Clobber-NVM: Log Less, Re-execute More

Yi Xu
University of California, San Diego
La Jolla, USA
y4xu@eng.ucsd.edu

Joseph Izraelevitz
University of Colorado, Boulder
Boulder, USA
joseph.izraelevitz@colorado.edu

Steven Swanson
University of California, San Diego
La Jolla, USA
swanson@cs.ucsd.edu

ABSTRACT

Non-volatile memory allows direct access to persistent storage via a load/store interface. However, because the cache is volatile, cached updates to persistent state will be dropped after a power loss. Failure-atomicity NVM libraries provide the means to apply sets of writes to persistent state atomically. Unfortunately, most of these libraries impose significant overhead.

This work proposes Clobber-NVM, a failure-atomicity library that ensures data consistency by reexecution. Clobber-NVM's novel logging strategy, *clobber logging*, records only those transaction inputs that are overwritten during transaction execution. Then, after a failure, it recovers to a consistent state by restoring overwritten inputs and reexecuting any interrupted transactions. Clobber-NVM utilizes a clobber logging compiler pass for identifying the minimal set of writes that need to be logged. Based on our experiments, classical undo logging logs up to 42.6× more bytes than Clobber-NVM, and requires 2.4× to 4.7× more expensive ordering instructions (e.g., `clflush` and `sfence`). Less logging leads to better performance: Relative to prior art, Clobber-NVM provides up to 2.5× performance improvement over Mnemosyne, 2.6× over Intel's PMDK, and up to 8.1× over HP's Atlas.

CCS CONCEPTS

• **Hardware → Emerging technologies**; **Non-volatile memory**; • **Software and its engineering → Software libraries and repositories**; **Compilers**; • **Computer systems organization → Processors and memory architectures**; • **Information systems → Storage class memory**.

KEYWORDS

Non-volatile Memory, Persistent Memory, Compiler, Storage Systems, Undo Logging, Clobber Logging

ACM Reference Format:
Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446730

1 INTRODUCTION

Non-volatile memories (NVMs) can expose persistent storage as fast, byte-addressable main memory, and allow the processor to access persistent data via load and store instructions directly using the memory bus.

The durability of NVMs enables applications' in-memory data to live beyond process lifetimes and even across system reboots and unexpected power failures, but leveraging this capability is not simple. As caches are volatile, their contents cannot survive a power loss, and, since caches may delay evicting a modified cache line, writes may not reach NVMs in program execution order. These limitations mean that in the case of an unexpected power loss, a set of logically atomic updates may be torn with only a subset of them reaching NVM, leaving persistent data in an inconsistent state.

NVM libraries aim to facilitate NVM programming. These libraries provide *failure-atomicity* for specified code regions: all writes within a specified code region will survive a power loss and become persistent in NVM, or none will. These transaction-like, "all-or-nothing" semantics make programming on NVM easier and hide architectural and caching details from programmers.

Failure-atomicity libraries give programmers the ability to designate failure-atomic code regions (transactions), but this support comes with high performance overhead. Most industrial failure atomicity systems [5, 48] use an undo logging approach. In undo logging systems, where incomplete transactions are undone, the logging of the old value must occur before each write. Undo-based systems have an significant advantage in that reads do not need be redirected via an interposition layer, but at the cost of many expensive persistence ordering fences [46].

To avoid the high logging cost, JUSTDO logging [33] proposed *recovery-via-resumption*. In contrast to undo logging, JUSTDO logging tracks enough program state (including the program counter) to *resume* a failure-atomic operation at recovery, resuming execution from the interrupted instruction. Subsequent work in iDO logging [42] dramatically increased performance by exploiting regions of code that are idempotent (a segment of code that does not overwrite its inputs). However, the runtime overhead of these systems remains quite high.

In this work, we propose Clobber-NVM, an NVM library that ensures failure-atomicity by reexecuting interrupted transactions. Clobber-NVM relies on a new logging method — *clobber logging*, which merges undo logging with recovery-via-resumption. Clobber logging undo logs any transaction inputs overwritten during transaction execution. If a transaction is interrupted by a failure, clobber logging recovers by first restoring the transaction's overwritten inputs then, subsequently, reexecuting the transaction to completion. This strategy result in our system requiring far less logging than traditional undo-based systems since it only logs a few inputs — in our experiments we reduce the log sizes by 1.1× to 42.6×, and the log count (ordering fences) by 2.4× to 4.7×.

This paper makes the following contributions:

- It presents *clobber logging*, a novel, recovery-via-resumption strategy that reduces log size and ordering fence frequency by only recording overwritten transaction inputs.
- It presents a clobber logging compiler pass for identifying those transaction inputs that need to be logged.
- It introduces Clobber-NVM, a compiler-based failure-atomicity solution based on recovery-via-resumption.
- It demonstrates that Clobber-NVM's performance compares favorably with the existing state-of-the-art systems, providing up to 2.5× improvement over Mnemosyne and 2.6× over Intel's PMDK, and up to 8.1× improvement over HP's Atlas.

The rest of this paper is organized as follows. Section 2 provides some background on NVMs and motivates Clobber-NVM. We discuss the clobber logging design and Clobber-NVM system implementation in Section 3 and Section 4, respectively. Section 5 showcases the performance of Clobber-NVM. We discuss related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND

This section provides necessary background and clarifies assumptions about our target systems. We begin by describing the expected machine model and programming model, then provide necessary concepts for our program analysis.

2.1 Machine Model

Clobber-NVM is designed for a modern machine equipped with some nonvolatile memory (e.g. Intel Optane DC PMM's [32]). This multicore, cache-coherent machine contains a set of processing cores, with private and shared write-back caches. The caches contain a program's current working set, loaded from the backing memory, which consists of both nonvolatile memory and traditional DRAM. Stores issued by the cores modify cache lines in the caches; subsequent cache line evictions write the modified data back to either volatile (DRAM) or nonvolatile memory, depending on the cache line's physical address. The hardware manages the cache line eviction policy, it may not evict cache lines in the same order they were modified. Moreover, writes are not persistent as long as they reside in the volatile cache.

The existence of volatile caches with uncontrolled eviction policies means that the programmer needs to reason about the order in which data updates reach nonvolatile memory. For example, in a stack `push()` operation, the node must be created *and made persistent* before pointed to by the `top` pointer. Otherwise the `top` pointer could be evicted from the cache and become persistent, while its target node, still in the volatile caches, could be lost in a power outage, leaving behind a dangling, persistent, pointer.

To avoid these inconsistencies, programmers must use cache-flush and memory-barrier instructions to enforce the ordering of stores into NVM. For example, on Intel CPUs, the `clflush` or `clwb` instructions explicitly force a dirty cache line into memory, and `sfence` ensures subsequent writes will not complete until previous flushes that issued before the `sfence` have reached memory. However, frequent ordering fences limit the overlapping of long-latency flush instructions, result in high runtime overhead [28].

We expect hybrid machines with both NVM and DRAM in the near term. It is important for the programmer to specify whether the manipulated memory is volatile or not. Programmers can mark regions of the program's address space nonvolatile, and these memory regions are associated each with some named file in a NVM-aware file system [9, 57, 59]. In the event of a failure, the file system can remap the file into another process's address space for recovery and further use. For managing, naming, and allocating from these NVM regions/files, we leverage Intel's PMDK library.

2.2 Programming Model

Clobber-NVM is one of many *failure-atomicity libraries* that provide a simpler programming model for NVM: we describe this model here. With these libraries, a programmer can designate a region of code to be *failure-atomic*, that is, all of the code region's effects will survive a failure (e.g. power loss), or none will. Once the effects of the code region are guaranteed to survive a crash, the operation is *committed*. In general, in order to ensure failure-atomicity, it is necessary to log extra information during normal execution to support *recovery* after a failure. Recovery code can then use this extra, *logged* information to clean up interrupted failure-atomic operations and return back to a consistent state.

In the literature, these failure-atomic code regions (or *operations*) are often termed *failure-atomic sections (FASEs)*, or *transactions*. Generally speaking, the boundaries of the transaction are programmer defined using some interface of the underlying failure atomicity runtime. Note that these transactions are not traditional ACID (Atomicity, Consistency, Isolation, Durability) transactions [25, 27] — depending on the system, they may or may not provide isolation. That said, many failure-atomicity libraries link failure atomicity to concurrency control, either by building full ACID semantics or marking lock-protected critical sections as failure-atomic regions.

Clobber-NVM is a recovery-via-resumption failure-atomicity system. In these systems, interrupted transactions are resumed; this strategy stands in contrast to more traditional undo [5, 48] and redo [29, 41, 55] logging based systems, where interrupted failure-atomic updates roll back. Recovery-via-resumption requires saving sufficient program state such that resumption is possible.

