FliT: A Library for Simple and Efficient Persistent Algorithms

Yuanhao Wei
Carnegie Mellon University, USA
yuanhao1@cs.cmu.edu

Naama Ben-David
VMware Research, USA
bendavidn@vmware.com

Michal Friedman
Technion, Israel
michal.f@cs.technion.ac.il

Guy E. Blelloch
Carnegie Mellon University, USA
guyb@cs.cmu.edu

Erez Petrank
Technion, Israel
erez@cs.technion.ac.il

Abstract

Non-volatile random access memory (NVRAM) offers byte-addressable persistence at speeds comparable to DRAM. However, with caches remaining volatile, automatic cache evictions can reorder updates to memory, potentially leaving persistent memory in an inconsistent state upon a system crash. Flush and fence instructions can be used to force ordering among updates, but are expensive. This has motivated significant work studying how to write correct and efficient persistent programs for NVRAM.

In this paper, we present FliT, a C++ library that facilitates writing efficient persistent code. Using the library's default mode makes any linearizable data structure durable with minimal changes to the code. FliT avoids many redundant flush instructions by using a novel algorithm to track dirty cache lines. It also allows for extra optimizations, but achieves good performance even in its default setting.

To describe the FliT library's capabilities and guarantees, we define a persistent programming interface, called the P-V Interface, which FliT implements. The P-V Interface captures the expected behavior of code in which some instructions' effects are persisted and some are not. We show that the interface captures the desired semantics of many practical algorithms in the literature.

We apply the FliT library to four different persistent data structures, and show that across several workloads, persistence implementations, and data structure sizes, the FliT library always improves operation throughput, by at least 2.1× over a naive implementation in all but one workload.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Hardware** → **Fault tolerance**.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508436

Keywords: Non-volatile Memory, Concurrent Data Structures, Recoverability

1 Introduction

The long-anticipated fast, byte-addressable non-volatile random access memories (NVRAM) are now a reality, with Intel Optane available alongside DRAM in the newest machines. NVRAM promises to revolutionize persistent algorithms, with speeds up to three orders of magnitude faster than SSD. However, designing correct persistent algorithms for NVRAM is notoriously difficult. Subtle bugs are easy to overlook. The main difficulty stems from the fact that, for the time being, caches and registers remain volatile.[1] This means that if programs are simply run as they would be on DRAM, significant parts of the state of memory could be lost upon a system crash, thus not achieving meaningful persistence. On the other hand, programs designed for SSD or disk cannot efficiently work as-is on NVRAM, due to the finer atomic granularity of this new memory technology. New techniques must therefore be developed to achieve correct and efficient persistence on NVRAM.

To prevent values on cache from being lost upon a crash, programmers must use explicit flush and fence instructions to push cache lines to NVRAM in a certain order. Care must be taken in deciding which values to flush and when to execute the flush and fence instructions, since these instructions are expensive. Researchers have therefore dedicated significant effort to carefully reasoning about inherent dependencies in algorithms, to omit flushes when it is safe to do so, yet still guarantee persistence on NVRAM [9–13, 16, 17, 24, 25, 30, 37].

Data races in persistent programs pose even more challenges. Since writing and persisting values cannot be done atomically, a value can be visible to other threads before being persisted. Thus, to avoid memory inconsistencies, a process may have to flush locations it reads, even if processes flush locations when they write as well. However, in

[1] While Intel announced the new eADR technology [21] that promises to persist cache contents as well, this would require powerful and expensive batteries to implement. Thus, it is unlikely that volatile caches will cease being a reality in the near future [31].

most cases, a writing process can finish persisting its new value before any other process reads it. In that case, it seems wasteful to have the reader flush this value as well. Existing work in the literature avoids these wasteful flushes by using a bit in each memory word to indicate whether or not it has already been flushed [14, 19, 35]. This optimization has been shown to have tremendous benefits in practice, but borrowing a bit from each word is not always possible. Furthermore, this optimization requires modifying memory using compare-and-swap, and therefore cannot be applied to data structures designed with other primitives, such as fetch-and-add or swap.

We propose a new technique for avoiding unnecessary flushes which is fully general in the sense that it can be applied to any code safely. The idea is to use counters (separate from the memory word) to keep track of ongoing stores for each variable. When a store begins, it *tags* the memory location it operates on by incrementing the corresponding counter. Loads check the counter when accessing a given memory location, and only execute a flush instruction on it if it is tagged. In this way, flush instructions are only executed when needed. This technique allows for flexibility in the placement of these counters. The counters can be, for example, placed next to each variable or in a separate hash table. We experiment with different options in Section 6.

We package this technique into an easy-to-use C++ library called *FliT*, or *Flush if Tagged*, which helps programmers easily design efficient persistent code for NVRAM, abstracting away details of flush and fence instructions, and applying the optimization under the hood. At a high level, the FliT library persists the effect of each instruction without requiring the programmer to handle low-level flushing and barriers.

The FliT library greatly improves the performance of persistent code, since it enables the program to safely skip flush instructions when they are not needed. Furthermore, FliT is easy to use, and its syntax requires minimal changes when applying it to existing code. Indeed, to use FliT, the programmer simply needs to modify the declaration of variables to be persisted, and annotate when an operation terminates – this already makes any linearizable data structure durably linearizable [23]. For example, a C++11 implementation of Harris's linked list [20] can be made durably linearizable using our library by changing just seven lines of code.

Another advantage of the FliT library is its flexibility; while, by default, the FliT library instruments each load and store instruction to access the tag counters, this does not have to be the case. Many previous works have focused on understanding which values must be persisted, and which can be left volatile [9, 11, 12, 14, 16, 17]. These efforts have led to many optimized persistent data structure implementations. The FliT library can complement these existing works by allowing the programmer to specify whether a specific instruction's arguments should be left volatile. In that case, the instruction can be annotated as such, and the flushing mechanism is bypassed. Thus, while the FliT library can be used to persist all memory values in a naive manner to yield a fairly performant solution, it can be combined with existing optimizations to yield even better results.

To more formally argue about the library's correctness, we define an *abstract interface*, called the *P-V Interface*, which the FliT library implements. Intuitively, the interface considers two types of instructions; those whose effects must be persisted (called *p-instructions*), and those whose persistence has been optimized away (called *v-instructions*). The P-V Interface describes the interaction between these two types of instructions and the resulting effect on the memory. We show that the P-V Interface captures persistence behavior in many algorithms in the literature. Intuitively, the P-V Interface abstracts flush and fence instructions down to their underlying meaning, and we use it to show that the FliT library behaves as expected. We believe that the P-V Interface offers a good balance between ease of programming and the efficiency of potential implementations. Since it is relatively low-level, it can be implemented efficiently, as is exemplified by FliT. Furthermore, designing durably linearizable data structures [23] is easy using the P-V Interface; if every instruction is made a p-instruction, a linearizable data structure becomes durable. On the other hand, carefully reasoned optimizations can also be applied by making some instructions v-instructions where possible. Thus, we believe that the P-V Interface may be of independent interest.

