difference between these two time points indicate the number of actual memory accesses that missed the L3 cache during this epoch and that should have been served by an NVM device. Quartz then calculates and injects the desired number of cycles to pause to account for this epoch.

4.2 Counting LLC Misses

The performance differences among different levels of ordering constraints take root in the number of last-level cache (LLC) misses that happen throughout a user operation. The Quartz emulator described in the previous section has a kernel module for reading the LLC misses PMC, however it has a rigid `ioctl` interface and cannot be used other than for performance emulation purpose. To

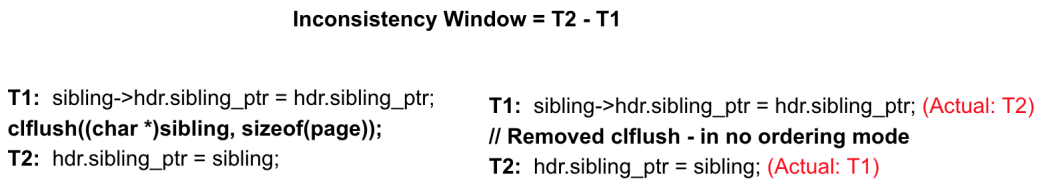

Figure 2: Inconsistency Window. Shows two static points in the program that gets reordered on is writing back to memory when the ordering constraints are removed. Also the formula that we use to calculate the inconsistency window is shown.

enable LLC misses counting at user request granularity, we use the portable PAPI library (Performance Application Programming Interface) [14]. The strength of PAPI is that it allows you to inline PMC reading with application code, instead of just running in a separate shell as a coarse-grained monitoring tool (`perf` and `bpftrace` are examples of this kind of monitoring tools).

We modify and instrument the B-tree implementations' entrance driver code with PAPI to enable cache misses counting for every user B-tree operation. A simplified code snippet looks like:

```
int papi_event_set = PAPI_NULL;
long long papi_values[1];

// Initialize PAPI.
PAPI_library_init(PAPI_VER_CURRENT);
PAPI_create_eventset(&papi_event_set);
PAPI_add_event(papi_event_set, PAPI_L3_TCM);
PAPI_start(papi_event_set);

// Perform a user B-tree operation.
PAPI_reset(papi_event_set); // reset to zero
btree->btree_search(key);
PAPI_read(papi_event_set, papi_values);
```

Notice that the `PAPI_L3_TCM` is the counter name for LLC misses and is available on all mainstream Intel server microarchitectures. With PAPI, we are able to record accurate numbers of NVM device accesses on a real production server.

4.3 Consistency and Volatility Evaluation

We use the gem5 [4] micro-architectural simulator, which is a cycle-accurate simulator, for our consistency and volatility analysis of the B-tree implementations. Gem5 is run in system-call emulation mode, because we are interested in the user-space data structure activity for the B-tree implementations alone. We use the simulator's x86 based in-order CPU model and simulate a two level cache hierarchy with a L1 data cache of size 32KB and an L2 data cache size of 256KB. We instrument gem5 to print list of virtual to physical translations to be able to correlate the virtual addresses corresponding to our data structure with the physical addresses we see at the memory controller. We make use of some of the debug functionality that gem5 has internally to generate timestamped memory traces at the L1 data cache and at the memory controller. At the L1 cache, it prints the range of addresses for which we perform a write. At the memory controller, it prints the physical address of the cache block we write to and additionally prints the kind of write that we're performing.

```
// Example from log at cache:
1e4375e250: WriteReq [388d8:388df]

// Example from log at the memory controller:
161097633000: system.mem_ctrls: recvAtomic:
WritebackDirty 0x451080

// Example translation instrumentation output:
Translation Virtual page number: 3a7
Physical page number: 409
```

These logs are used to compute the volatility interval and the inconsistency window across all invocations of a pair of static memory references which are expected to be ordered in an implementation with forced ordering. We collect a trace of the memory writes which happen in the L1 data cache. Here, we log the physical address of the cache block that gets written and the time at the which this write occurs, which constitute a timestamped memory trace. Similarly, we log the cache blocks that get written to the DRAM at the memory controller, where we log the cache block address as well as the kind of write that flushed the block to memory. The write may have been a part of a normal cache block eviction, or a forced cache line flush to ensure ordering, which we log additionally. We use Gem5 to collect translations from virtual to physical addresses, which we use to correlate the accesses coming from our data structure to the memory controller, where the logging is done completely using physical addresses.