Clobber-NVM uses an interface of transactions conceptually compatible with Intel's PMDK [48] library. Following this interface, we expect the programmer to explicitly mark failure atomic transactions. Furthermore, following PMDK's concurrency model, we expect transactions to acquire and release locks in a conservative, strong strict two-phase locking pattern [49, 56]: that is, locks protect memory locations from data races; transactions acquire the associated lock at transaction begin and in a fixed order (to prevent deadlock); transactions hold the locks until transaction commit. Assuming the programmer follows these constraints, both PMDK and Clobber-NVM transactions provide true ACID semantics.

2.3 Program Analysis Definitions

Clobber-NVM's design relies on compiler dependency analysis to minimize the amount of logging needed for proper reexecution and recovery of interrupted transactions. We describe key concepts required for this analysis here.

A Clobber-NVM *transaction* is a programmer-delineated code region with a single entry and (possibly) multiple exits. Following standard terminology [15], a code region (resp. transaction) *output* is a variable value which is assigned within the region and is *live-out* from its exit. That is, a region output is a value written in the region and read after it. Similarly, we define a code region (resp. transaction) *input* to be a variable value that is *live-in* to the region and used within the region. That is, a region input is a value assigned before the region's execution and read within it. Note that both inputs and outputs refer to values, not variables (a natural consequence of LLVM's SSA-based [19, 21] program represention).

A code region is *deterministic* if, for a given set of inputs, it always produces the same set of outputs. Clobber-NVM expects its transactions to be deterministic, and to not cause runtime errors or program exits (e.g., segmentation faults). This assumption is universal for recovery-via-resumption systems [2, 33, 42].

3 CLOBBER LOGGING DESIGN

Clobber logging is a recovery strategy that minimizes logging calls and log size by recording overwritten inputs to a transaction. If the transaction is interrupted, recovery proceeds by restoring the overwritten inputs then reeexecuting the transaction. Clobber logging's key insight is that logging only overwritten inputs is sufficient to reexecute a transaction with the exact same results, and, as a consequence, values at other addresses never need to be logged, since they will be overwritten upon reexecution.

Following the concepts introduced in Section 2.3, we deem a transaction input a *clobbered input* if it may be overwritten within this transaction, and term this write a *clobber write*. Clobbered inputs are a problem for reexecution. If an input is clobbered during transaction execution, reexecuting the code will use a new value for the input.

3.1 Undo-Then-Reexecute

To understand how clobber logging can ensure failure atomicity, we first describe a naive version of clobber logging that ignores dependency analysis. We term this explanatory failure-atomicity strategy *undo-then-reexecute*.

At every store, undo-then-reexecute records an undo log entry containing the old, overwritten value. On recovery, it replays the undo log backwards, leaving erasing all effects of the transaction. Instead of stopping at this point (as conventional undo logging would), undo-then-reexecute then reexecutes the transaction from start to finish.

Undo-then-reexecute differs from classical undo logging in a few notable ways. First, it recovers the program state to a point after the interrupted transaction, as opposed to before. Second, once started, a transaction never rolls back, so a transaction can be marked as committed soon as it begins. Finally, undo-then-reexecute assumes that inputs unmodified by the transaction will be available for reexecution during recovery. In our machine model, this assumption does not hold for data in DRAM or shared with other threads. Inputs in volatile memory will disappear during a power failure, and shared inputs might change prior to reexecution.

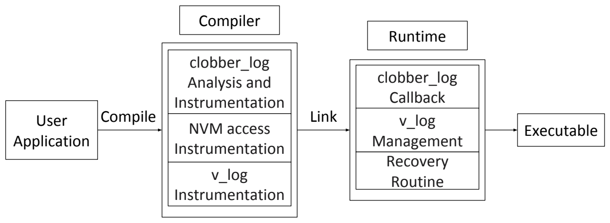

Figure 1: Clobber-NVM system overview

A correct version of undo-then-reexecute must address these challenges: First, it must make a persistent copy of volatile inputs so they are available after restart. Second, it must adapt a concurrency model that prevents transaction inputs from being changed after being read and transaction outputs from being read before commit (e.g. through its locking scheme).

3.2 Improving Performance

Undo-then-reexecute and undo logging are both slow since they log the same information at every store. For a NVM library to optimize undo-then-reexecute, it should attempt to remove extraneous logging. First, it should understand what logging is truly necessary in order to ensure that reexecution gives the same result. Second, it should understand what logging operations are unnecessary because it will reexecute the transaction.

To ensure reexecution of a transaction gives the same result, the transaction needs to be deterministic and have the same inputs (arguments and memory state) as the previous interrupted execution. This is why the undo-then-reexecute strategy needs the "undo" step: it needs to revert the transaction's clobbered inputs to their unmodified state.

However, undo logging records more than the clobbered transaction inputs: it also records the old values before writes to the transaction output addresses (e.g. modified memory locations). Traditional undo logging relies on these records to roll back the transaction's changes. But undo-then-reexecute does not need to undo its changes. Because it will reexecute the transaction, and the transaction will write the same values to the same memory locations, any written data from the previous execution will be regenerated and overwritten.

3.3 Clobber Logging

The clobber logging strategy is simply this: undo-then-reexecute, but *only* undo log before clobber writes. Like any correct undo-then-reexecute strategy on a persistent memory system, clobber logging needs to preserve volatile inputs and adapt an appropriate concurrency scheme. Clobber logging optimizes undo-then-reexecute by logging the clobbered inputs and ensuring all non-clobber inputs are available during recovery. Then, it reexecutes the transactions to produce the correct results. As we will show, clobber logging require less logging and incurs lower runtime overheads than conventional undo logging.

4 CLOBBER-NVM IMPLEMENTATION

The Clobber-NVM failure-atomicity system is a joint compiler/runtime library. Key system components are shown in Figure 1.

Our compiler extension, built on top of LLVM [21, 37], is principally used to identify clobber writes within transactions using dependency analysis. Due to aliasing, this identification requires significant reasoning about dependency chains. After this analysis, the compiler adds callbacks to both clobber writes and to memory accesses within transactions to automate logging and recovery.

Our runtime library manages the program during execution and recovery. It catches callbacks inserted by the compiler and programmer and directs the accesses to the appropriate logs. The runtime manages two logs: the `clobber_log`, which logs clobbered inputs, and the `v_log`, which logs volatile transaction inputs (inputs that reside either on the stack or in the volatile heap and would not be available during recovery. Usually these are function arguments).

The runtime will also initiate recovery of interrupted transactions after a crash. Recovery, for each transaction, proceeds by first restoring volatile state from the `v_log` and restoring clobbered state from the `clobber_log`, then reexecuting the interrupted transaction using the restored inputs.

To create a Clobber-NVM program, a developer needs to write transactions in a manner that meets our programming model, compile using our compiler with its extensions, and link to our runtime library. Recovering a Clobber-NVM program is done by restarting the program.

4.1 Using Clobber-NVM

Clobber-NVM's C programming model is designed to be easily used — most of its annotations and requirements have equivalents in Intel's PMDK library.

Figure 2(a) shows an example transaction written for Clobber-NVM, which executes a persistent list insertion (the equivalent PMDK C and C++ code are also shown). First, note the Clobber-NVM transaction is isolated within a function (line 1), termed the `txfunc`. Clobber-NVM's compiler instruments the call-site to collect the function name and its arguments within the `v_log`, and the function provides a convenient handle to initiate reexecution.

Upon entering the function, the appropriate locks are acquired on both persistent data (the list) and the volatile data (the new value). As with PMDK, transactions must be synchronized using conservative, strong strict two-phase locking for proper recovery [51]. We use the `txbegin` macro to start the actual transaction (line 4). As we intend to use a volatile non-local pointer within the transaction, we must record it using the `vlog_preserve` macro (line 5). With locks acquired and volatile inputs recorded, we can execute the transaction.

At the transaction's beginning, we allocate memory from NVM using the `pmalloc` interface (lines 6 and 7). Note that on line 12, we will change the list head to the new node, thereby clobbering a transaction input — this clobbered input will be identified automatically by the compiler, and then recorded by the runtime. Also note that Clobber-NVM does not require special macros when accessing persistent memory — the appropriate callbacks are added by the compiler. Assuming no power loss, the transaction will terminate and commit with the `txend` macro (line 13), and release its locks (line 14).

4.2 Runtime and Callbacks

Clobber-NVM's runtime manages the program's persistent state during execution and recovery. This task requires it to manage a few internal structures and to catch all necessary callbacks, added by both the user and the compiler.