We evaluate the FliT library by using it to implement four different durable data structures; a linked-list [20], a BST [27], a skiplist [14], and a hash table [14]. Furthermore, for each data structure, we evaluate three different ways of making it durable; one that makes all instructions p-instructions, and two more optimized settings that appear in the literature; we consider the NVTraverse methodology [16], which allows us to have v-loads while traversing the data structure, and a manually optimized durable version of the same data structure [14]. We also evaluate different settings for the placement of the counters in the implementation of FliT, and compare these to the existing bit-tagging technique [14, 19, 35]. We observe that, the FliT library provides up to 200× speedup over a durable linearizable version implemented with plain flush instructions. Furthermore, even for highly optimized implementations, the FliT library still provides up to 4.32× speedup and never slows down any implementation.

In summary, the contributions of the paper are as follows.

- We present a new technique for tracking dirty cache lines that is fully general.
- We present the FliT library, which uses this technique to instrument instructions giving an easy way to design efficient persistent code.
- We formalize the FliT library interface as the P-V Interface, which captures many practical use cases, and

gives a simple way of creating durably linearizable data structures.
- We evaluate the FliT library and show it can significantly improve the performance of even the most optimized persistent algorithms.

The rest of this paper is organized as follows. Background is discussed in Section 2. In Section 3, we present the P-V Interface definition. We rely on this interface when presenting the FliT library syntax in Section 4, and its implementation and algorithmic ideas in Section 5. We evaluate the FliT library in Section 6. Finally, we discuss additional related work and conclude the paper in Sections 7 and 8.

2 Background and Preliminaries

Flushing on current hardware. In existing architectures, there are specific *flush* instructions which write back a value from a specific cache line to the main memory. The flush instructions might differ by their strength and whether they invalidate the cache line, which influences performance. In addition, there are *fence* instructions which provide ordering. A store fence ensures that all preceding writes and flushes executed by a specific process are visible to other processes before any writes or flushes executed after the fence. A flush followed by a fence blocks until all previously flushed locations have reached main memory, which may be volatile (i.e., DRAM), or non-volatile (i.e., NVRAM), depending on the mapping of the specific flushed address.

In the rest of this paper, we will use the term *pwb* (persistent-write-back) to refer to the weakest form of flushing, which does not block or invalidate. It is *persistent*, since in all our use-cases, the memory it flushes is mapped to NVRAM. As mentioned above, after a non-blocking flush instruction, a fence must be called to ensure the completion of the flush. In this paper, we use *pfence* to refer to a fence instruction. A pfence called by a process i is assumed to order all previous pwb instructions called by i before any pwb or write instructions that are executed after the pfence. The pwb and pfence instructions are architecture-agnostic.

Previous flushing optimizations. We defer most of the discussion of related work to Section 7. However, some flushing optimizations have appeared in the literature that are reminiscent of the FliT library's implementation, so we briefly discuss them now. David et al. [14] introduce a technique they call *link-and-persist* to avoid executing pwb instructions when the variable being flushed is clean. Their technique works by using a single bit in each memory word as a flag indicating whether or not it has been flushed since the last time it was updated. When a new value is written, it is written with the flag up. The writing process then executes a pwb and a pfence to persist the new value, and then executes another store to flip the flag down. A reader executes a pwb on any location it read that had the flag up, and skips flushing every time the flag is down. This technique has appeared in the literature under different names [19, 35, 39], always optimizing redundant pwbs, and yielding faster algorithms. This technique is similar to the implementation of our FliT library. However, the FliT library is more general and flexible. For one, it does not require taking a bit in every memory word. While pointers leave unused bits in each word, some algorithms make use of these bits for other parts of their logic. The link-and-persist technique is not applicable to such algorithms. Furthermore, for link-and-persist to work, all stores must be executed using a CAS instruction (as opposed to, for example, *fetch-and-add* or *swap*), to prevent accidentally removing a flag for a value that has not yet been flushed. The FliT library does not suffer from these restrictions. Finally, as will be shown in the rest of the paper, the FliT library also provides flexibility in allocating the space used for metadata tracking persistent state, which can sometimes be a useful way to optimize implementations.

Model. We consider a shared memory setting in which n processes access two types of memory; volatile memory and persistent memory. Volatile memory roughly corresponds to caches and registers, as well as DRAM, on real architectures, whereas persistent memory corresponds to the NVRAM. We assume that upon a *system crash* anything that is in persistent memory remains, but anything on volatile memory is lost.

Processes can access shared memory using *read*, *write*, or read-modify-write (RMW) instructions, like *compare-and-swap (CAS)*, *fetch-and-add (FAA)*, and *test-and-set (TAS)*. We sometimes refer to read instructions as *loads*, and to all other instructions collectively as *stores*. Each memory location is categorized at any given point in time as *shared* or *private*. There is a *root* location in memory, that is always shared. A private memory location can only be accessed by a single process i, which can make that location shared by executing a specific store on some shared location. This store depends on the algorithm; it could be releasing some lock, or swinging a shared pointer to point to this location.

All accesses are applied to volatile memory. To make a value that is in volatile memory appear in persistent memory, processes can execute *persistence instructions*, which include *pwb* and *pfence* (as long as these addresses are mapped to the persistent memory). From now on in the paper, whenever we invoke pwb on some location, we assume that location is mapped to persistent memory. A pwb instruction takes a memory location as a parameter. The value v in memory location ℓ is said to be *flushed* if a pwb instruction was executed on ℓ when v was in ℓ. After a process i executes a pfence instruction, any value that was flushed by i is in persistent memory.

A data structure D defines a set of operations, along with a *sequential specification*, defining how the operations behave in a sequential execution. Histories are composed of *operations* and *crash events*. A crash event erases all values in volatile memory, but leaves the persistent memory intact.

Furthermore, after a crash event, new processes are spawned. A history H of operations of data structure D, with no crash events, is *linearizable* if there is a single point in time during the execution of each operation at which that operation *takes effect*, such that the sequence of these points adheres to the sequential specification of D. A history with crash events is *durably linearizable* [23] if it is linearizable after all crash events are removed from it. A data structure implementation is linearizable (resp. durably linearizable) if all possible histories of it are linearizable (resp. durably linearizable).

3 Persistent-Volatile Instruction Interface

Before presenting the FliT library, we define the abstract interface that it implements. This interface, called the P-V Interface, is important for discussing the correctness of the FliT library implementation; we later prove that our implementation satisfies the abstract interface. Furthermore, this interface allows users of the FliT library to reason about their code in a precise manner.