To understand these traces, we defined the volatility interval and inconsistency window metrics in section 3 and built a tool for analyzing the traces and calculate these two metrics.

5 RESULTS

We present our experiment and analysis results on NVM-based B-tree performance, consistency, and concurrency behavior in this section. Our experiments and analysis are carried out using the methodology described in section 4.

5.1 Hardware Configuration

Performance experiments are sensitive to the underlying hardware configuration. We carried out our performance experiments on a CloudLab `c220g1`-type node. The node is equipped with Intel Xeon E5-2630 v3 CPU at 2.40GHz frequency. The processor has 32 cores

spread across 2 sockets, 32KB L1 data cache, 32KB L1 instruction cache, 256KB L2 cache, and 20MB shared L3 cache. Cacheline size is 64 bytes. The node has 128GB of DDR4 memory at 1866MHz frequency. The storage device is a single Intel DC S3500 480GB SATA SSD.

The dual-socket CPU composes of two NUMA sockets. To ensure an accurate emulation of NVM performance, we pin the B-tree thread to one core and limit all of its the memory accesses to go to only its local socket memory through Quartz. Non-performance experiments running on simulators will be marked out. They do not rely on this specific hardware configuration.

For all performance-related experiments, we emulate with Quartz an NVM device with read latency 350 ns, write latency 1000 ns, and write bandwidth upper bound 2400 MB/s.

5.2 Latency Performance

We show that the performance of NVM-based B-trees with ordering constraints is critically sensitive to two major characteristics of user workload: *monotonicity* and *write ratio*. We also show that, in contrast to our intuition on traditional storage systems, the *locality* of user workload does not affect NVM B-tree latency significantly, since the CPU cache is forced to be bypassed/flushed most of the time.

5.2.1 *Latency vs. Workload Monotonicity & Locality*

Figure 3 shows the latency performance running five different user workloads over the three versions of B-trees. The upper plot shows the average latency per user insertion request in nanoseconds, and the bottom plot shows the number of L3 cache misses behind these latency numbers. There are four critical observations we could get from this figure.

Number of LLC misses match the latency behavior. Comparing the top figure with the bottom one, we could see that the number of LLC misses perfectly match the latency seen by the user. It confirms that the NVM device latency processing memory bus requests is the major contributor to the overall B-tree operation latency. The B-tree data structure is memory-intensive but not computation-intensive.

Latency is sensitive to workload monotonicity if it has ordering constraints. We define *monotonicity* as the probability that the user is inserting a sequence of monotonically *increasing* keys in a row. The `monotonic` workload has monotonicity = 1, which means the user is purely inserting a monotonically increasing list of keys. The `reverse` workload has monotonicity = 0, i.e., the user is inserting a monotonically decreasing list of keys. The `uniform` workload is uniformly random and thus has a monotonicity close to 0.5. Comparing the latency across three synthetic workload monotonicity, we see that the performance of NVM-based B-tree is linearly dependent on monotonicity if it has ordering constraints, either with a full ordering scheme or with the FAST optimization. It is because the number of forced `clflush`'s comes from how often we do the linear swapping of elements in B-tree node arrays. With monotonicity = 1, every inserted key is larger than any existing keys, so ideally there will be no swapping at all. With monotonicity = 0, every inserted key will be swapped till the beginning of the node array, introducing a huge amount of swapping operations. A uniform dataset sits somewhere in between these two extremes.