The user is responsible for invoking four callbacks. The `txbegin` and `txend` macros inform the runtime of a starting or committing transaction, triggering a persistent update of the transaction's status. The `pmalloc` macros informs the runtime to allocate memory from NVM, instead of DRAM. As the application's semantics decide which sets of writes should happen failure-atomically and which memory allocation should allocate from NVM, the compiler is unable to assist with these callbacks. The `vlog_preserve` macro informs the runtime that some volatile non-local pointer input will be used during transaction execution — this input must be preserved persistently in the `v_log`. As compiler analysis is necessarily incomplete with respect to the transaction's read set, and as the appropriate `v_log` callbacks must occur at transaction begin, the developer must provide this information.

The compiler inserts other callbacks handled by the runtime. The first type of callback occurs at possible clobber write sites. This callback triggers a logging action in the runtime to record the clobbered inputs in its `clobber_log`. The second callback occurs at every memory access, and allows the runtime to appropriately swizzle pointers to support relocatable backing NVM storage. The final callback occurs on the top of `txfuncs` — this callback is used to record soon-to-begin transaction, collects the function name and its arguments.

Our runtime is implemented over Intel's PMDK v1.6 libpmemobj — it replaces the transaction, logging, allocation, and recovery management but preserves the user-facing interface for NVM region management and crash detection. In particular, Clobber-NVM's runtime manages two key logging systems: the `clobber_log`, which holds clobbered inputs, and the `v_log`, which holds both volatile inputs and tracks ongoing transaction state. Every ongoing transaction maintains one log `v_log` entry.

Our `clobber_log` is built over PMDK's undo log API. This design choice leaves Clobber-NVM's `clobber_log` very simple, and provides an additional benefit — as PMDK's performance improves, so does Clobber-NVM.

In contrast, the `v_log` is directly managed by the runtime. We manage the per-thread `v_log` using a global linked list resident in persistent memory, and allocate it on thread creation. The thread will use this log to manage its (at most one) active transaction. Using both the `vlog_preserve` macro and the compiler instrumented callback on entry into the `txfunc`, the log records the function arguments, function name and additional needed volatile data in the log at transaction begin. We use a single bit in each `v_log` to decide if re-execution is necessary on its corresponding thread. When the transaction begins, it is marked as ongoing, and the bit is cleared at commit.

4.3 Recovery

Clobber-NVM recovers NVM data to a consistent state via re-execution. We here describe the Clobber-NVM's recovery process.

```
void plist_ins(plist* lst, char* v,
 size_t vsz, lock* v_lk){
  lock(lst->lk); lock(v_lk);
  txbegin();
  vlog_preserve(v,vsz);
  pnode* n = pmalloc(sizeof(pnode));
  n->val = pmalloc(strlen(v));
  strcpy(n->val, v);
  n->nxt = lst->hd;
  // lst->hd is a clobbered input
  // and will be clobber logged
  lst->hd = n;
  txend();
  unlock(lst->lk); unlock(v_lk);
}
```

(a) Clobber-NVM

```
void plist_ins(TOID(plist) lst,
 char* v, lock* v_lk){
  lock(lst->lk); lock(v_lk);
  TX_BEGIN(pop){
    TOID(struct pnode) n =
      TX_NEW(struct pnode);
    D_RW(n)->val =
      TX_NEW(strlen(v));
    TX_ADD_FIELD(n, val);
    strcpy(D_RW(n)->val, v);
    TX_ADD_FIELD(n, nxt);
    D_RW(n)->nxt = D_RO(lst)->hd;
    TX_ADD_FIELD(lst, hd);
    D_RW(lst)->hd = D_RO(n);
  }TX_END {}
  unlock(lst->lk); unlock(v_lk);
}
```

(b) PMDK C

```
void plist::ins(char* v,
 lock* v_lk){
  auto p = pool_by_vptr(this);
  transaction::run(p,[this,v]{
    auto n =
      make_persistent<pnode>(v);
    strcpy(n->val, v);
    n->nxt = this->hd;
    this->hd = n;
  },this->lk,v_lk);
}
```

(c) PMDK C++

Figure 2: List insert operation using PMDK and Clobber-NVM

Figure 3(top row) shows a transaction progressing through normal execution. At transaction begin (top left), its inputs are already initialized and its outputs have not yet been touched. As the transaction executes (Figure 3, top center), it reads inputs, from both NVM and DRAM, and writes to output addresses to both NVM and DRAM. Note that some writes to output addresses may be clobber writes and will overwrite some inputs. During normal execution, the transaction will progress to completion and commit (Figure 3, top right), completely writing all outputs.

A transaction interrupted by a crash follows a different path. As with normal execution, the transaction starts with initialized inputs and untouched outputs (Figure 3, top left), then progresses through execution by writing to some output addresses (Figure 3, top center), including clobber writes. However, a power failure during execution drops the transaction to the recovery path.

After the power loss, the transaction loses all volatile memory and some NVM values that still resided in the machine's (volatile) caches (Figure 3, bottom left). At restart, Clobber-NVM first detects if there are any uncommitted, ongoing transactions using the per-thread v_logs. Using the transaction's logs, Clobber-NVM restores both the volatile and clobbered inputs, though the values at output addresses may still be inconsitent (Figure 3, bottom center).

With its inputs restored, the transaction is ready to restart (Figure 3, bottom right). Clobber-NVM recovers each thread independently — a valid strategy since our locking scheme ensures all ongoing transaction lock sets are disjoint. To recover a transaction, the corresponding txfunc is called, reexecuting the transaction from the beginning. Once the transaction executes past the point when the failure happened, the transaction has overwritten any incomplete values, erasing any inconsistencies caused by the power loss (Figure 3, top center). The transaction will continue to progress to completion and commit (Figure 3, top right).

4.4 Compiler

Clobber-NVM compiler identifies clobber writes and insert other utility callbacks.

The first compiler pass identifies writes within a transaction that may clobber an input, then instruments the clobber_log callback before the writes happen. This pass relies on classic alias analysis

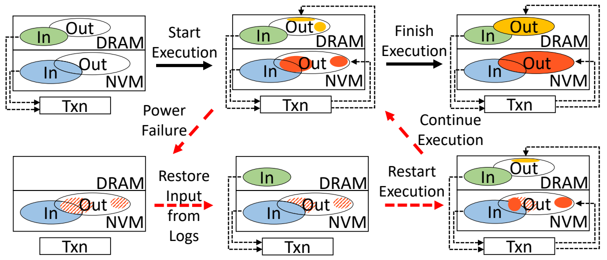

Figure 3: Recovery process of one transaction. The *In* and *Out* indicates input and output addresses within both DRAM and NVM.

to identify clobber writes. However, this basic alias analysis is not necessarily precise and may over identify clobber writes, though this is a performance, not a safety issue. The pass subsequently refines the result through novel analysis propagation.

Then the second pass adds callbacks to all memory accesses: these callbacks allow the system to intercept accesses to NVM and, as necessary, swizzle the pointers to redirect the accesses to the relocatable backing region. Finally, the third pass instruments the code for recovery: it adds v_log callbacks to the txfunc to record their names and arguments.

Clobber-NVM's compiler is built on top of LLVM [21, 37]. We use clang [17] as the frontend compiler to translates C/C++ code to LLVM IR, and introduce the three passes [20] described above to the LLVM compiler toolchain. All passes operate on LLVM IR [19].

In the remainder of this subsection, we describe our clobber write identifying pass in detail, starting with the conservative implementation, then describing our iterative refinement.

Conservative Clobber Writes Identification Clobber-NVM's conservative clobber write identification follows two steps. The first step identifies *candidate input reads*, that is, reads that could conceivably be the first operation on a value. The second step uses these reads to find *candidate clobber writes*, that is, writes that could conceivably overwrite an input. Note that both steps are conservative — all possible input reads and all possible clobber writes will be candidates.

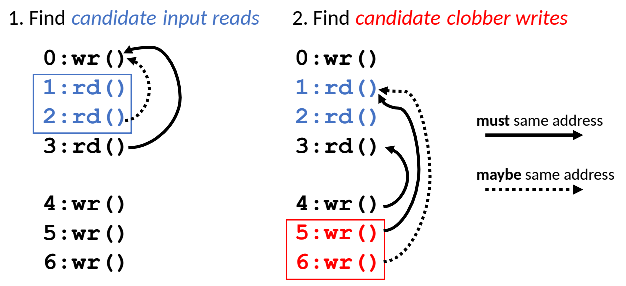

Figure 4: Conservative candidate clobber write identification

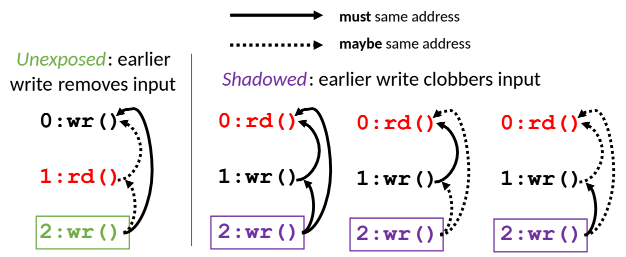

Figure 5: Removing false candidate clobber writes

The identification pass relies on LLVM alias analysis [18] to identify candidate input reads and clobber writes. Alias analysis produces pair-wise results that indicate two memory accesses (1) cannot, (2) may or (3) must point to the same location.