The P-V Interface aims to capture the behavior of a program with both volatile and persistent memory. Firstly, handling persistence should not affect the behavior of the volatile memory. In particular, this means that we should expect to see the same sequential semantics on volatile memory as we do in a classic system. That is, any load on volatile memory should return the value written by the most recent store. For persistence, we expect the interface to capture the behavior of code that uses pwb and pfence instructions. We also note that dependencies between instructions can play a role in when we expect a value to be persisted; if a value has been written but never read, it may be ok for it to be lost upon a system crash, since its effects have not yet been observed. We formalize these intuitions below.

We begin defining the interface by introducing terminology to separate two types of instructions: we say an instruction is a *p-instruction* if it has to be persisted (defined below), and a *v-instruction* if it does not. More specifically, we refer to persisted loads and stores as *p-loads* and *p-stores* respectively, and to their volatile counterparts as *v-loads* and *v-stores*. If we do not specify whether an instruction is persisted or volatile, then it could be either.

To nail this down precisely, we further distinguish between *shared instructions*, which can race with other shared instructions to the same location, and *private instructions*, which cannot race with any other instruction. A private instruction may allow more flexibility in when it is persisted, since other processes cannot observe its effects.

We refer to the memory location an instruction operates on as its *location*. Furthermore, we associate a *value* with each instruction; a *load*'s value is what it returned (read from its location), and a *store*'s value is the value newly written on its location. When we say an instruction is *persisted*, we mean its value is on persistent memory.

To create durable code, we must reason about *dependencies* among different instructions. In particular, for a new store to be safe in a persistent setting, a process i must ensure that all its dependencies have been persisted *before* executing the store. That is, the values that process i used to determine the value and location of the new store must not be lost at a later time. Furthermore, to maintain a store-order guarantee for persistent memory, previous store instructions by the same process i must also be persisted before i's new store. Finally, to prevent losing the effects of a completed operation, we must persist all of i's dependencies and store values before i completes an operation.

The P-V Interface, defined in Definition 1, formalizes the meaning of dependencies in terms of p-stores and p-loads. Conditions 2 and 3 of Definition 1 define the "depends on" relationship which is later used in Condition 4. Intuitively, a process i *depends on* its own p-stores (Condition 2), and on previous p-stores on locations on which i executes a p-load (Condition 3). The interface then requires that these dependencies be persisted before i executes a store that is visible to other processes (shared), or before it completes an operation (Condition 4). To capture which p-stores become dependencies, we consider the *linearization* of instructions. Intuitively, an instruction linearizes at the time it accesses volatile memory (Condition 1). Note that Conditions 1, 2, and 3 apply to both private and shared instructions, and that v-instructions do not add dependencies.

Definition 1. *[The P-V Interface.] Each instruction has a linearization point within its interval, such that:*

1. ***Keeping Volatile Memory Behavior.*** *A load r on location ℓ returns the value of the most recent store on ℓ that linearized before r.*
2. ***Store Dependencies.*** *Let s be a linearized p-store executed by a process i. i depends on s.*
3. ***Load Dependencies.*** *Let r be a p-load by process i on location ℓ. i depends on every p-store on ℓ that was linearized before r.*
4. ***Persisting Dependencies.*** *Let t be either the linearization point of a shared store by process i, or the time at which i completes an operation. The value of every store i depended on before time t is persisted by time t.*

3.1 Applicability of the P-V Interface

In this subsection, we show that for many algorithms designed for NVRAM, it is easy to replace their memory accesses and all pwb and pfence instructions with p-instructions (for dependencies) and v-instructions (for instructions optimized out as non-dependencies).

Simple durability. We begin by considering how to guarantee durability using the P-V Interface for any given linearizable algorithm. Izraelevitz et al. [23] show that, for any linearizable data structure, if every load-acquire and store-release is accompanied by a pwb and a pfence, and stores are

followed by a pwb, then the data structure becomes durable. We show that declaring these instructions as p-instructions achieves the same guarantee. Furthermore, using our implementation of the P-V Interface yields a much faster solution.

Theorem 3.1. *Given a linearizable data structure, if we make all its loads and stores p-instructions, then the resulting data structure is durably linearizable.*

Due to lack of space, the full proof is deferred to the supplementary material.

NVTraverse. While declaring each load and store as a p-instruction is very simple, and can be easily applied to any linearizable algorithm to make it persistent, there may be opportunities to optimize such a construction if some instructions could be identified as non-dependencies (marked as v-instructions). This can give more flexibility to the underlying implementation to omit pwb and pfence instructions where possible. Indeed, there are several constructions of durable data structures in the literature that do not persist every memory instruction. For example, Friedman et al. [16] present a general construction to make certain lock-free data structures persistent more efficiently than the construction of Izraelevitz et al. mentioned above. In particular, they consider data structures in *traversal form*, in which each operation has a read-only traversal phase followed by a short critical phase. Many lock-free data structures, including linked-lists, BSTs, and skiplists, can fit this form. Friedman et al. show that such data structures do not need to execute any pwb instructions during the traversal phase. That is, any load in the traversal phase can be thought of as a v-load, and any instruction (load or store) in the critical phase can be thought of as a p-instruction. There is a short *transition* between the traversal and critical phases in NVTraverse, in which some locations that were read during the traversals are flushed. This can be achieved by executing p-loads on those locations.

Other Algorithms. Many other NVRAM algorithms appear in the literature, with various techniques to optimize the interaction with persistent memory. As a general rule of thumb, any instruction that is not immediately followed by a pwb in such algorithms can be seen as a v-instruction, and any other instruction can be seen as a p-instruction. The 'dependency' terminology is used intuitively in several works [14, 17]; generally, non-dependencies in those works can be seen as v-instructions.

4 The FliT Library and Interface

In this section, we introduce the FliT library, which implements the P-V Interface defined in Section 3. At its core, FliT provides an interface with which to declare each instruction as either a p- or v-instruction (using the `pflag` parameter).