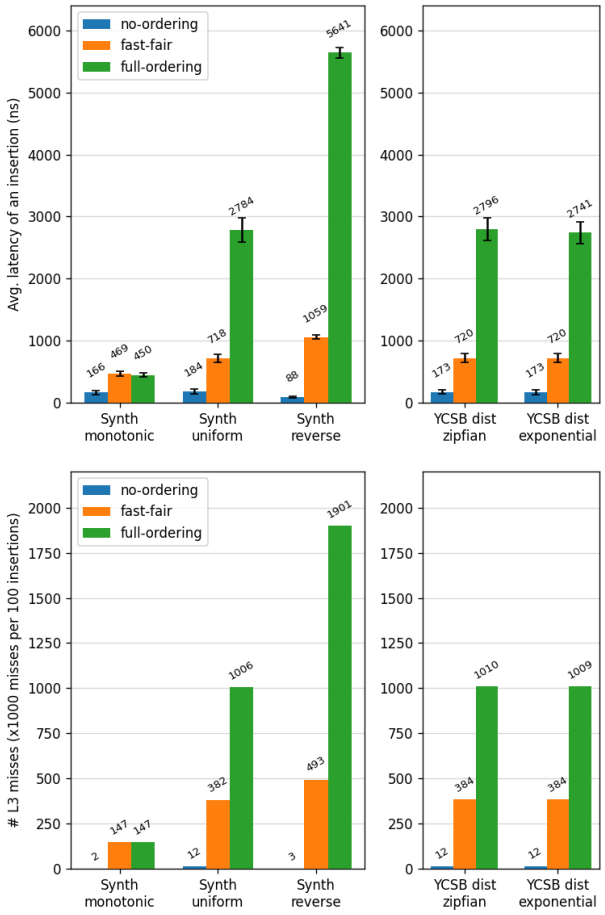

Figure 3: Latency (top) and Number of LLC Misses (bottom) on Insertion, with Different Levels of Ordering Constraints and Different Workload Monotonicity. The three groups on the left are synthetic workloads. The two groups on the right are YCSB-generated workloads with different locality distribution, representing uniform real-world B-tree workloads in key-value stores.

Latency is *not* sensitive to workload locality distribution, at least on local NVM devices. Synthetic uniform, YCSB zipfian, and YCSB exponential workloads all have nearly-uniform monotonicity, but with different locality distributions. The synthetic uniform dataset has no locality because it is purely uniformly random. The two YCSB traces have zipfian and exponential locaity distributions respectively. Results show that locality in access keys has no significant effects on insertion latency. The reason is that NVM devices are connected to the memory bus directly and there is no extra layer of caching apart from the CPU cache. When the CPU cache is constantly cleared/flushed by the consistency ordering constraints, there is effectively no caching effect in NVM B-tree implementations. This is true at least when running the B-tree on local NVM devices. With remote NVM enabled through technologies such as RDMA, this statement may no longer hold.

Full ordering constraints introduce huge overhead, but the FAST optimization brings it down. Comparing no-ordering

version of the B-tree with full-ordering version, we can see that adding in strict ordering constraints into the system brings latency performance significantly worse, up to 64x higher. This is because the no-ordering version can fully utilize the three levels of CPU cache, while the full ordering version flushes a cacheline from all three levels with `clflush` every time it modifies any element residing in that line. The good news is the FAST optimization does a great job bring that penalty down, by up to 5x. This is expected with 64 bytes cacheline size and 64 bits integer keys. Every cacheline holds 8 array elements, hence every 6 or 7 swappings in a row of 8 swappings will be eliminated by the FAST optimization. This will bring a speedup in range $8/(8-6) = 4 \sim 8 = 8/(8-7)$.

5.2.2 *Latency vs. Write Ratio*

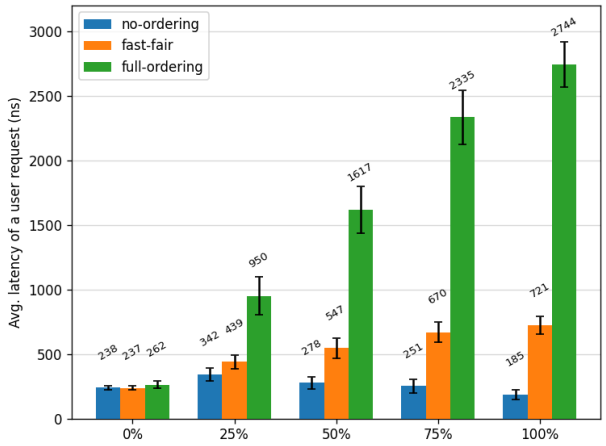

Figure 4: Average Operation Latency on Workloads with Varying Write Ratios. All five workloads have a uniform monotonicity and a YCSB zipfian locality distribution.