Using alias analysis, the first step traverses all reads in the transaction, searching for candidate input reads. Reads that are *dominated* by an earlier write (all paths to the read first execute the write), and whose dominating write must modify the same address, cannot be candidate input reads. All other reads are labeled as a candidate input read. Figure 4(left) shows this process.

In the second step, the compiler identifies candidate clobber writes for each candidate input read. Candidate clobber writes include all *successor* writes (writes that may be executed after the input read) that may write to the same address as the input read. Figure 4(right) shows this identification.

At this point, the compiler has conservatively identified all clobber writes, but some candidates may never overwrite an input. Figure 4(right) shows an example of such a candidate clobber write on line 6: the input will already be clobbered on line 5.

Dependency Analysis Propagation To reduce Clobber-NVM's logging cost, we perform additional dependency analysis propagation to remove candidate clobber writes that, upon further analysis, can never be true clobber writes. We target two general types of *false clobber candidates*.

The first type of false clobber candidate we term *unexposed*. This false candidate may indeed be the first write to overwrite a candidate input, but, if it does — it is provable the input candidate is a false input. The scenario is shown in Figure 5(left). In this scenario, the candidate input read is dominated by some earlier write. Since the earlier write and the read may not access the same location, the read is considered a candidate input. The subsequent write may also access the read's location, and so it is considered to be a clobber candidate. However, both writes are guaranteed to access the same location through alias analysis — as a consequence, if the later write overwrites the read, the read cannot be an input (it will be dominated by the earlier write).

The second type of false clobber write candidate is a candidate which is *shadowed* by some earlier clobber write. A shadowed candidate may indeed overwrite an input, but if it does, it is guaranteed that some earlier write already clobbered it. This relationship requires two conditions. First, it requires some earlier write to dominates the shadowed write. Secondly, the alias relationship between the input read, earlier write, and shadowed write must ensure that if the shadowed write does overwrite the input, the earlier write will have first. There are three alias combinations between the three accesses that meet these criteria: they are shown in Figure 5(right). In practice, this kind of false candidate occurs often in loops: the first iteration clobbers the input, but subsequent iterations do not need to log.

Our analysis searches for both types of false clobber candidates by iterating over every pair of a candidate clobber write and its corresponding input read, and then looking for an additional write that would then form one of these four cases.

5 EVALUATION

In this section, we evaluate Clobber-NVM's performance to provide answers to the following questions:

- How much improvement does Clobber-NVM provide for persistent data structures compared to other libraries?
- What is the reason for Clobber-NVM's high performance?
- What is Clobber-NVM's recovery overhead compared to PMDK?
- What is Clobber-NVM's performance on application-level code?
- How does the underlying data structure of an application affect its performance when build with Clobber-NVM?
- What is the effectiveness of Clobber-NVM compiler optimization?
- How much longer does it take for an application to compile with Clobber-NVM, compared to Clang?

Our experimental workloads include four data structure benchmarks and three recoverable applications.

5.1 Evaluation Setup

We compare against three popular NVMM libraries with Clobber-NVM.

PMDK [48] is Intel's failure atomicity library. Its later versions use hybrid undo-redo logging techniques [30]. The technique is based on a combination of undo logging for modify a group of memory atomically and redo logging for memory allocation and deallocation [52]. Based on our experiments, PMDK v1.6 generally provides the best performance among all available versions — we show this version in all experiments.

Atlas [5] is another undo-log system that allows for complicated locking schemes within failure-atomic regions. It uses lock operations to infer failure-atomic operation boundaries. Due to its weak concurrency requirements, Atlas tracks dependencies between failure atomic operations and is prepared to rollback even completed operations — this dependency tracking incurs signficant runtime cost [33].

Mnemosyne [55] is a redo-log based system. Unlike the other systems, it uses the C++ transactional memory model to parallelize code, instead of using locks.

Prior recovery-via-resumption systems (JUSTDO [33] and iDO [42]) do not have publically available implementations. To compare Clobber-NVM's performance with their's, we implemented a compiler instrumentation pass to collect iDO's transaction information.

We run the benchmarks on a platform with two 24-core Intel Cascade Lake SP processors, running at 2.2 GHz. The platform has a total of 192 GB of DRAM and 1.5 TB (6 ×256 GB) of Intel Optane DC Persistent Memory directly attached to each processor [32]. We configured our test machine such that Optane DCPMM is in 100% App Direct mode [1]. In this mode, software has direct byte-addressable access to the Optane DCPMM. All experiments use `Ext4` to manage persistent pools and directly access NVM pages via DAX [40].

5.2 Data Structure Benchmarks

In our first experiment, we compare Clobber-NVM's throughput with comparison systems on data structure benchmarks:

B+ Tree uses reader-writer locks at the granularity of individual nodes, stores keys in the internal nodes, and adds both the key and the value to the leaf nodes.

HashMap is adapted from the PMDK repository [31]. We create 256 instances of the HashMap, treat each one as a bucket, and protect each bucket with a reader-writer lock.

Skiplist is a skiplist with 32 levels. We use a single global lock for the entire data structure.

Red-Black Tree is implemented in accordance with the version in Linux kernel. We use a global reader-writer lock for the tree.

On all data structures except B+ Tree, we insert key-value pairs with key size 8 bytes and value size 256 bytes. On B+ Tree, the inserted key size is 32 bytes. The benchmark runs YCSB [12] workloads against different versions of the data structures. It populates each structure with 1 million entries (YCSB-Load workload).

We make the following observations. Firstly, for single thread, because undo-log entries are immediately followed by `fence` instructions, the number of log entries imposes more latency on Clobber-NVM and undo-log systems compared to redo-log systems like Mnemosyne. Hashmap insertion is relatively simple. So Clobber-NVM shows over 2.13× performance of Mnemosyne on hashmap benchmark. B+ Tree has the longest transaction. Mnemosyne can provide performance comparable to Clobber-NVM, despite it actually logs much more data.

Secondly, Clobber-NVM always outperforms undo log systems significantly on one thread. PMDK undo log shares the same logging subsystem with Clobber-NVM `clobber_log`. But it usually does significantly more undo log entries. Therefore, Clobber-NVM shows 1.82× of its performance on average, and can perform up to 2× compare to it on hashmap and skiplist. Atlas has to track dependencies between transactions, which imposes significant runtime overhead. On average, Clobber-NVM provides 4.3× of Atlas performance.

Thirdly, on multithread workloads, scalablility of Clobber-NVM and other lock-based system is mostly determined by the locking scheme. It is the programmer's responsibility to use finer granularity locks, in order to achieve good scalablility. For example, Clobber-NVM shows the best scalablility on B+ Tree among all data structures, since it uses per node granularity locks. Clobber-NVM provides 1.8× of Mnemosyne performance, 6.6× of Atlas performance, and 1.9× of PMDK performance. Since PMDK and Clobber-NVM rely on the same underlying lock scheme, they scale similarly across all data structures. Clobber-NVM outperforms PMDK by over 1.9× across all data structures with 24 threads. On data structures with a single global lock, Mnemosyne scales better than Clobber-NVM and PMDK, result in matching Clobber-NVM performance on rbtree and skiplist with 24 threads.

5.3 Performance Breakdown

In this experiment, using the same data structure benchmarks, we incrementally enable different logs in Clobber-NVM and PMDK (full undo log) to understand where the performance costs of Clobber-NVM reside.

No-log is the baseline performance, where the data structures do not do any logging. **Clobber-NVM-vlog** only does `v_log` in Clobber-NVM. **Clobber-NVM-clobberlog** is Clobber-NVM that only enables `clobber_log`. These three systems are not failure-atomic. **Clobber-NVM-full** represents the full version of Clobber-NVM. It does both `v_log` and `clobber_log`. **PMDK** shows PMDK's performance. It does full undo log for each transaction.