The FliT library is implemented in C++ and is available at https://github.com/cmuparlay/flit. To use the library, a programmer must declare variables as `persist<>`. The `persist` template can take any type. Declaring a variable in this way essentially allows the FliT library to track its persistence state. Whenever this variable is accessed for loads or stores, the instruction is overloaded with the library's implementation of it, which we call a flit-instruction. Each flit-instruction takes the standard arguments for its underlying instruction, in addition to a flag specifying whether it is a v- or a p-instruction. Finally, a special `operation_completion` function is made available, which must be called at the end of each data structure operation. Figure 1 shows the basic interface.

```
1  class persist<T, default_pflag> {
2  public member functions:
3    T load(bool pflag = default_pflag);
4    void write(T value, bool pflag = default_pflag);
5    bool CAS(T oldval, T newval, bool pflag = default_pflag);
6    T exchange(T newVal);
7    int FAA(int amount, bool pflag = default_pflag);
8    // FAA is only supported if T is an int type
9  public static functions:
10   void operation_completion(); } ;
```

Figure 1. Basic interface of FliT.

```
1  struct Node {
2    persist<int, flush_option::persisted> key;
3    persist<T, flush_option::persisted> value;
4    persist<std::atomic<Node*>, flush_option::persisted> right;
5    persist<std::atomic<Node*>, flush_option::persisted> left; };

7  persist<Node*> root;

9  void lookup(int key) { // automatic BST lookup
10   Node* node = root->left;
11   while(node->left != nullptr) {
12     if(key < node->key) node = node->left;
13     else node = node->right; }
14   bool result = (node->key == key);
15   persist::operation_completion();
16   return result; }

18 bool insert(K key, V val) { // automatic BST insert
19   ...
20   persist::operation_completion();
21   return result; }
```

Algorithm 2. FliT library used for a concurrent BST.

The FliT library further improves the syntax of this interface to allow for minimal code changes to apply it. In particular, when declaring a variable in the `persist` template, a default `pflag` value can be specified, making the `pflag` argument optional when executing instructions on this variable. Furthermore, we overload the `->` and `=` operators to execute FliT loads and stores instead of the default one. These operators can only be used with the default `pflag` value though, since it does not allow for an additional argument.

Algorithm 2 shows an example of the implementation of a concurrent binary tree, achieving durability by making all instructions p-instructions. The change over the original code is highlighted in red. All fields within a node are declared with the `persist<>` template, and given the persisted option as a default for the `pflag`. This means that without any code

changes, all accesses to these node fields will be persisted flit-instructions. In the example, all code elided inside the '. . .' remains identical to the original implementation. Note that FliT is purely library-based and does not require any changes to the compiler or run-time environment.

The example above only shows the use of a single setting; all instructions are called as the default p-instructions. The FliT library is in fact more flexible, and is still easy to use even for more complicated code. We note that while not shown in the example, it is also possible to leave a variable declaration as-is, without using the `persist` template, if that variable never requires persistence. This use case arises in some algorithms. For example, Friedman et al. [17] present a durable queue implementation that completely avoids flushing the head and tail pointers of the queue. In this case, these variables can be declared normally, without the FliT library.

5 The Algorithm

We now describe the implementation of the FliT library, which satisfies the P-V Interface specified in Section 3. At a high-level, each p-store flit-instruction executes a pfence before its store, and a pwb on this location after the store. This means that already, conditions 1, 2 and 4 are satisfied (ignoring dependencies from Condition 3). If we had a guarantee that every p-load to any location ℓ will always happen after persisting the most recent p-store on ℓ, then Condition 3 would be satisfied as well, without having to change the implementation of load instructions at all. However, this is not the case; since we cannot store and persist atomically, it is possible for another process to read a value written into ℓ by a shared p-store before the writing process persists.

One way to handle dependencies from Condition 3 is to have each p-load execute a pwb after reading its value. However, this would introduce many unnecessary pwbs, since most pwbs do not execute concurrently with a pending p-store on the same location. Our goal in this work is to avoid as much excessive flushing as possible.

The basic idea behind the implementation of FliT is to associate each `persist` variable with a counter, which we call the *flit-counter*. Intuitively, this counter keeps track of the number of pending p-store flit-instructions. When a p-store flit-instruction begins, it increments its associated flit-counter. It then executes its modification, followed by a pwb on this location, and then decrements the counter. This counter is checked by all p-loads on this location, and if its value is non-zero, the p-load executes a pwb after reading the value. A location whose flit-counter is non-zero is said to be *tagged*, and p-loads only flush locations that are tagged (i.e. Flush if Tagged (FliT)); this is where the FliT library gets its name. This counter will only go above one if there is contention.

Store flit-instructions also execute a pfence before beginning their execution, and another one before decrementing the counter (in the case of a p-store). These pfences ensure

```
1  T shared_load(T* X, bool pflag) {
2    T val = X->load();
3    if (pflag && flit_counter(X) > 0)
4      PWB(X);
5    return val; }

7  T private_load(T* X, bool pflag){ return X->load(); }

9  void shared_store(T* X, T args, bool pflag) {
10   PFENCE();
11   if (pflag) {
12     flit_counter(X).fetch&add(1);
13     X->store(args);
14     PWB(X);
15     PFENCE();
16     flit_counter(X).fetch&sub(1); }
17   else X->store(args); }

19 void private_store(T* X, T args, bool pflag) {
20   if (pflag){
21     X->store(args);
22     PWB(X); // flush all cachelines spanned by *X
23     PFENCE();
24   } else X->store(args); }

26 void completeOp(){ PFENCE(); }
```

Algorithm 3. The Flush-Marking Algorithm

that all modifications are persisted at the correct times according to Definition 1. In particular, the pfence before a store ensures Condition 4 holds, by making sure all values pwbed by this process (which includes all of its dependencies) have been persisted. The pfence before decrementing the counter is required for Condition 3; if this pfence is not executed, a p-load may observe the flit-counter at value 0 and avoid flushing the location, even though the written value has not yet been persisted.

The FliT library implementation distinguished between shared and private accesses to the memory. The details above in fact describe the implementation for shared accesses. If a given flit-instruction is private, then its implementation is more efficient; we can ignore the flit-counter associated with the accessed location, and avoid the pfence before a private p-store. Intuitively, if a flit-instruction cannot be concurrent with any other, then the accessed location is guaranteed not to be tagged (i.e. its counter has value 0), and return to this state (in the case of a store) before the next flit-instruction accesses it. Therefore, there is no need to check it, or to leave any traces for other processes. Furthermore, note that Condition 4 of the P-V Interface only requires persisting before *shared* stores, so we can skip the pfence before private stores. Unless specified otherwise, we always discuss shared flit-instructions in the text, since their implementation is more involved than that of private ones. The pseudocode of the implementation of instructions on `persist` variables is presented in Algorithm 3. Recall that p- and v-instructions are distinguished by the `pflag` argument. We combine all types of store flit-instructions (CAS, FAA, write, etc) into one in the pseudocode, since their behavior is the same.

Note that we do not specify how each memory location is associated with a flit-counter. In Section 5.1, we discuss possible ways to assign counters to memory locations, but we note that this is flexible. In particular, having many concurrent stores to the same memory location, or sharing a flit-counter among several locations, cannot result in unsafe behavior (though it may result in extra pwbs executed).

Theorem 5.1. *Algorithm 3 satisfies Definition 1.*

The proof of this theorem can be found in the full version of the paper [36].