Figure 4 shows the latency performance running workloads with different write ratios. Here, the x-axis shows how much of the operations in the workload trace are insertions (writes). The remaining operations are searches (reads).

Insertions domininate NVM B-tree performance if it has ordering constraints. When we shift the write ratio from 0% to 100%, the average latency to complete a user request on a full-ordering B-tree goes up to 10x higher, and for the FAST_FAIR B-tree up to 3x higher. This shows that insertions dominate NVM B-tree performance. Insertions will trigger element swapping in B-tree node arrays, which in turn will trigger expensive cacheline flushes. Without consistency guarantee, the no-ordering version is allowed to utilize the CPU cache and the cost of a node swapping is no larger than a linear search in the cache. Hence, the no-ordering version is not sensitive to workload write ratio at all.

5.3 Concurrent Throughput Performance

Real-world deployment of B-trees often exploit the potential benefit of concurrency. In this case, multiple B-tree worker threads accept user requests and serve these requests concurrently. Locking is often implemented in a fine-grained way, where a thread locks individual B-tree nodes used by its request, instead of locking the whole tree. The overall throughput of a concurrent NVM B-tree will be bounded by the bandwidth of the NVM device.

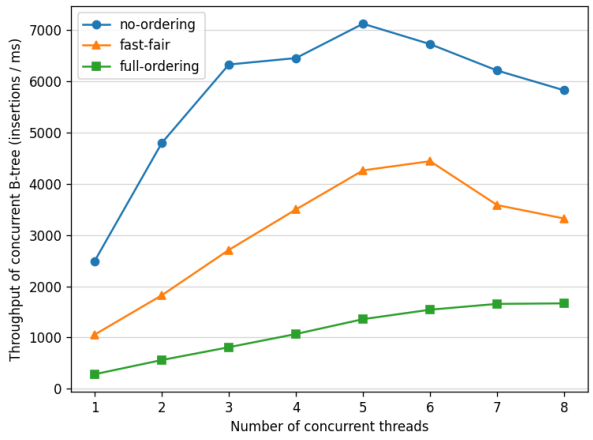

Figure 5: Throughput of Concurrent NVM B-tree Insertions vs. Number of Worker Threads. All experiment rounds run the YCSB zipfian workload with a total number of 2000000 insertion requests, distributed uniformly across all worker threads.

Figure 5 shows the throughput performance against the number of concurrent worker threads. For this particular experiment, we pin the worker threads to CPU cores in one NUMA socket, in order for Quartz to produce accurate bandwidth emulation. There are two observations we could get from this figure.

NVM B-tree does not benefit a lot from parallelism. In all cases, the NVM device bandwidth saturates at 5-7 concurrent worker threads. The reason is two-fold. First, the device is connected through the memory bus, which does not support enough internal parallelism compared to external SSD storage. Second, NVM device is fast enough and only a few threads are able to saturate its bandwidth.

Concurrency fills the gap between the three levels of ordering constraints. For a single thread, the latency difference of a write-heavy workload between no-ordering and FAST_FAIR can be up to 3.8x. The difference between no-ordering and full-ordering can be up to 14.8x, according to figure 4. With the help of concurrency, though no-ordering still performs better than the other two levels with consistency guarantee, the gap between them has been reduced to 1.5x and 3.9x respectively. This is because worker threads interleave with each other and the penalty of doing `clflush`'s is partially overlapped by other threads' computation. It indicates that, for a real deployment of concurrent NVM B-tree in highly-parallel systems such as key-value stores, the penalty of having ordering constraints is not as bad as it may seem on a single thread.

The throughput drop happening on the right half of this figure is due to the way Quartz injects delays to emulate memory bandwidth throttling. We expect the curves to be flatter on a real NVM device. We did not look into lock contention problems in our B-tree implementation, because we believe it is more related to data structure design and is orthogonal to its performance on NVM devices.