Here, we use one thread to insert key-value pairs from YCSB benchmark. Figure 7 shows the measured logging overhead on different data structure. In our evaluated workloads, Clobber-NVM-vlog always have one log entry per transaction. Clobber-NVM-clobberlog typically use 15.8% to 39.5% as many log entries as PMDK per transaction, Clobber-NVM, in total, uses 21.5% to 42.3% as many log entries as PMDK. Regarding the log size, PMDK requires 16.7× to 154.5× more bytes compared to Clobber-NVM-clobberlog. PMDK's log size is 1.2× to 58× of Clobber-NVM-vlog's log size, and 1.1× to 42.6× of Clobber-NVM's log size, depending on the data structure.

The comparison between Clobber-NVM-vlog and PMDK shows that `v_log` is very cheap due to its implementation. On all data structures, a great portion of log bytes are used in `v_log` (more than 70%), but Clobber-NVM-vlog's performance is comparable to No-log on most data structures. The high performance of `v_log` comes from two perspective — The `v_log` entry count is always one for the whole transaction, result in only two necessary `fences`. And the pre-allocated `v_log` buffer made it much faster compared to traditional undo log entry.

Most of Clobber-NVM's overhead is imposed by `clobber_log`. Generally speaking, fewer log entries and smaller log size result in better performance. And log entry count usually matters more than log size, which is consistent with the fact that a `fence` is usually more expensive than a `flush`.

The only exception is the hashmap. Its Clobber-NVM-vlog version performs 10% slower than its Clobber-NVM-clobberlog version, indicating that most of Clobber-NVM's overhead is caused by `v_log` on the hashmap insertion benchmark. On this benchmark, its `clobber_log` log count is one, and its log size is 8 bytes.

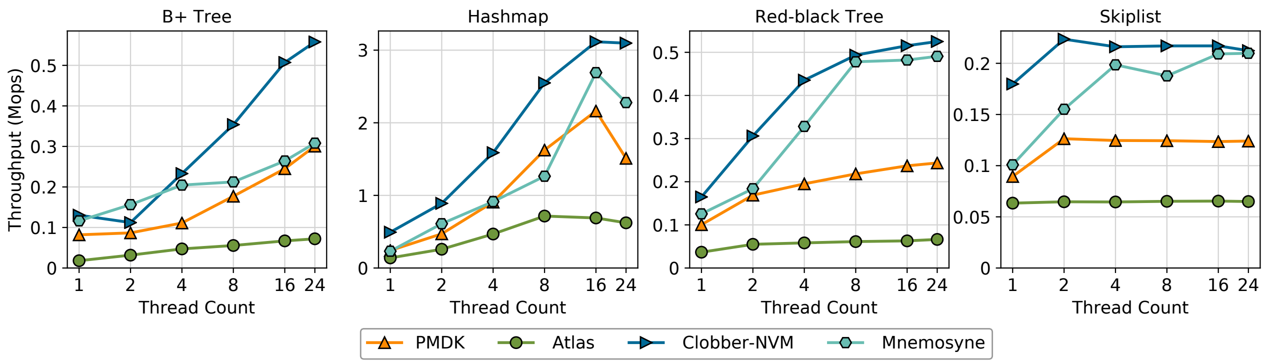

Figure 6: Measuring the throughput of different NVM libraries: each data structure is scaled up to 24 thread. Clobber-NVM shows better performance over other libraries on all data structures

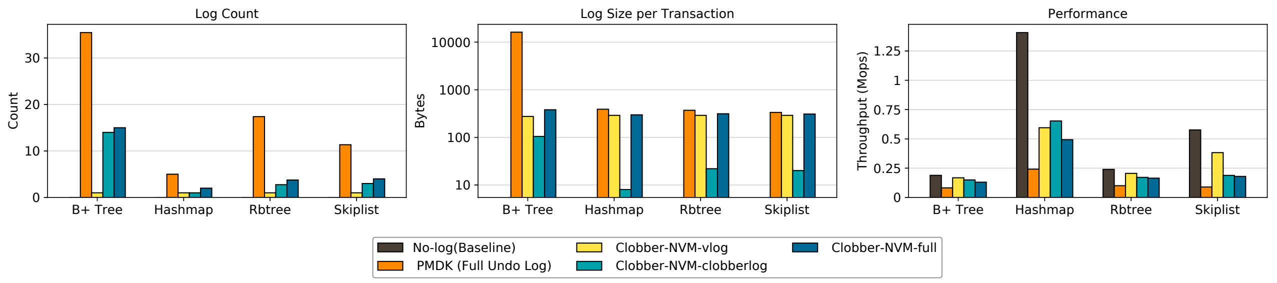

Figure 7: Measuring the overhead of different logging strategies: different log count and log size result in different performance. The operated key-value pair has key size 8 bytes (32 bytes for B+ Tree), and value size 256 bytes.

Because the v_log log size is much larger than the clobber_log log size, v_log component causes more latency than clobber_log component on the hashmap insertion.

5.4 Comparison to iDO

iDO is a state-of-the-art recovery-via-resumption library. iDO's compiler pass breaks transactions into a series of idempotent regions, logging at the boundary between idempotent regions — failure during an iDO transaction triggers the reexecution of the idempotent code region, followed by the resumption of the remainder of the interrupted transaction. Because its code is not publically available, we reimplemented a compiler instrumentation pass to instrument the code, and collect transaction information as iDO would have.

iDO will always have the at least as many bytes persisted per transaction as Clobber-NVM. iDO's strategy of logging at the boundary of idempotent code regions requires larger log entries than Clobber-NVM — the log entries at each boundary require a snapshot of most registers, plus a flush and fence for any modified memory location. The strategy generally also incurs more logging calls than Clobber-NVM's logging of clobber writes. Clobber-NVM only needs to log when a write clobbers a transaction input, whereas iDO logging needs to log whenever a write clobbers an idempotent region's input (all transaction inputs are included in some region's input, but not all region inputs are transaction inputs as they may be intermediate state local to the transaction). Additionally, iDO places the program stack in persistent memory to avoid the need of

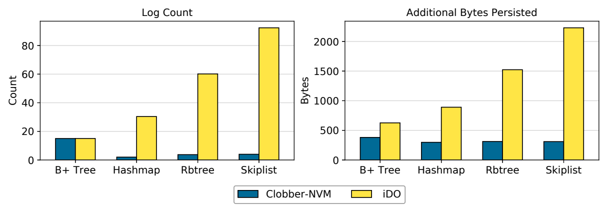

Figure 8: Clobber-NVM and iDO log size per transaction

manually copying stack variables into the persistent heap on FASE initialization [42] (the purpose of our v_log), but consequently needs to track accesses to stack variables during transaction execution and log them if necessary.

As shown in Figure 8, iDO not only requires more logging points, it also persists significantly more data. It logs 1× to 23× more frequently compared to Clobber-NVM, depending on the specific data structures and workload. On average, iDO logs 4.2× more bytes than Clobber-NVM, and it logs up to 7.2× more bytes on skiplist benchmark.

5.5 Recovery Overhead

In this experiment, we compare Clobber-NVM's recovery overhead with PMDK's recovery overhead on the same four data structures. Clobber-NVM's recovery process is composed by three steps. Firstly, it opens the persistent pool. Secondly, it applies the clobber_log entries to their corresponding addresses. Lastly, it reads v_log and re-executes the interrupted transactions based on valid v_log

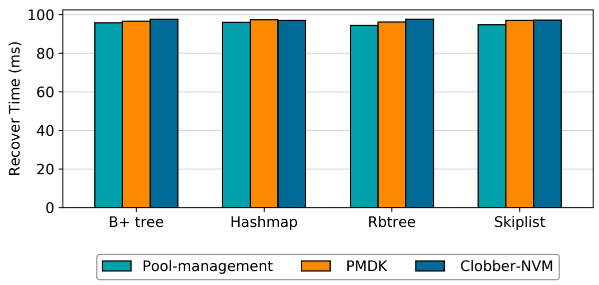

Figure 9: Recovery overhead on different data structure

entries. PMDK share the first two steps with Clobber-NVM, but instead of applying `clobber_log` on selected addresses, it reads undo log entries of uncommitted transactions, and rolls back the transaction by rewriting the values in undo log to all pmem variables that were updated in the transactions.

We randomly crash the data structure benchmarks, and measure the average recovery overhead. As shown in Figure 9, the recovery latency of Clobber-NVM and PMDK are similar. Most of their recovery latency is spent on pool managements. PMDK shows marginally lower recovery overhead on bptree, rbtree and skiplist, but Clobber-NVM recovers slightly faster on hashmap. When most of writes in a interrupted transaction are not clobber writes, Clobber-NVM would have less and shorter `clobber_log` entries recorded. And if the latency of re-execution on the interrupted transaction is small, Clobber-NVM is likely to have lower recovery overhead.

5.6 Memcached

Memcached [44] is a production-quality key-value store. It is already integrated with Mnemosyne [45], and we modified its volatile version to use both PMDK and Clobber-NVM. Memcached is consists of a server side and a client side. We used the tool memslap [38] as the client to generate a stream of Memcached requests according to a desired distribution. We used 4 client threads, which generated requests with uniformly distributed 16-byte keys and 64-byte values.

We experimented with 4 types of workloads: insertion-intensive (95% insertion / 5% search), insertion-most (75% insertion / 25% search), search-most (25% insertion / 75% search), and search-intensive (5% insertion / 95% search). As shown in Figure 10, Clobber-NVM outperforms PMDK and Menmosyne on all workloads. Clobber-NVM's `clobber_log` entry count is always smaller than redo/undo log entry count. And the per transaction `v_log` is less expensive. It outperforms PMDK and Menmosyne more on more insert intensive workloads, because PMDK and Menmosyne require more log entries. On more search intensive workloads, more operations do not involve logging mechanisms. Clobber-NVM's performance gain is smaller. But the longer read path of redo-log based system result in lower performance of Menmosyne compared to both Clobber-NVM and PMDK.

Clobber-NVM provides up to 2.5× of Menmosyne's throughput and 1.8× of PMDK's throughput on single thread workloads. However, older versions of memcached were notorious for exhibiting poor scaling due to coarse-grain locking [16, 42]. Therefore, we replace the exclusive lock in its original code with spinlock and reader-writer lock. As expected, spinlock works better for insert-intensive workloads, and reader-writer lock provides better scalablility for search intensive workloads.

5.7 Vacation

We also evaluated the STAMP benchmark suite's vacation application [8] performance with Clobber-NVM, PMDK and Mnemosyne. Vacation simulates the transactions of a travel agency, and transactions span several tables simulating travel booking reservations.

Vacation is consists of four tables. Similar to prior implementations [28] [55], we persist the tables in persistent memory, and leave the client threads in volatile memory. The tables are originally implemented on red-black trees. Here, we also replace it with an AVLtree implemented in the STAMP suite [8] to show the applications performance on a different underlying data structure. The database has 100000 records of each reservation item. The workload is consisted by 99% of item reservation of cancellation, and the rest create or destroy items. We adjust the number of queries per task to create different workloads. Again, we use **No-log** as the baseline.

Figure 11 shows that No-log, PMDK, and Clobber-NVM performs 17%, 9% and 7% better on avltree version compared to the red-black tree version, indicating that the undo log entries (and `clobber_log` entries) are data structure dependent, but the `v_log` entries in vacation are the same across different underlying data structures.

Because the number of queries per task indicate the propotion of read in one transaction, the logging overhead of PMDK and Clobber-NVM decreases when the number of queries per task increases. When the query number is six, PMDK and Clobber-NVM's overhead is 74% and 68%, respectively. We also find that, `v_log` size increases as the read propotion increases. Therefore, Clobber-NVM outperforms PMDK more with less number of queries per task. Since Mnemosyne is a redo-log based system, its logging overhead increases as the number of queries per task increases. Its overhead compared to the No-log baseline increases from 176% to 200%.

5.8 Yada

Yada, also from the STAMP suite [8], is a volatile mesh refinement application. It implements Ruppert's algorithm for Delaunay mesh refinement [50]. The input mesh is refined so that it has a certain minimum angle. We use the data files *ttimeu10000.2* provided in the STAMP suite as input, which is consisted of 19998 elements. Here, we compare Clobber-NVM performance with PMDK performance.

Again, **No-log** is the baseline performance in which the application is running without any logging mechanisms. We persist the graph that stores all the mesh triangles, the set that contains the mesh boundary segments, and the task queue that holds the triangles that need to be refined. We set the angle constraining from 15 degrees to 30 degrees, and show each version of Yada performances.

Figure 12 shows that, on all angle constraints, PMDK imposes about 42% overhead compared to No-log version. Clobber-NVM reduces the overhead to about 27%. Because Yada is more compute intensive compared to key-value stores. The logging overhead on Yada is low. Therefore, the potential optimization space of Clobber-NVM is small.

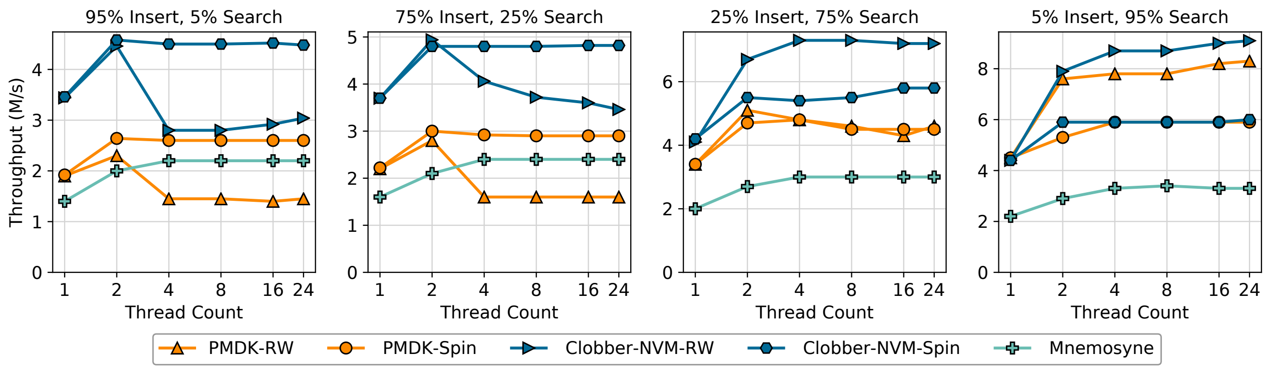

Figure 10: Memcached Performance on Different Workloads and Threads

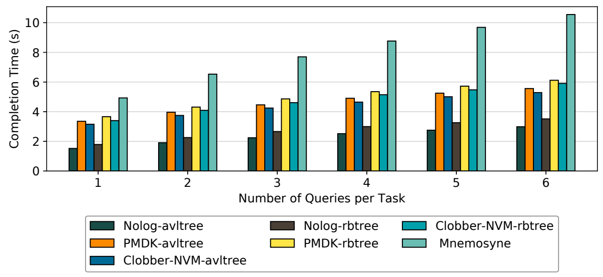

Figure 11: Vacation performance on different data structure

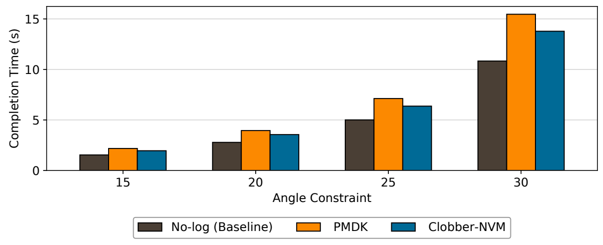

Figure 12: Yada Performance on Different Angle Constraining

5.9 Optimization Effectiveness

Clobber-NVM compiler passes identify potential clobber writes. In order to reduce the overhead introduced by conservative identification, Clobber-NVM performs additional dependency analysis propagation to remove false clobber candidates, as introduced in Section 4.4. We show performance improvement by avoiding redundant `clobber_log` on the four data structures and three applications in Figure 13.

On the four data structures, skiplist shows the most performance improvement of up to 15%. We find that the compiler pass removes two clobber candidates out of five, end up requiring only three `clobber_log` entries per transactions. On memcached workloads, the one consisted by 95% insert and 5% search requests improves the most, to 15%. As can be expected, the avoided `clobber_log` entries are on the write request path. The unoptimized version incurs up to 32% more `clobber_log` entries and 47% more bytes. Among the three STAMP applications, Yada shows the most performance improvement. It improves its performance by 2.4%, and reduces its `clobber_log` frequency by 36%.

The effectiveness of compiler optimization pass is application and workload dependent, we expect one application to benefit from the optimizations when:

- Its transaction is long, and is consisted mostly by writes.
- Its memory accesses follows certain patterns. For example, it updates and reads an address at each iteration of a loop.
- Its latency is mainly imposed by the logging.

5.10 Compile Time Overhead

Clobber-NVM relies on compiler analysis and instrumentation. In this experiment, we show the compile time overhead of Clobber-NVM. We compare the compilation latency of Clobber-NVM with Clang-7.0.0. Figure 14 shows the compile latency of four data structures and three applications.

The compile overhead is similar among four data structures, Clobber-NVM adds 29% latency on average. Clobber-NVM takes 55% longer than Clang on memcached. The higher compilation latency is because we compile all files of memcached projects with Clobber-NVM compiler, while only compile the files that has pmem accesses in data structure benchmarks. With more accurate identifications of memcached pmem accesses, the compilation latency is expected to be lower. The STAMP applications also show higher compilation overhead. Since in these applications, pmem accesses are spread across relatively more files, the code analysis and instrumentation also takes longer.

6 RELATED WORK

Researchers have proposed many NVM-optimized data structures [6, 11, 22, 47, 54, 60], and the architecture community has been working on better hardware support for failure-atomicity [3, 14, 24, 34, 35, 53]. Recently, the research community has also focused on using persistent memory for specific applications [7, 39]. For example, researchers [39] proposed creating failure atomic GPU kernels by leveraging idempotence to reduce logging cost, taking advantage of the necessary copies used for GPU kernel execution. However, high efficiency general purpose libraries based on commodity hardware is still an open research area. Undo log approach includes NV-heaps [10], Atlas [5], and PMDK-v1.4 [48]. Mnemosyne [55] SoftWrAP [23] and NVthread [10] relies on redo logs. Undo-logging

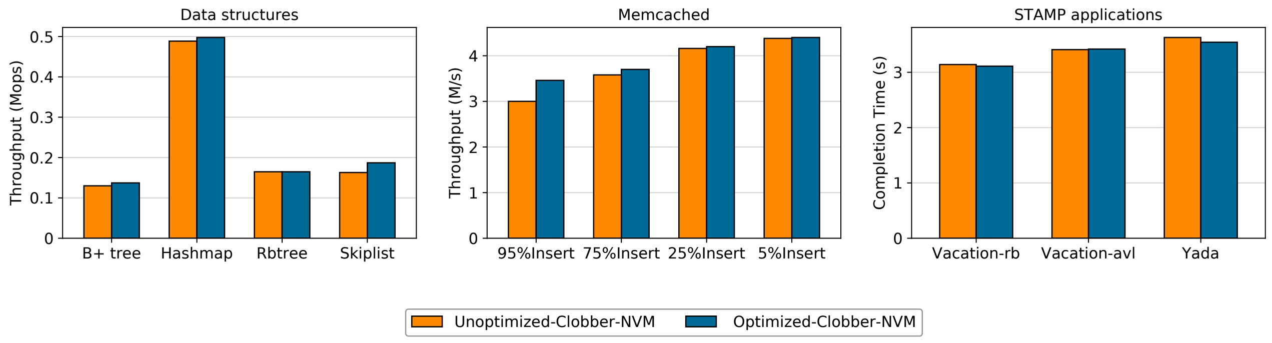

Figure 13: Optimization effectiveness on data structures and applications

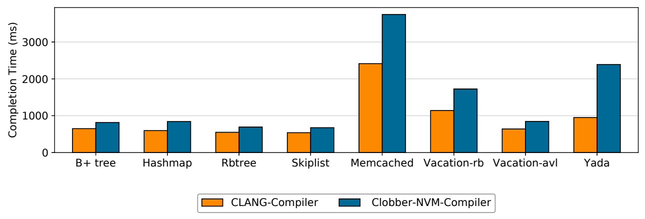

Figure 14: Compile latency on data structures and applications

systems usually requires expensive ordering fences at least proportional to the number of contiguous data ranges modified in each transaction. In contrast, redo-logging implementations generally require fewer ordering fences regardless of the transaction size. However, their load interposition and load redirection to updated NVM addresses slows down read speed and increases system complexity.

A number of systems were proposed to optimize redo/undo systems by maintaining a shadow copy of working set during runtime. Kamino-Tx [43] relies on dual NVM copies to achieve memory persistence. Romulus [13] uses a volatile redo-log with a shadow copy in NVM. Both DudeTM [41] and NV-HTM [4] uses a persistent redo-log with shadow copy stored in DRAM. PMThread [58] maintains a DRAM copy and an additional NVM copy. All of these approaches at least double the memory consumption of the application as at least two copies of the data are maintained. Clobber-NVM only have log entries during an update operation, the additional space overhead will usually be even much smaller than conventional redo and undo log systems.

Currently, NVM failure-atomicity systems have operation semantics that rely on either lock-inferred failure atomic sections (FASEs) [5, 29, 33, 42], classical transactions [4, 41, 43, 55] or programmer delineated transaction boundary with a proper lock scheme [23, 48]. Several recent works optimize scalability by relaxing the traditional ACID semantics. Pisces [26] exploits snapshot isolation on persistent memory, and TimeStone [36] provides three isolation levels to user and achieves higher concurrency on more relaxed isolation model. Compared to these libraries, we target applications that require full ACID semantics.

A recent line of works provide better performance by only ensure periodic persistence [47] [11] [58]. By periodic persistence, their consistency guarantees is made at per-epoch granularity, as oppose to per failure-atomic operation in Clobber-NVM and most other systems [5, 42, 48, 55]. In these periodic persistence systems, after failure, persistent data will be recovered to the state of the last completed epoch.

In JUSTDO [33] and iDO [42] logging, the system recovers by resuming execution of the interrupted failure-atomic section. JUSTDO logs and persists the program counter, the to-be-updated address, and the value to be written before each `store` happens. Because of the high cost to operate on conventional machines, JUSTDO assumes it will work on a machine with persistent cache. Its successor, iDO logging, avoids logging before individual stores by using compiler-support to identify idempotent regions and instruments adds logging at their boundaries (almost all idempotent regions contain fewer than 4 writes). iDO logs and persists program state - registers, live stack variables, and the program counter. Clobber-NVM is different from JUSTDO and iDO primarily in its use of clobber logging, which restarts the entire transaction from the beginning, instead of at an intermediate logging point. Therefore, Clobber-NVM logging overhead is, in general, significantly lower than both these systems, as their logged state at each logging point is much larger than Clobber-NVM's, and they always require more logging points [42]. Clobber-NVM also supports volatile data usage in transaction, which is generally not supported by these systems. In JUSTDO, use of volatile data and cache values in registers are forbidden during transaction execution, and iDO does not allow volatile heap usage during a failure atomic section while maintaining the stack in NVM to reduce logging cost.

7 CONCLUSION

This paper describes Clobber-NVM, a system that recovers by re-executing interrupted transactions. Clobber-NVM leverages compiler analysis to identify necessary log entries, and automatically adds logging for selected variables — clobber input addresses and non-local volatile variables — at compile time. At recovery, interrupted transactions roll back to the prior-execution state by applying the `clobber_log` and `v_log`, then roll forward to a consistent after-execution state. Compared with existing NVM user-level libraries, Clobber-NVM simplifies the logging system and reduces runtime overhead. Our evaluation shows that Clobber-NVM significantly improves performance compared to state-of-the-art failure-atomic libraries.

ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We would like to thank our shepherd Milind Kulkarni and the anonymous reviewers for their insightful feedback. We are also thankful to Intel Corporation for hardware access.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact description provides information to build Clobber-NVM and run its evaluations. It includes four data structures and three applications, as evaluated in Section 5. In this appendix, we first describe the the hardware/software requirements for building and running the experiments. Next, we introduce the datasets used in evaluating Clobber-NVM, and then outline the necessary steps to run the experiments from Section 5. Finally, we explain how to read the evaluation results. Note that the artifacts do not depend on NVMM to enable functional reproduction. However, our evaluations were performed on the specific hardware detailed in Section 5.1. We expect other hardware may introduce different results.

A.2 Artifact Check-List (Meta-information)

- **Algorithm:** Clobber log and `v_log`.
- **Program:** Clobber-NVM's library (compiler and runtimes), four data structures (Clobber-NVM, PMDK, Atlas and Mnemosyne versions), Memcached v-1.2.5 (Clobber-NVM and PMDK versions), Memcached v-1.2.4 (Mnemosyne version), vacation (Clobber-NVM, PMDK and Mnemosyne versions), yada (Clobber-NVM and PMDK versions).
- **Compilation:** GNU C/C++ and LLVM Clang compilers.
- **Data set:** Traces from YCSB, Mnemosyne and STAMP.
- **Runtime environment:** See Section 5.1 for details.
- **Hardware:** The evaluation results can be reproduced by running the experiments on a machine equipped with at least 24 physical cores per socket and 32 GB of memory to run all experiments.
- **Execution:** See A.4 and A.5 for details.
- **Metrics:** Performance of data structures, Memcached, vacation and yada. Performance breakdown of data structures.
- **Output:** Performance of data structures, Memcached, vacation and yada. Performance breakdown of data structures.
- **How much time is needed to prepare workflow (approximately)?:** The experiments are ready to run in about 40 minutes.
- **How much time is needed to complete experiments (approximately)?:** About 6 hours to run all the experiments.
- **Publicly available?:** Code, datasets, tools, and benchmarks are publicly available.
- **Archived (provide DOI)?:** 10.5281/zenodo.4322233

A.3 Description

A.3.1 How to Access. The artifacts are publicly available through Zenodo archival repository. You can access the code by using its DOI.

A.3.2 Hardware Dependencies. We have evaluated Clobber-NVM's performance using the testbed from Section 5.1. The evaluations, however, only require 24 physical cores per socket and 32 GB of NVMM. In absence of access to real NVMM (e.g., Intel Optane DC), you need to reserve 32 GB of memory to emulate NVMM (see https://pmem.io/2016/02/22/pm-emulation.html for instructions).

A.3.3 Software Dependencies. We have evaluated Clobber-NVM on Ubuntu 18.04, with GNU 7.3.1, and LLVM 7.0.0. Run `./deps.sh` to install main dependencies. You may need `root` permission to install some libraries. Run `sudo mnemosyne-gcc/usermode/library/pmalloc/include/alps/install-dep` to install Mnemosyne dependencies. You can also install them manually.

- PMDK and its dependencies: autoconf, pkg-config, libndctl-dev, libdaxctl-dev, libjemalloc-dev
- Atlas and its dependencies: LLVM clang-3.9, ruby, libboost-graph-dev
- Mnemosyne and its dependency: scons
- Memcached and its dependencies: libevent-dev, memslap driver
- jemalloc, autogen, numactl, libconfig-dev, libelf-dev.
- CMake, build-essential, uuid-dev, libz-dev

A.3.4 Data Sets. Clobber-NVM's performance tests use traces from YCSB (workload: Load) to measure throughput of benchmark data structures. It uses the publicly available `memslap` as the client to generate a stream of Memcached requests. It also uses the workloads from STAMP suites. The vacation benchmark generates its input datasets randomly. The yada benchmark uses a sample input. The data sets are either publicly available or can be generated by publicly available code. They are all included in the artifact.

A.4 Installation

Clobber-NVM is a joint compiler/runtime library. It uses `make` for the compilation of both the compiler and runtime components.

Use the build script `build.sh` to build the compiler component of Clobber-NVM using the GNU C/C++ compiler, and the runtime/library component of Clobber-NVM using LLVM Clang compiler. You will need `root` permission at this step. Check the `build.sh` script for instructions on building a specific component.

```
$ cd Clobber-NVM
$ ./build.sh
```

Also, make sure to have a NVMM file-system mounted at `/mnt/ram`.

A.5 Experiment Workflow

There are two ways to run experiments. You can choose to run the experiments altogether by running the script `./run_all.sh`, or customize and run individual experiments, as suggested in the `README` files. There are separate scripts to run benchmarks in each category.

A.5.1 Running Experiments Altogether. Run the following commands run the experiments of Clobber-NVM altogether.

```
$ cd Clobber-NVM
$ ./run_all.sh
```

The script dumps the evaluation results in `Clobber-NVM/fig*.csv`.

A.5.2 Running Individual Experiments. You can also configure and run experiments individually, following the instructions in the `Clobber-NVM/README.md` and `README` files under specific directories in the source repository.

A.6 Evaluation and Expected Result

Once you've ran the experiments as suggested above, you can compare the outcome with the expected results. The key results, as shown in Section 5, is obtained on the machine with the specific hardware configurations. Our experiments should be able to successfully run benchmarks on machines without real NVMM equipped. But the performance numbers collected on such machines are expected to be different, given different underlying hardware. Both PMDK and Clobber-NVM included are compatible with newer PMDK versions. But the performance number are also expected to change, given potentially different memory allocation and undo log implementations.

The results are either reported in the `Clobber-NVM/fig*.csv` files or printed to the screen. You can also check the redirected local files for the screen output (see `README`s).

For the csv files, most numbers/parameters are self-explaining. Check the scripts for detailed explanation if meanings of the numbers cannot be determined. For instance, below is the output for running skiplist with Clobber-NVM (Figure 6, reported in `Clobber-NVM/fig6.csv`), where the data size is 256 bytes, the thread count is 1 thread and the average throughput across 5 runs is 181 Kops/sec.

```
clobber, skiplist, 1, 0, 256, 183226
clobber, skiplist, 1, 2, 256, 178963
clobber, skiplist, 1, 3, 256, 179784
clobber, skiplist, 1, 4, 256, 182346
clobber, skiplist, 1, 5, 256, 180285
```

For the results printed to the screen, most of their parameters are also printed. For example, below is the output for running Yada with Clobber-NVM (Figure 12). The angle constraint is 15 degree, and the completion time is 1.538s.

```
Angle constraint = 15.000000
Reading input... done.
Initial number of mesh elements = 19998
Initial number of bad elements = 2931
Starting triangulation... done.
Elapsed time = 1.538
Final mesh size = 30158
Number of elements processed = 5062
Final mesh is valid.
```

A.7 Experiment Customization

Refer to the documentation under the benchmark directory in the code repository for details on configuring the benchmarks.

A.8 Notes

The documentation (i.e., `README` files) that accompanies the source code contains additional information for using the code as well as further instructions on setting up and running the benchmarks.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Alper Ilkbahar. 2018. Intel Optane DC Persistent Memory Operating Modes Explained. https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/

[2] Mohammad Alshboul, Hussein Elnawawy, Reem Elkhouly, Keiji Kimura, James Tuck, and Yan Solihin. 2019. Efficient Checkpointing with Recompute Scheme for Non-Volatile Main Memory. *ACM Trans. Archit. Code Optim.* 16, 2, Article 18 (May 2019), 27 pages. https://doi.org/10.1145/3323091

[3] Miao Cai, Chance C Coats, and Jian Huang. 2020. HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 584–596.

[4] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 368–377.

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[6] 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

[7] Sui Chen, Faen Zhang, Lei Liu, and Lu Peng. 2019. Efficient GPU NVRAM persistence with helper warps. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[8] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*. 35–46.

[9] Dave Chinner. 2015. xfs: updates for 4.2-rc1. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[11] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/3210377.3210392

[14] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1173–1186. https://doi.org/10.1145/3373376.3378481

[15] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/2254064.2254120

[16] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.* 1, 2, Article 13 (Feb. 2015), 42 pages. https://doi.org/10.1145/2686884

[17] The LLVM Foundation. 2019. Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org/

[18] The LLVM Foundation. 2019. LLVM: Basic Alias Analysis. https://llvm.org/docs/Passes.html#basic-aa-basic-alias-analysis-stateless-aa-impl

[19] The LLVM Foundation. 2019. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html

[20] The LLVM Foundation. 2019. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#memcpyopt-memcpy-optimization

[21] The LLVM Foundation. 2019. The LLVM Compiler Infrastructure. https://llvm.org/

[22] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

[23] Ellis R Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[24] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 652–665.

[25] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[26] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 913–928.

[27] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)* 15, 4 (1983), 287–317.

[28] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/3373376.3378472

[29] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[30] Intel Corporation. 2017. Pmdk issues: introduce hybrid transactions. https://github.com/pmem/pmdk/pull/2716

[31] Intel Corporation. 2019. Examples for libpmemobj: A Transactional HashMap. https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/hashmap

[32] Intel Corporation. 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[34] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[35] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 58, 13 pages.

[36] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 335–349. https://doi.org/10.1145/3373376.3378483

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[38] libMemcached.org. 2009. libMemcached. http://www.libMemcached.org

[39] Zhen Lin, Mohammad Alshboul, Yan Solihin, and Huiyang Zhou. 2019. Exploring memory persistency models for gpus. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 311–323.

[40] Linux Kernel Organization. 2020. Direct Access for Files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[41] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[42] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 258–270.

[43] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[44] memcached 2009. Memcached. http://memcached.org/.

[45] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[46] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[47] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[48] pmem.io. 2017. Persistent Memory Development Kit. http://pmem.io/pmdk.

[49] Yoav Raz. 1992. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 292–312.

[50] J. Ruppert. 1995. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms* (1995).

[51] Steve Scargall. 2020. *Concurrency and Persistent Memory*. Apress, Berkeley, CA, 277–294. https://doi.org/10.1007/978-1-4842-4932-1_14

[52] Steve Scargall. 2020. *PMDK Internals: Important Algorithms and Data Structures*. Apress, Berkeley, CA, 313–331. https://doi.org/10.1007/978-1-4842-4932-1_16

[53] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/3079856.3080240

[54] 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.

[55] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[56] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[57] Matthew Wilcox. 2014. Add Support for NV-DIMMs to Ext4. https://lwn.net/Articles/613384/.

[58] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 623–637. https://doi.org/10.1145/3385412.3386000

[59] 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

[60] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 167–181.