5.1 Placement of the Counter

In Algorithm 3, we intentionally abstracted away how flit-counters are assigned to memory locations, using the unspecified `flit-counter()` function. Note that the flit-counter is completely decoupled from the memory locations it represents, so it can be placed anywhere, and can be shared by any number of locations. Furthermore, the flit-counters can usually be very small; the maximum value in a flit-counter is at most the number of threads, since each thread can increment at most one flit-counter at most once before decrementing it. Therefore, on most machines, including the one we test on, 8 bits suffice to store a flit-counter without the possibility of overflow, assuming the program does not spawn more threads than processors. If larger counters are needed, the user can specify a maximum thread count and our library will ensure the counters are large enough to store this number.

In this section, we discuss a couple of practical implementations for the `flit-counter()` function, which we later implement and test. However, we remind the reader that other practical implementations are possible, and that the FliT library allows the flexibility of modifying the counter placement to suit the needs of the user.

Adjacent Counter. One straightforward way to implement the flit-counters is to place each counter adjacent to the memory word that uses it. That is, we can make each memory word in an algorithm be a double-word, and use the second word for the flit-counter. The advantage of this is that the counter for each word X is on the same cache line as X, and therefore accessing it has minimal cost. However, this approach can be inconvenient and wasteful, since this fundamentally changes the memory layout of a given data structure's objects. Indeed, an object that fit in a single cache line might overflow it if all its fields double in size.

Hashed Counter. Another flit-counter placement strategy is to use a hash table; each memory location X hashes into the table, which has a counter in each of its entries. This method allows different memory locations to use the same flit-counter. The number of collisions depends on the ratio of the size of the hash table and the number of threads in the system, since each thread can access at most one hash-table entry per flit-instruction. The advantage of this approach is two-fold. First, it saves memory. In many data structures, especially if they are not highly-contended, most memory locations will have no pending p-stores most of the time. This means that sharing counters results in a negligible amount of extra flushing. Secondly, it does not require changing the layout of memory in the data structure itself, since the flit-counters are not placed in the same cache lines as the data structure elements. However, this can also be a downside in some situations; since the flit-counter is in a separate cache line, accessing it could incur an additional cache miss.

Note also that the hashing method allows us to compact the memory usage of flit-counters even further, by squeezing several counters into each word. Recall that 8 bits suffice for each flit-counter in our experiments, so we can fit 8 counters in a single memory word. However, compacting the flit-counters in this way can increase false-sharing; many different memory locations could be mapped to counters on the same cache line.

6 Evaluation

The FliT library's implementation optimizes pwb instructions on shared locations. To highlight its effects and focus on them in the evaluation, we evaluate the library applied to lock-free data structures, in which most memory accesses are shared. We apply the FliT library to 4 lock-free data structures; a linked-list [20], a binary search tree (BST) [27], a skiplist [15], and a hash table which uses Harris's linked list to implement each bucket [20]. For each data structure, we implement three different ways of making it durable; the first is the **automatic** transformation discussed in Section 3.1, in which all instructions are made p-instructions, the second is using the **NVtraverse** framework [16], and the third is a hand-tuned (**manual**) construction based on algorithms presented by David et al [14]. We also study the effect of various policies for placing the flit-counters in memory with respect to the memory locations they are associated with. In particular, we implement the adjacent counter variant (flit-adjacent) and a hash table (flit-HT), for which we test five different sizes. We evaluate the tradeoffs of the different approaches. Finally, we also implement the link-and-persist technique described by David et al. [14][2]. We compare these implementations of the P-V Interface with the **plain** version, which places pwb and pfence instructions where necessary, but does not utilize any tagging method to avoid pwbs in the loads.

Setup. We run experiments on a machine with two Xeon Gold 6252 processors (24 cores, 3.7GHz max frequency, 33MB L3 cache, with 2-way hyperthreading). The machine has 375GB of DRAM and 3TB of NVRAM (Intel Optane DC memory), organized as 12 × 256GB DIMMS (6 per processor).

[2]There are other implementations of link-and-persist [19, 35, 39], but they all perform essentially the same steps.

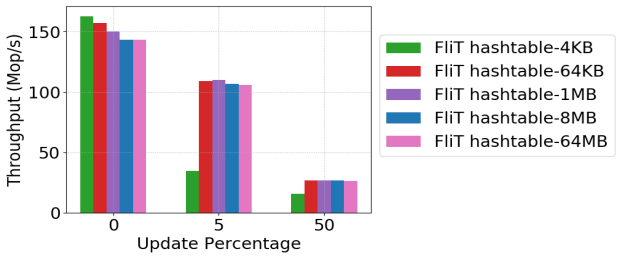


Figure 4. Tuning hashtable size for the FliT library. Throughput shown is for the automatic BST with 10K keys.

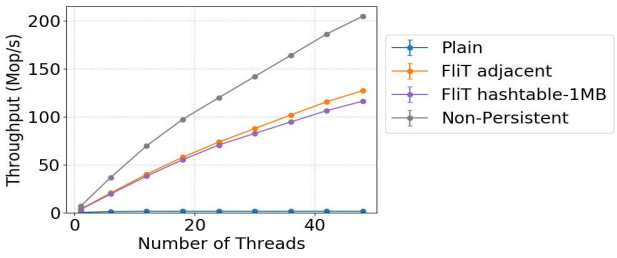


Figure 5. Scalability graph for the automatic BST with 10K keys and 5% updates.

On Intel/AMD architectures [1, 22], the three available flush instructions are *clflush*, *clflushopt*, and *clwb*, where *clwb* is not blocking and supposed to not invalidate the cache. Thus, *clwb* is the most efficient one, and is the one we use in our implementation. The processors are based on the Cascade Lake SP microarchitecture, which supports the `clwb` instruction for flushing cache lines (pwb). However, its implementation of *clwb* still invalidates cache lines. Performance might be improved in future platforms where clwb does not invalidate cache lines. For ordering, we use the *sfence* instruction. The equivalent instructions on ARM are *DC CVAP* and a full system *DSB* instruction for flush and fence execution [2]. We use *libvmmalloc* from the PMDK library to place all dynamically allocated objects in NVRAM, which is configured in an App-Direct mode to let the NVRAM reside alongside the DRAM and allow byte addressable access. All other objects are stored in RAM. The operating system is Fedora 27 (Server Edition), and the code was written in C++ and compiled using g++ (GCC) version 7.3.1. We use `std::atomics` with relaxed memory orders where appropriate. In our implementation of Algorithm 3, some of the pfence instructions can be omitted because on our Intel machine, atomic instructions (such as CAS and FAA) perform an implicit pfence.

We avoid crossing NUMA-node boundaries, since unexpected effects have been observed when allocating across NUMA nodes on the NVRAM . Hyperthreading is used for experiments with more than 24 threads. Unless stated otherwise, all data structures are tested with three different workloads; 0% updates, 5% updates, and 50% updates. Updates are split 50/50 between inserts and deletes, and chosen randomly. All experiments were run for 5 seconds and an average of 5 runs is reported.