5.4 Measuring Consistency Behaviour in NVM

We measure and quantify the consistency behavior of the B-tree implementation, making use of the two metrics *inconsistency window* and *volatility interval*. We also log the number and kind of *memory writes* issued within the application to the memory controller, which we use to quantify the amount of ordering enforced in each of these applications to ensure consistency on a crash. Measurements were done using the timestamped memory traces collected using the cycle-accurate and deterministic simulations carried out in gem5. More details follow in the coming subsections.

5.4.1 *Writes Behavior*

Memory writes are classified into *write-backs* and *write-cleans*. Write-cleans correspond to writes that were forcibly flushed from the cache to DRAM to ensure ordering, and write-backs correspond to writes that were written back from cache to DRAM on a normal cache eviction. We have no write-cleans for no-ordering and the most write-cleans for full-ordering based on the definition of ordering constraints.

From Figure 6, **we see that the no-ordered implementation has the fewest number of memory writes** as they do no extra work to ensure ordering across memory references, and hence induce no extra flushes to memory and have their writes composed of write-backs alone.

For the uniformly random and reverse monotonic datasets, the number of writes are more for full ordered B-tree implementation compared to fast-fair implementation, because we are able to successfully leverage the optimization offered by fast-fair optimization and hence induce a lot less writes compared to a naively ordered implementation. We see the number of write-cleans to be a lot lesser for the fast-fair implementation compared to the fully-ordered implementation.

However, for the monotonic dataset, fast-fair optimization never gets triggered, causing the same number of writes to be issued in both the fast-fair and full-ordered implementation and hence we see the write-cleans (orderings) to be the same for both implementations and write-backs to be almost the same.

5.4.2 *Volatility Interval*

We measure the volatility interval for a *static pair of memory references* for the no-ordered B-tree implementation as shown in Figure 2, where the ordering has been relaxed. The volatility interval is measured with different workloads, which are a uniformly random dataset, a monotonic dataset and a reverse monotonic dataset.

Volatility interval computes the amount of time data stays in volatile storage before being persisted to memory. For implementations that impose less or no ordering, this metric can be used to assess how much persistence we get naturally from the system, without enforcing a lot of ordering.

Clearly from table 1, **we see that no-ordering implementation has a larger volatile interval compared to the full-ordered implementation**, because we have no ordering constraints that we enforce in the no-order implementation, and hence data tends to stay in volatile storage longer.

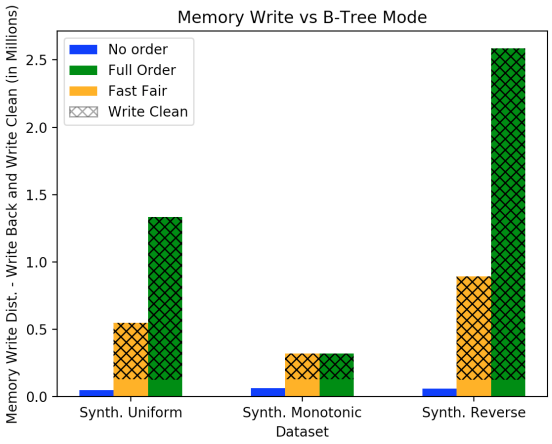

Figure 6: Distribution of Memory Writes for Different Workloads. All experiment run with 1000000 insertion requests with the distribution of interest.

Additionally, for the no-ordering implementation, we also observe that the volatility interval is dependent on the kind of workload we are running. A more *regular access pattern* (reverse monotonic or monotonic) induces lesser cache contention and hence causes data to stay in the volatile storage longer compared to a more *irregular access pattern* (uniform random). Clearly we observe the average and the lower percentiles of volatility intervals to be smaller for the uniformly random distribution compared to monotonic and reverse-monotonic workloads, for both implementations from table 1.

5.4.3 *Inconsistency Window*

We measure the inconsistency window across the same static pair of reference as shown in Figure 2, where the ordering has been relaxed. The inconsistency window is measured with different workloads, which are a uniformly random dataset, a monotonic dataset and a reverse monotonic dataset.

We observe that the inconsistency window shows sensitivity to the nature of the workload as well. This is because, when we have increased contention in the cache, then inconsistency seem less probable because there is a likelihood of both the blocks which are to be ordered experiencing contention and getting evicted soon. We observe that there are a lot few inconsistencies with the uniformly random dataset compared to the monotonic dataset, as shown in Table 2. Uniformly random dataset causes increased contention in the cache, as we see from the increased cache miss rate shown in same table, because of the irregular data access pattern, compared to reverse monotonic and monotonic datasets which have more regular data access patterns. Higher cache miss rates would mean that we are getting some amount of implicit flushing via increased contention in the cache, which has the ability to even eliminate some of the inconsistencies. For the reverse workload we observe something even more interesting, we see that the blocks to be ordered second stays the same across all dynamic references to same static code. So, the second block tends to stay in volatile storage for long, while the block to be ordered first is

Volatility Interval (us)						
No-ordering	1 %ile	5 %ile	50 %ile	95 %ile	99 %ile	Avg.
uniform	1199.87	1876.39	14541.07	27148.09	28481.44	13872.90
monotonic	12680.86	13719.48	15298.04	16699.99	16783.85	15259.96
reverse	11659.27	12399.27	14084.85	15190.76	15334.89	14031.15
full-ordering	1 %ile	5 %ile	50 %ile	95 %ile	99 %ile	Avg.
uniform, *monotonic*, *reverse*	0.11	0.11	0.12	0.13	0.14	0.12

Table 1: Volatility Interval. Measured with different workloads and B-tree implementations with no-ordering and full-ordering.

Dataset	Inconsistency Window		
	Prob	Avg. time (us)	Cache-miss rate (%)
uniform	0.49	2426	4
monotonic	0.96	79	0.2
reverse	0	–	0.2

Table 2: Inconsistency Window with Different Workloads. Measured for the B-tree implementation with no-ordering.

persisted at different times, bringing down the inconsistencies. The probability of an inconsistency is heavily dependent on the access pattern of the references within the workload which are expected to be ordered.

6 RELATED WORK

6.1 NVM Device

Intel Optane DIMMs [2] are recent successful examples of commercially available persistent memory devices. Yang et al. provide unwritten contracts from their experiments as four principles on how to build and tune systems for Intel Optane DIMMs [18], including avoiding small random accesses and limiting the number of concurrent threads accessing the device.

6.2 NVM Consistency

Through the exploration of NVM devices, crash consistency has been brought to attention. NVM devices connect to the system through memory bus, hence the CPU controls the device directly by cache promotion/eviction, or more infrequently, by store commands which bypass the cache. Earlier NVM based file systems such as BPFS [7], employ *short-circuit shadow paging* to provide atomic and fine grained updates to persistent storage. More recent general-purpose NVM file systems such as NOVA [17] use a *journaling* technique that records NVM updates in a separate log before doing it in-place, much like traditional disk-based file systems. Ordering of operations is enforced with `mfence` and `clflush` instructions. Recent work on *memory persistency models* [13] has also focussed on providing a more general framework to reason about ordering of writes in applications and expressing constraints on their order, building on the learnings from multiprocessor memory consistency models. Richer interfaces either lead to developers overusing or under utilizing interfaces leading to either lower performance or program bugs, similar to problems seen with multiprocessor memory consistency models [8].

Ad-hoc NVM data structures [5, 9, 10, 15] do not run over a general file system, but instead assume their memory accesses go to the NVM device directly. Therefore, they may deploy their own ordering constraints based on the data structure internals.

OptFS [6] studied the ordering behavior of journaling file systems and introduced the notion of *inconsistency windows* to demonstrate inconsistency vulnerabilities. We port this notion to NVM ordering mechanisms to help analyze their consistency properties.

6.3 B-Trees on NVM

There are existing works which study consistent and durable data-structures in the case of non-volatile and byte addressable memory [15]. They show good performance by avoiding extra log writes and by avoiding file system overheads. Another work on persistent B+-trees [5] proposes a new modified B+-tree, which minimizes the movement of index entries on insertions and deletions, thereby avoiding writes and shows good improvements in performance.

FAST_FAIR [9] is one of the latest works on NVM-based B-trees with publicly available source code. It has been explained in detail in section 2.

In CAWBT [10] (Cache line sized Atomic Write B+tree), the search and insert operations are optimized by converting the size of the operation to the size of the cacheline. Since `clflush` and `mfence` instructions are expensive, by using cacheline sized atomic units we minimize the use of cache ordering instructions.

6.4 NVM Bug Analysis

Unlike some of the existing tools like XFDetector [11] which find the persistent memory bugs via developer annotations, the Agamotto tool [12] solves the problem by using symbolic execution. It uses a Persistent Memory model to track the updates in the memory region based on the type of operations, and when it sees some operations (i.e `mfence`, `clflush`) missing, it reports a correctness

bug, or when some operations are redundant, then it reports a performance bug.

For state-space exploration, it initially does a static analysis to determine instructions that modify persistent memory and assigns priorities to other instructions by back-propagation algorithm based on the number of PM write instructions that are reachable from that instruction. This way the search space for the state exploration can be reduced by avoiding paths that do not modify persistent memory.

6.5 Experiment Tools

For our performance experiments, we used the DRAM-based epoch-based NVM emulator called Quartz [16]. It has been described in detail in section 4.

Hardware PMC counter reading is achieved using the PAPI library [14]. Other monitoring tools such as old Linux `perf` [3] and recent Linux eBPF [1] can be used to insert probe points into arbitrary hook points into the kernel, gather statistics, and then monitor the statistics in a separate shell. We did not find these tools helpful for our purpose, since memory operations do not interfere with the kernel unless at page allocations/unmapping. They do provide hardware-hooked PMC reading functionality, but it cannot be easily inlined with application code for doing accurate counter reading at user request granularity.

Architecture-level memory tracing is achieved by running the binary over gem5 [4], a powerful microarchitectural simulator, and collecting the memory operations seen by the memory controller and the L1 data cache.

7 CONCLUSION

In conclusion, we present a thorough analysis of B-trees running directly over local NVM device. We cover its latency performance and concurrent throughput performance. We quantify its consistency properties with the *volatility interval* and the *inconsistency window* metrics. Results show that ordering constraints add in tremendous overhead to single-threaded B-tree by up to 64x. Its latency depends greatly on workload monotonicity and write ratio, but less on access locality due to frequent cache flushing. Concurrent B-tree fills the gap between levels of ordering constraints down to 1.5x, showing that concurrency helps B-trees to fully exploit the bandwidth limit of the device. Results show that the volatility intervals are a lot higher for B-tree implementation with no-ordering, because there are no ordering constraints here, compared to an implementation with full-ordering. Additionally, we observe that the volatility intervals for the implementation with no-ordering is dependent on the nature of workload. Results also show that the probability of having an inconsistency is also dependent on the kind of workload being run, where we see probabilities of inconsistency varying between 0 for reverse-monotonic to almost 1 for the monotonic dataset.

REFERENCES

[1] [n.d.]. The eBPF Official Site. https://ebpf.io/. Accessed: 2020-12-12.

[2] [n.d.]. The Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2020-12-12.

[3] [n.d.]. The Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-12-12.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[6] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 228–243. https://doi.org/10.1145/2517349.2522726

[7] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[8] Mark D Hill. 1998. Multiprocessors should support simple memory consistency models. *Computer* 31, 8 (1998), 28–34.

[9] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, USA, 187–200.

[10] Dokeun Lee, S. Lee, and Youjip Won. 2019. CAWBT: NVM-Based B+Tree Index Structure Using Cache Line Sized Atomic Write. *IEICE Trans. Inf. Syst.* 102-D (2019), 2441–2450.

[11] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1187–1202.

[12] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[13] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 265–276.

[14] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[15] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California) *(FAST'11)*. USENIX Association, USA, 5.

[16] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 37–49. https://doi.org/10.1145/2814576.2814806

[17] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[18] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.