FliT Hash Table Size. We begin our evaluation by testing the effect of the size of the flit-HT on performance. There is a trade-off between memory footprint and the number of collisions on the counters; keeping the table small allows it to fit in cache, making accesses to it potentially cheaper. However, if it is too small, hash collisions could cause cache coherence misses. Figure 4 shows the result of different flit-HT sizes on the BST, with three different update ratios. We show the automatic BST implementation. Other data structures showed similar patterns, and are omitted for brevity.

We first note that for 0% updates, we see that the larger the hash table, the lower the throughput. This is as expected; as the flit-HT grows, less of it fits in cache, and therefore accesses to it more frequently incur cache misses. Furthermore, at 0% updates, the flit-counters are never updated, so coherence misses are not a concern. Starting at 5% updates, we see a stark performance drop for the 4KB hash table. Two types of hash collisions can occur in this framework: (1) two locations hash to the same counter, resulting in potentially redundant pwbs executed, if the flit-counter balance is inflated due to an ongoing p-store on a different location. More severe, however, is the second type of hash collisions: (2) cache line collisions; the 4K flit-counters in the hash table are packed into only 64 cache lines. This means that if any two p-instructions, at least one of which is a p-store, occur on locations that hash to the same cache line (quite likely), they suffer a coherence cache miss. In such a small hash table, this effect is very prominent. This is much less noticeable in the larger hash tables.

For the rest of the plots, we show only one hash table size; the 1MB flit-HT. We note that this size fits in the L3 cache, but is large enough to avoid most hash collisions.

Varying Number of Threads. We now consider the scalability of data structures that use the FliT library, as the number of threads grows. The results can be seen in Figure 5. Again, the automatic 10K BST with 5% updates is shown. Note that in this plot, aside from the flit-HT, we show a few different settings for comparison. In particular, the gray line shows a non-persistent version of the data structure, in which no pwb or pfence instructions are issued. This forms a baseline that cannot be significantly outperformed by any persistent implementation. We run the non-persistent version also on NVRAM to show the software overheads of persistence. In Figure 5, this overhead is about 44% for flit-HT.

The blue line shows a BST version implemented with plain pwb and pfence usage, without applying the FliT library at all. This version performs many more pwbs, and its performance

and scalability suffer. We show both the flit-HT and the flit-adjacent versions of the FliT library. Both of them scale similarly, and quite well.

We also ran scalability experiments for our linked list, hashtable, and skiplist data structures and the shape of the graphs were similar to the one shown in Figure 5. The remaining graphs in this section focus on workloads with 44 threads because the relative performance of the algorithms stays the same across thread counts.

Comparing Durability Methods. Figure 6 shows the four implemented data structures, each with their three different methods of durability: automatic, NVTraverse, and manual. When using the FliT library, these methods differ in how many v-instructions they execute; the automatic version only executes p-instructions, the NVTraverse executes many v-loads while traversing the data structure, and the manual version carefully reasons about these individual data structures to make a larger fraction of the instructions be volatile. All plots show 5% updates, and the smaller size of the tested data structure (10K nodes for the scalable data structures, and 128 nodes for the linear linked-list). For each setting, we show a plain implementation, flit-adjacent, flit-HT, and link-and-persist where applicable. The plots also show the performance of the original, non-persistent version of each data structures using a dotted line at the top of each graph.

Generally speaking, the link-and-persist method follows the same patterns as the FliT implementations. We note that the more optimized the underlying durability implementation is, the less it benefits from FliT. However, for all settings, the performance boost from FliT is still substantial; while in the automatic version, FliT boosts throughput by a factor of at least 6.68× (in the hash table), and at most 99.5× (in the skiplist), we still observe an improvement of at least 2.17× when using FliT in all data structures under all durability methods. However, it is also important to note that across the board, the optimized durability methods with FliT outperform the automatic durability method with FliT. Thus, while benefiting less from the FliT library, optimizations that allow using more v-instructions are still useful, and should still be implemented using the FliT library.

Interestingly, while optimized solutions do perform better, the automatic version implemented with the FliT library performs surprisingly well; it significantly outperforms the NVTraverse and manual versions without the FliT library for the BST and hash table, and approximately matches their performance in the linked-list and skiplist.

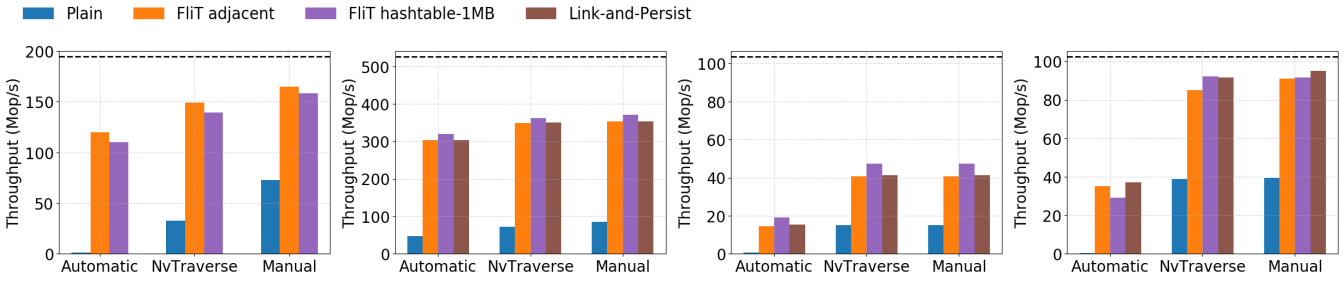
Effect of updates. In Figure 7, we show each data structure with two different sizes, and in each subplot, we vary the update ratio of the workload. These plots are normalized to the throughput of the non-persistent baseline for each data structure. It is easy to see that the more updates executed, the worse the performance of all persistent versions when compared to the non-persistent baseline. This is expected; pwb and pfence instructions are executed more the more update operations occur. Note that in 0% update workloads, no pwb or pfence instructions are executed, other than in initialization and at the end of each operation in the FliT and link-and-persist versions, since loads only ever execute a pwb if the location is tagged (and only p-stores can tag memory locations).

Furthermore, in 0% updates, the flit-adjacent and the link-and-persist do better than the hash-table variant. This is because the latter implementations never have to incur an extra cache miss to access the flit-counter, whereas the flit-HT incurs L2 misses every time it accesses the counter.

Comparing FliT and Link-and-Persist. We note that in general, the flit-adjacent and the link-and-persist implementations perform almost identically. This is because they both avoid this extra cache miss when accessing the flit-counter (or flush-bit in the case of link-and-persist). The exception to this rule is in the skiplist, where link-and-persist outperforms the flit-adjacent. This is because flit-adjacent doubles the size of each node. In most data structures, it goes unnoticed, since each node still fits in a single cache line. However, the skiplist node stores many pointers, and thus can overflow a cache line when each word in it is doubled to fit the flit-counter. This problem does not occur with the link-and-persist. However, we note that link-and-persist is not as general, and cannot be implemented with the BST, since this BST algorithm makes use of all bits in each word.

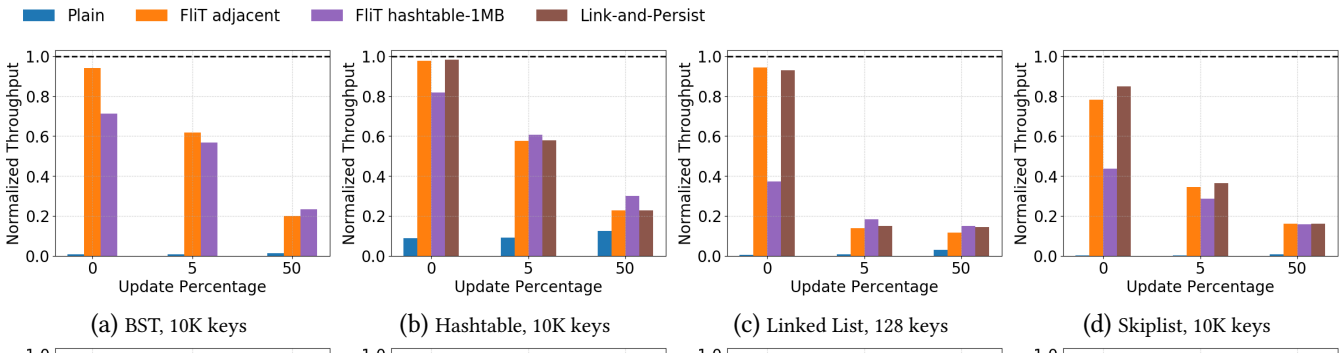
Interestingly, while flit-adjacent/link-and-persist perform best when there are 0% updates, this is not always the case when more updates occur. This is most noticeable in the smaller hash table and linked-list implementations (Figures 7b and c). This is because, while in our implementation, we use Intel's *clwb* instruction to perform pwbs, which should not invalidate cache lines update flushing, invalidations still occur. Indeed, Intel confirms that clwb, while available for use, is not currently implemented in hardware. Therefore, p-stores in the flit-adjacent incur a cache miss when decrementing the flit-counter, since they always do so after having executed a clwb on that cache line, thereby evicting it from memory. The same thing happens in the link-and-persist implementation, when flipping the flush-bit after having flushed the cache line. Since the flit-HT does not place the flit-counter on the same cache line as its p-store is accessing, decrementing the flit-counter does not incur this cache miss. This effect is less prominent in larger data structures, in which traversing the data structure dominates the overall execution time. Furthermore, we believe that this effect will disappear once Intel implement their non-invalidating flush option in hardware.

The effects of data structure size on performance and experiments measuring number of flushes per operation are discussed in the full version of the paper [36].



(a) BST, 10K keys (b) Hashtable, 10K keys (c) Linked List, 128 keys (d) Skiplist, 10K keys

Figure 6. Throughput of 44 threads with 5% updates. Dotted bar represents throughput of the non-persistent version of each data structure on NVRAM.



(a) BST, 10K keys (b) Hashtable, 10K keys (c) Linked List, 128 keys (d) Skiplist, 10K keys

(e) BST, 10M keys (f) Hashtable, 10M keys (g) Linked List, 4K keys (h) Skiplist, 10M keys

Figure 7. Throughput results for 44 threads, automatic, normalized to the throughput of the non-persistent version of each data structure. Dotted bar represents throughput of the non-persistent version of each data structure on NVRAM.

7 Related Work

There have been many papers focusing on finding how to easily and efficiently program for NVRAM. Izraelevitz et al. [23] present the notion of *durable linearizability*, a correctness condition for persistent data structures. At a high level, durable linearizability requires a data structure to be linearizable despite any number of system crashes that occur during its execution. Izraelevitz et al. also showed how to place pwb and pfence instructions in linearizable code with acquire-release consistency to guarantee durable linearizability. In this paper we show how to rewrite this construction in terms of the P-V Interface, and use the FliT library to optimize this implementation. Izraelevitz et al, also introduce a weaker correctness guarantee, called buffered durable linearizability, which we do not consider in this paper.

Researchers have also introduced other correctness criteria for persistence and explored how to support them efficiently [3, 4, 6, 7, 17]. These works consider not only the state of shared memory upon recovery from a system crash, but also whether processes can continue their previous execution. For example, detectability [17], requires that each process be able to find out whether its most recently called operation had completed before a crash. These conditions can be achieved by storing extra metadata beyond what is stored in a non-persisted execution. We believe that using the P-V Interface when designing algorithms for these other correctness criteria can improve performance and portability just as much as it does for durably linearizable implementations.

Many algorithms have been designed for NVRAM in the context of file systems and database indexes [10, 24–26, 33, 37, 38]. These algorithms are often lock-based, rather than the lock-free data structures that we have compared to in our evaluation. We believe that the P-V Interface, and its implementation in the FliT library, can also be used to enhance such algorithms. Indeed, Lee et al. use a technique (like link-and-persist [14]) in their B-tree algorithm [25]. However, we focused on lock-free data structures in our evaluation since the largest benefits in the FliT library's implementation can be seen in contended workloads, which are less prominent in lock-based algorithms. Still, the P-V Interface captures lock-based algorithms as well, leaving room for optimized solutions by treating private instructions (those inside a lock) separately from shared instructions. Similarly, we believe the P-V Interface can be used to write and reason about efficient persistent transactional memories, a topic that has also drawn significant attention in recent years [5, 13, 30, 32].

Several papers provide other programming interfaces for NVRAM. Mnemosyne [34] provides an interface for using persistent memory through *persistent regions*. Atlas [8] provides persistence for general lock-based programs, but does not capture lock-free algorithms. Gogte et al [18] propose semantics for persistent synchronization-free regions. Other works capture the persistence semantics offered by modern architectures, like Intel-X86 [28] and ARMv8 [29]. This line of work differs from ours in its goals; we propose an interface for easy persistent programming, which can be implemented in hardware using the semantics formalized in these papers.

8 Conclusion

In this paper, we introduce the FliT library, a C++ library for designing simple and efficient persistent programs for NVRAM. FliT avoids unnecessary flushing by using *flit-counters* to track dirty cache lines.

We test FliT on an Intel machine with Optane DC memory, and demonstrate that the FliT library not only achieves remarkable speedups over even the most optimized persistent data structures, but is also widely applicable.

Our implementation tested two different ways of allocating the flit-counters and mapping them to memory locations. Many other variants are possible, and it would be interesting to see the effects of different counter allocation strategies on algorithms that use FliT. One natural option that we did not explore is to assign one counter per cache line rather than at the granularity of words.

While FliT's default mode makes any linearizable data structure durable with minimal code changes and impressive performance, it also allows further optimizations. In particular, it allows a programmer to specify some instructions that do not need to be persisted. We capture this flexibility with the P-V Interface, which defines the semantics of code in which some memory instructions can remain volatile, while others must be persisted. This interface is language- and architecture-agnostic, and we show it captures persistence behavior in many algorithms; we believe that the P-V Interface can be implemented on different architectures, and would achieve similar performance gains as those achieved by FliT. Note that the algorithm for maintaining flit-counters is more general than the P-V Interface and can be used to implement other persistent interfaces as well.

Acknowledgments

We thank the anonymous referees for their comments. This work was supported by the National Science Foundation grants CCF-1901381, CCF-1910030, and CCF-1919223 as well as the Israel Science Foundation grant No. 1102/21.

References

[1] AMD. [n. d.]. *AMD64 Architecture Programmer's Manual.* https://www.amd.com/system/files/TechDocs/24594.pdf

[2] ARM. 2018. *ARM Architecture Reference Manual ARMv8.* https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf

[3] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2020. Tracking in Order to Recover-Detectable Recovery of Lock-Free Data Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 503–505. A full version is available from https://arxiv.org/abs/1905.13600.

[4] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 7–16.

[5] H Alan Beadle, Wentao Cai, Haosen Wen, and Michael L Scott. 2020. Nonblocking persistent software transactional memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[6] Naama Ben-David, Guy Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory,. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[7] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust shared objects for non-volatile main memory. In *Conf. on Principles of Distributed Systems (OPODIS)*, Vol. 46.

[8] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, Vol. 49. ACM, 433–452.

[9] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. 2016. NVMOVE: Helping Programmers Move to Byte-Based Persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*. USENIX Association.

[10] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB)* (2015).

[11] Joel Coburn, Adrian 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[12] Nachshon Cohen, Rachid Guerraoui, and Mihail Igor Zablotchi. 2018. The inherent cost of remembering consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM.

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 271–282.

[14] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX Annual Technical Conference*.

[15] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[16] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[17] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Vol. 53. ACM, 28–40.

[18] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. 2018. Persistency for synchronization-free regions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[19] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. 2020. Efficient multi-word compare and swap. In *International Symposium on Distributed Computing (DISC)*.

[20] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*. Springer, 300–314.

[21] Intel. [n. d.]. *eADR: New Opportunities for Persistent Memory Applications.* https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html

[22] Intel. [n. d.]. *Intel Architecture Instruction Set Extensions Programming Reference.* https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[23] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing (DISC)*. Springer, 313–327.

[24] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems.. In *USENIX Conference on File and Storage Technologies (FAST)*. 257–270.

[25] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 462–477.

[26] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. 2009. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 5 (2009), 869–883.

[27] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM.

[28] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. In *ACM Symposium on Principles of Programming Languages (POPL)*.

[29] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, Vol. 3.

[30] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*.

[31] Steve Scargall. 2020. *Persistent Memory Architecture.* Apress, 11–30.

[32] PMDK team. 2018. *Persistent Memory Programming.* https://pmem.io

[33] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *USENIX Conference on File and Storage Technologies (FAST)*, Vol. 11. 61–75.

[34] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–104.

[35] Tianzheng Wang, Levandoski Justin, and Larson Per-Ake. 2018. Easy lock-free indexing in non-volatile memory. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[36] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2021. FliT: A Library for Simple and Efficient Persistent Algorithms. arXiv:2108.04202 [cs.DC]

[37] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *USENIX Conference on File and Storage Technologies (FAST)*. 323–338.

[38] 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 *USENIX Conference on File and Storage Technologies (FAST)*. 167–181.

[39] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*.

9 Artifact Evaluation Appendix

9.1 Abstract

This artifact contains an implementation of the FliT library as well as the source code and scripts needed to reproduce all the graphs in Section 6.

9.2 Artifact check-list (meta-information)

- **Algorithm:** FliT library along with the data structures described in Section 6.
- **Program:** microbenchmarks
- **Compilation:** g++ 9.3.0
- **Binary:** binary not included
- **Run-time environment:** Ubuntu 16.04.6 LTS
- **Hardware:** Multi-core machine, preferably with at least 48 logical cores and NVRAM
- **Output:** graphs from Section 6 as PNG files.
- **Experiments workflow:** one script for compiling the experiments and one script for generating all the graphs.
- **Disk space required (approximately):** 120 MB
- **Time needed to prepare workflow:** approximately 5 minutes
- **Time needed to complete experiments:** approximately 6 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

9.3 Description

9.3.1 How delivered. Available as open source in the `artifact` branch of the following GitHub repository: https://github.com/cmuparlay/flit.

9.3.2 Hardware dependencies. To accurately reproduce our experimental results, a multi-core machine with at least 48 logical cores is recommended. A newer Intel CPU that supports CLFLUSHOPT or CLWB is preferable, but the artifact will also work with the older CLFLUSH instruction. A machine with Intel Optane DC persistent memory is ideal. The artifact includes instructions for running on both persistent memory (NVRAM) and DRAM.

9.3.3 Software dependencies. Our artifact is expected to run correctly under a variety of Linux x86_64 distributions. Our experiments were compiled using g++ 9. For scalable memory allocation in C++, we used jemalloc 5.2.1 as well as the PMDK libvmmalloc allocator to allocate memory from NVRAM. Our scripts for running experiments and drawing graphs require a Python 3 installation with mathplotlib. We used the `numactl` command to evenly interleave memory across the NUMA nodes or restrict the program to a single socket, depending on the experiment.

9.3.4 Data sets. None.

9.4 Installation

Source code can be complied by running `make bench`.

9.5 Experiment workflow

After compiling, use `bash runall-dram.sh` to run all the experiments on DRAM and `bash runall-nvram.sh` to run on NVRAM. The generated graphs will be stored in the `graphs/` directory. Before running the NVRAM experiments, ensure that the machine is configured to App-Direct mode and that the `VMMALLOC_POOL_DIR` parameter (used by libvmmalloc) is properly configured in `runall-nvram.sh`.

9.6 Evaluation and expected results

Given an NVRAM machine with at least 48 logical cores on numa node 0, the graphs generated by `bash runall-nvram.sh` should be very similar to the ones reported in our paper

9.7 Experiment customization

For instructions on how to customize the number of threads, workload, and data structure size in each experiment, please see the README file in the `artifact` branch of the GitHub repository (https://github.com/cmuparlay/flit).

9.8 Notes

None.

9.9 Methodology

Submission, reviewing and badging methodology:

- https://ctuning.org/ae/submission-20190109.html
- https://ctuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging