



go-pmem: Native Support for Programming Persistent Memory in Go

Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian,
and Pratap Subrahmanyam, *VMware*

<https://www.usenix.org/conference/atc20/presentation/george>

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the
2020 USENIX Annual Technical Conference
is sponsored by USENIX.

go-pmem: Native Support for Programming Persistent Memory in Go

Jerrin Shaji George*
VMware

Mohit Verma*
VMware

Rajesh Venkatasubramanian
VMware

Pratap Subrahmanyam
VMware

Abstract

Persistent memory offers persistence and byte-level addressability at DRAM-like speed. Operating system support and some user-level library support for persistent memory programming has emerged. But we think lack of native programming language support is an impediment to a programmer's productivity.

This paper contributes *go-pmem*, an open-source extension to the Go language compiler and runtime that natively supports programming persistent memory. *go-pmem* extends *Go* to introduce a runtime garbage collected persistent heap. Often persistent data needs to be updated in a transactional (i.e., crash consistent) manner. To express transaction boundaries, *go-pmem* introduces a new *txn* block which can include most *Go* statements and function calls. *go-pmem* compiler uses static type analysis to log persistent updates and avoid logging volatile variable updates whenever possible.

To guide our design and validate our work, we developed a feature-poor Redis server *go-redis-pmem* using *go-pmem*. We show that *go-redis-pmem* offers more than 5x throughput than unmodified Redis using a high-end NVMe SSD on memtier benchmark and can restart up to 20x faster than unmodified Redis after a crash. In addition, using compiler microbenchmarks, we show *go-pmem*'s persistent memory allocator performs up to 40x better and transactions up to 4x faster than commercial libraries like PMDK and previous work like Mnemosyne.

1 Introduction

At present, there is a large gap between the access latency for memory and storage hierarchy. The fastest tier of storage, e.g., SSDs, which provide persistence, can be 100x-1000x slower than DRAM, which are volatile and lose all their data on a power cycle. Applications carefully place their working set data in DRAM where the access latency is between 50-100ns, and every so often, perhaps under the guidance of the user via policies, save the data from DRAM to the storage tier. Over

the past three decades, applications and operating systems have evolved to orchestrate this data movement in a highly efficient way.

Persistent memory (*pmem*) is a new type of random-access memory that offers persistence and byte-level addressability at DRAM-like access speed [19]. Intel[©] Optane[™] DC Persistent Memory [12] is an example of a readily available persistent memory product. Persistent memory is now becoming increasingly available in servers [1]. Operating systems such as Linux have had support to use *pmem* as faster storage disks for some time now [6].

The obvious way to consume *pmem* is as a faster storage device (block mode access) by running an unchanged application on top of a file system. We demonstrate a significant improvement in the throughput of Redis when it is run on top of Linux ext4 filesystem using persistent memory as the block storage device. But, we were also able to show even further performance improvements with a Redis that was hand modified to use byte-addressable persistent memory for its in-memory database. Unsurprisingly, we were also able to show that a good bit of the application's I/O processing code that store the data from volatile DRAM to storage, can now be retired as data is being immediately persisted all over the application. So, using the byte-addressable mode of *pmem* delivers more performance than using it in block mode, and also lowers the overall complexity in the application code. We delve deeper into this in §2.

We argue that databases like Redis are fast and popular because they highly optimize the data structures used in volatile memory. However, applications must write these data structures into serial buffers for persistence and these optimizations are lost. With byte-addressable persistent memory, applications can directly persist and retrieve their data structures without serialization. And so, the main focus of this work is that we strive to make manipulating persistent memory similar to manipulating volatile memory.

Towards this, we contribute *go-pmem*, an open source extension to the popular Go programming language. *go-pmem* provides a familiar Go-esque programming model to the appli-

*Co-first authors ordered alphabetically

cation developers to use persistent memory at byte-level granularity. We arrived at our current model by implementing a feature-poor implementation of Redis (*go-redis-pmem*) (§5.3) that directly uses byte-addressable pmem. With *go-pmem*, we contribute a programming model with the following design goals:

1. Single type system. No separate persistent types as in [18, 32].
2. Pointers remain unchanged, i.e., no fat pointers. A consequence of this is we implement pointer swizzling (§4.3.2).
3. Support two heaps, volatile and persistent: *go-pmem* extends Go runtime to manage pmem and uses pointers to identify objects in pmem.
4. Allow pointers both across and within the persistent and volatile heap, but make system safe across crash and recovery. To do this, *go-pmem* extends Go's garbage collector (GC) to work across both the heaps.
5. After restart, allow applications to retrieve back data stored in pmem by associating it with a string name. These are called named-objects (§4.4.1).
6. Allow transactional code blocks by requiring the user to demarcate these with a new Go *txn* keyword.
7. Reuse functions to operate on data in volatile or persistent memory.
8. Allow allocation and update of pmem resident data structures outside a transaction as long as they are not reachable from a *named object*. In case of a crash-and-recovery, *go-pmem*'s GC will garbage collect such objects thus avoiding any persistent memory leaks.

In a set of microbenchmarks, we see that *go-pmem* performs up to 40x better than other pmem libraries languages. *go-redis-pmem* offers 5x more throughput than unmodified Redis on an SSD against memtier benchmark and restarts up to 20x faster than unmodified Redis. We explain our evaluation methodologies more in §6. To the best of our knowledge, ours is the first expansive effort to change Go to support persistent memory. In §5, we explain how Go's existing design features helped and challenged us towards our idea of supporting pmem.

go-pmem is developed from the Go 1.11 code base, and is fully open-sourced. Links to the respective repositories can be found at <https://vmware.github.io/persistent-memory-projects/>.

2 Background

To give more context into our work, we begin by asking why should anyone care about persistent memory?

2.1 Experiments with Redis

Over the years, a lot of work has gone into optimizing data intensive applications. To reduce the latency of data access, these applications keep frequently used data in DRAM, and flush the dirty data from DRAM to disks/SSD at certain well-

defined points. For example, Redis [24] allows users to persist their data to disks/SSD in an append-only-file (AOF). But if no persistence mode is used, a crash at any point would cause all the in-memory data to be lost. If the user cannot afford any data loss at all, he has to write to the AOF file after each write request and as such sees significantly reduced throughput.

In all the experiments in this section, we run Redis in the zero data loss mode, as that is the fair way to compare with byte addressable persistent memory which provides zero data loss. To see the benefits of persistent memory, we now conduct the following experiment:

1. Use the memtier benchmark as the load generating input workload [25], configured to issue read-write requests in a 70:30 ratio.
2. Run unmodified Redis-3.2, in the following two modes:
 - (a) saving the AOF on an SSD and
 - (b) saving the AOF on a persistent memory device.
3. Run Redis-3.2 hand modified for byte-addressability taken from [11].

Figure 1 shows that just by running unchanged Redis on a pmem device as block storage, the throughput increases by up to 4x. This configuration gives a higher throughput owing to the inherent faster access time of persistent memory. Figure 1's *Pmem Block IO* curve confirms this. The third curve (PMDK-Redis) [11] shows the performance of a Redis server modified to write data to persistent memory used in byte-addressable mode. In fact, PMDK-Redis even outperforms Redis running on pmem as block-storage.

To understand this, we perform another experiment. We run unmodified Redis-3.2 once with AOF disabled and once with *always-fsync-AOF* option on *nullfsvfs* virtual file system [2]. This file system treats all read and write system calls to the storage media as no-ops. So the difference in performance is entirely due to serializing data in the application and the overheads in making system calls. This is the reason why in Fig. 2 Redis with AOF enabled on *nullfsvfs* is slower than Redis with AOF disabled. Because pmem is faster than SSD, the time spent accessing the device is significantly reduced and the overhead of the software stack become a significant portion of the overall application latency.

These two experiments convince us that using persistent memory can give significant throughput improvements in data intensive applications and that the most efficient way for applications to access persistent memory is through direct CPU load/store instructions bypassing any file-system/PCIe overheads. A more extensive study on persistent memory can be found in [37]. Since memory is accessed in 64-byte cache lines, the CPU reads only what it needs to read, instead of rounding every access up to a block size, like storage. Linux also allows persistent memory to be used in a direct-access (DAX) mode [6, 20, 21]. This mode allows users to mmap files in persistent memory into their virtual address space and access it through loads/stores, bypassing the OS page cache

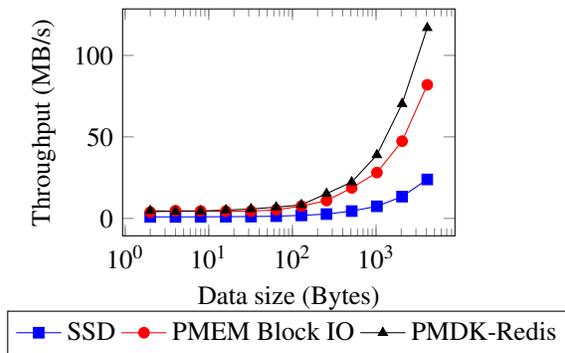


Figure 1: Redis throughput comparison against memtier

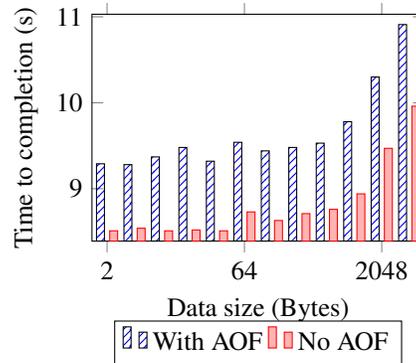


Figure 2: Redis-server runtime on nullfsvfs

and file systems. In the rest of this work, we use pmem in *byte-addressable* mode, unless we specifically mention block storage mode.

2.2 Why Change a Programming Language?

Hand porting applications to use byte-addressable pmem can introduce fragility in the code. Just think of a situation, where the developer misses to guard a single store to persistent memory within a transaction - the bug will be very difficult to discover. So, the next question is: how can the porting process be made easier? Ideally, any acceptable solution should have the following features:

1. Be similar to existing programming models.
2. Work transparently for systems not supporting persistent memory. I.e., we want to be able to write functions that can operate on data in persistent memory or volatile memory.
3. Fast execution time.

One approach is via ad hoc libraries. As we will discuss in §3 and §4, these libraries expose a programming model different than existing programming models for volatile memory. In particular, memory management becomes tricky and these solutions either don't provide a simple and complete programming model, or go through complicated steps to keep it simple. We argue that programming languages already manage volatile memory. So, persistent memory which is a special type of memory and is also byte-addressable, should also be managed by programming languages.

Such a language should at least provide:

1. Persistent memory allocations on heap
2. Garbage collection of persistent heap objects
3. Support modifying persistent memory in a crash-consistent way
4. Support recovery from crashes, i.e., include support for reverting inconsistent updates.
5. Not require offline processing of persistent data regions
6. Similar to existing programming models, for easier adoption.

2.3 Why Go?

We chose Go because it is a high-level managed language with an easily extendable runtime. Moreover, we use some of Go's design to our advantage. For example, we reuse Go's mark-and-sweep garbage collector to garbage collect pmem. Go uses static escape analysis to determine the scope of objects at compile time and intelligently places objects on the stack or the heap. We extend this to track accesses to volatile and persistent heap and prevent unnecessary pmem accesses. Additionally, the potential benefits of high-level languages are well understood. Automatic memory management in HLLs like Go reduce programmer effort and use-after-free bugs. These kinds of bugs are more dangerous with the use of pmem because any memory leak will survive crashes/restarts. Go's type-safety helped us to avoid special data types and invalid memory accesses. Go also has an active developer community with increasing adoption in systems community (e.g. popular software such as Kubernetes, Docker etc. are written in Go) and this fits well with our plan to open-source our changes.

3 Related Work

Most of the previous efforts to program persistent memory fall into two categories:

1. Ad hoc library to support pmem. This library usually allows special objects to be created/updated/destroyed in a crash-consistent way through special APIs [17, 18, 30].
2. Programming language enhancements to manage pmem and instrument user code with transactions [26, 27, 32, 45, 46].

In section 6, we compare go-pmem's performance with works in both categories, PMDK [18] (a library) and Mnemosyne [45], Makalu [26] (language changes).

Previous efforts often provide a new allocator for pmem and require the user to free memory allocated in pmem [7, 18, 26]. Either they don't handle leaks in persistent memory [45] or use special objects and data types to maintain reference counts [26, 30, 32] for garbage collection. Often they provide offline

Library (language)	Model	References	Transactions			Heap	
			Semantics	Implementation	Fn calls	Growth	Reloc.
go-pmem (Go)	Compiler support, Library	Direct pointers	Explicit - user tagged	Undo logs	Yes	Yes	Yes
PMDK [18] (C,C++)	Library	Fat pointers	Explicit - user tagged	Undo logs	Yes	No	Yes
Mnemosyne [45] (C)	Compiler support, Library	Direct pointers	Explicit - user tagged	Redo logs	No	No	No
Makalu [26] (C)	Library	Direct pointers	N/A	N/A	N/A	No	No
Atlas [27] (C)	Compiler support, Library	Direct pointers	Implicit - using locks	Undo logs	No	No	No
Autopersist [43] (Java)	Compiler, JVM support	Direct references	Explicit - user tagged Implicit - durable roots	Undo logs	Yes	No	No
Espresso [46] (Java)	Compiler, JVM support	Direct references	Explicit - user tagged	Persistent objects	Yes	No	No
iDO [38] (C,C++)	Compiler support	Direct pointers	Implicit - using locks	Atomicity via resumption	No	No	No
JUSTDO [36] (C,C++)	Library	Direct pointers	Implicit - using locks	Atomicity via resumption	No	No	No

Table 1: Table comparing features of various pmem libraries

tools to check pmem file for garbage [18, 26]. Some require that users must do pmem memory allocations within transactions [18]. To access data in pmem they usually provide fat pointers [18, 30] or allow direct pointers and ignore pointer swizzling [26, 46]. We did not find any previous work that handled growing persistent heap at runtime. In section 6.4.2 we discuss why this is important.

[31, 45] also provide concurrency through transactions. Of course, transactions for programming languages has been a well-researched topic [34]. However, we don't pursue transactions as a way of solving concurrency, but more from a point of view of crash-consistent/atomic updates while minimally getting in the way of Go's existing concurrency paradigms. In this regard, we are similar to [32, 43] but are more flexible (see section 4.5).

Similar to [27, 32] we use a modular SSA pass to inject undo transaction logging statements to user code. But unlike [32] we do not ask the user to provide special pragmas for faster execution. We believe these optimizations complicate the programming model. We also allow function calls within transactions unlike [45] and want the functions to be reused for persistent and volatile data.

Another design point is how to demarcate the transactional code block. For example, Atlas [27] uses existing locks and constructs a happens-before graph to determine order of logs and updates. Autopersist [43] on the other hand, is quite differ-

ent. Like go-pmem, they allow users to explicitly demarcate transactional code blocks. But, they also allow users to write normal Java code and when they point to a volatile data from a pmem root, Java runtime moves the transitive closure of the volatile data to pmem transactionally. go-pmem allows updates to pmem-resident objects outside transactions as long as they are not pointed to by a named object.

So far, most of these efforts have been confined to C, C++ with some work in Java [17, 43, 46] and OpenJDK [15]. There have been calls for these programming languages to support persistent memory [4] but we don't know of any concrete changes yet. Table 1 presents a concise summary comparing the features of various pmem libraries. *Fn calls* captures whether the transaction semantics allows function calls inside a transaction. *Heap Growth* indicates whether the library supports a growable heap design and *Heap Reloc.* states if the library supports relocating the pmem heap on an application restart.

4 Design

The design of go-pmem is driven mainly by two considerations:

1. The changes should be accepted to the Go language. This translated to reusing existing Go compiler techniques and keeping our changes to a minimum.
2. Provide a familiar programming model to developers

that has a minimal interface and is congruent with the existing Go infrastructure. This would help in easier acceptance and adoption by the Go community.

4.1 Programming Model

Listing 1 shows how to add a new node to a linkedlist resident in pmem using go-pmem. We have highlighted how this is different than normal Go code adding a node to a linkedlist in volatile memory. Memory in pmem is allocated using *pnew*, similar to Go's *new* and all updates are made transactional using codeblock *txn("undo")*. We argue that this is very similar to how Go code is written today. How we managed to get a programming model like Listing 1 is discussed in the rest of this section.

```

1  package main
2  import "pmem"      // <-
3  import "transaction" // <-
4
5  // add new node to tail; return new tail
6  func addNode(tail *node) *node {
7      n := pnew(node) // <-
8      txn("undo") { // <-
9          mutex.Lock()
10         n.prev = tail
11         updateTail(tail, n)
12         mutex.Unlock()
13     } // <-
14     return n
15 }
16
17 func updateTail(tail, n *node) {
18     txn("undo") { // <-
19         tail.next = n
20     } // <-
21 }

```

Listing 1: Add node to a linkedlist in pmem

4.2 Language Constructs

We have added two new APIs to Go semantics to support persistent memory allocations:

```

func pnew(Type) *Type
func pmake(t Type, size ...IntType) Type

```

Just like *new* [8], *pnew* creates a zero-value object of the *Type* argument in pmem and returns a pointer to this object. The *pmake* API is used to create a slice in pmem. The semantics of *pmake* is the same as the *make* API in Go.

4.3 Runtime Design

Go runtime uses datastructures such as *mcache*, *mspan*, *mcentral*, *mheap*, etc. to store the metadata related to the heap. A span is a contiguous region in memory (one or more pages) from which the allocator allocates similar-sized objects. *mspan* stores metadata about a span. *mcache* is used to cache spans at a thread level. If an allocation request can be satisfied using the cached span, then it can be done so without acquiring any locks, making such allocations very fast. If not,

a new span is obtained from the *mcentral* or *mheap*. *mcentral* is a central store of small spans (object size <= 32K) and *mheap* is a central store of freed spans and large spans.

The heap is managed in arenas of size 64MB. Each arena data-structure stores the following metadata:

1. Span table - Span table is an array that holds a reference to the *mspan* object corresponding to that virtual page index.
2. Heap type bitmap - The heap type bits are used by the GC to identify what regions in memory have pointers in them. The GC uses these bits while walking the heap. The heap type bits corresponding to an object is set by the allocator when it allocates that object.

Linux exposes pmem to applications as a file. Byte-level load/store access to pmem is available once a pmem file has been *mmap*'d to an application's address space [33]. We do all this in Go runtime and abstract these out through a Go package called *pmem*. We have incorporated a simple approach to enabling pmem support in Go - one with minimal changes to the design of the existing memory allocator and garbage collector (GC). Rather than make all the runtime data-structures crash-consistent, we log additional metadata to keep track of allocated regions. This saves us from extensive code changes that would have been necessary in the runtime and aids in the implementation of a robust design. A novelty of our approach is the minimal amount of additional metadata that we log.

4.3.1 Growable Heap Design

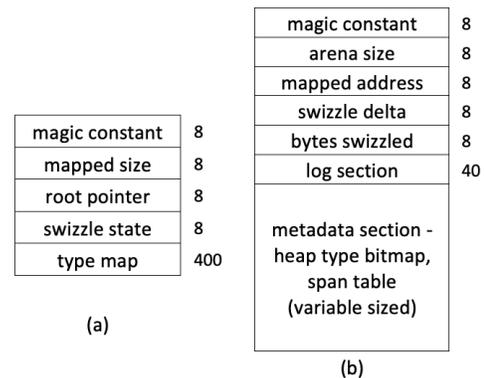


Figure 3: (a) pmem file header (b) arena header layout. Storage space in bytes adjacent to each field.

The runtime maps the pmem file into memory in arenas of sizes that are a multiple of 64MB. Figure 3(a) shows the layout of the global header stored in the beginning of the pmem file. *magic constant* is an 8-byte random number which helps to distinguish between first and subsequent initialization of persistent memory. *mapped size* is the size of the file currently mapped. *root pointer* is used to store the pointer to the application named objects (§4.4.1). *swizzle state* is used for

implementing pointer swizzling (§4.3.2). *type map* is used to cache a fixed number of data types which are the most frequently allocated, helping pmem allocations of such types to be completed significantly faster (§5.1.5).

Each arena is divided into two sections - the arena header section and the region managed by the allocator. Figure 3(b) shows the layout of the header section in each arena. *arena size* is the size of the arena and *mapped address* stores the address at which the arena is currently mapped. *swizzle delta* and *bytes swizzled* help to implement the pointer swizzling algorithm. The *log section* helps to implement a minimal undo log in runtime which is used only during swizzling. The *metadata section* stores the runtime metadata for this arena as mentioned in §5.1.3. Whenever runtime runs out of persistent memory space, the persistent memory file is grown to accommodate a new arena. The metadata in the global header is updated in a consistent manner to reflect the addition of this new arena.

4.3.2 Pointer Swizzling

Pointer swizzling is a powerful feature that allows direct pointers to be stored in persistent memory and dereferenced, even after multiple invocations of the application. Since persistent memory is exposed through files mapped into memory, the virtual address of the mapping can change during each invocation. In the common scenario, go-pmem is able to map arenas at the same address, avoiding the need to swizzle pointers. If the mapping address changes, then all pointers stored in the pmem heap becomes garbage. Pointer swizzling is the process of 'fixing' these pointers by re-writing them with their new mapped address. Some libraries such as PMDK work around this problem by using a *base, offset* pair object as a reference to an object in persistent memory. Dereferencing such an object involves a hashmap lookup and offset computation which can get expensive.

Swizzling is done during pmem initialization in a per-arena manner. The algorithm uses the *swizzle delta* field in arena header to store the offset by which pointers that point into this arena should be changed. *bytes swizzled* help track how many pointers have already been swizzled. The *log section* in the arena help update pointers transactionally. The swizzling algorithm is resilient to crashes during swizzling. If a crash occurs, any partially executed swizzling is completed on the next run, before swizzling all pointers to the new mapped address. go-pmem uses a parallelized algorithm to swizzle pmem arenas. As an added advantage, the swizzle algorithm also implements pointer safety. On application restart, any pointer stored in the pmem heap that point outside the pmem heap are garbage. The swizzle algorithm zeroes out any such pointers to ensure applications do not access such rogue pointers. This gives users the freedom to store both pmem and volatile pointers in the pmem heap.

4.4 Restarting After a Crash/Exit

Our design currently handles graceful exits and non-corrupting failures. Listing 2 shows code starting/recovering from a crash. We provide a *pmem* package that handles initializing pmem and restarts.

```
1 package main
2 import "pmem"
3 func init() {
4     firstInit := pmem.Init("database")
5     var head *node
6     if firstInit {
7         // Create a named object called root
8         head = (*node)(pmem.New("root", head))
9     } else {
10        // Retrieve the named object "root".
11        // One-line restart!
12        head = (*node)(pmem.Get("root"), head)
13    }
14 }
```

Listing 2: Code for start/restart of application

4.4.1 Roots/Named Objects: pmem Package

Any data in volatile memory is lost on restart, so volatile pointers pointing to data in pmem will be lost too. This means only pointers residing in pmem pointing to data in pmem can be used to access pmem-resident data after a restart. We allow the applications to retrieve these pointers through string names. These can then be used to navigate other objects stored in pmem. We call these objects "*named objects*" and they can be pointers to native types, structs, or Go slices. Any updates to these named objects must be made through *pmem* package APIs as shown in Listing 2.

4.5 Transactions as a Part of Go: txn Block

To make sure no data is left in an inconsistent state, we use transactions. We change Go compiler to natively support undo transactions. We pursue transactions in Go with the intention of providing crash consistency and durability for updates to data in pmem. We introduce a new keyword to Go called *txn* that automatically intercepts stores to pmem and logs them in an undo log. We add a new SSA (Static Single Assignment) pass to Go compiler's backend that injects statements to Go's intermediate representation of the user code. Our design is derived from a similar technique used in Go compiler to add write barriers for garbage collection to stores in volatile heap [14]. Our new *LogStore* SSA pass comes after most of the existing Go SSA passes, so we don't lose on the existing optimizations. For example, successive stores to a pmem resident location can be eliminated by Go's *deadstore elimination* SSA pass even after our changes.

To demarcate transactions, we require users to contain their code within a *txn("undo")* code block. The "*undo*" indicates we currently support automatic code generation only for undo logging. We briefly discuss how we provide typical transaction properties below.

1. Atomicity: The `pmem.Init()` call (see Listing 2) initializes `pmem` and reverts any incomplete updates stored in the transaction logs of an application in case of a restart.
2. Consistency: We rely on the user to explicitly demarcate updates to `pmem`-resident data by using a `txn` block around the code.
3. Isolation: Simultaneous transactions accessing common data must not see any updates till a transaction is committed. We do not support isolation through software transactional memory but rely on programmers to use Go's mutex locks for critical sections. Our concurrency model is simple:
 - (a) All the locks acquired within a transaction must be released within the transaction.
 - (b) All the locks acquired outside a transaction must be released outside the transaction.

Our design then makes sure that all the updates to shared data structures are visible only at the end of a transaction. This is achieved by delaying unlocking any locks acquired within a transaction until the end of the transaction. This is similar to the 2PL locking strategy used in database transactions [40].

4. Durability: All the changes made to data in `pmem` are made durable at the end of a transaction by flushing relevant data stored in the processor caches or buffers.

```

1 // txn block of addNode()
2 tx.Begin() // <-
3 mutex.Lock()
4 tx.Log(&n.prev) // <-
5 n.prev = tail
6 updateTail(tail, n)
7 tx.End() // <-
8 mutex.Unlock() // <- generated after End()
9
10 // txn block of updateTail()
11 tx.Begin() // <-
12 if inPmem(&tail.next) { // <- extra check
13 tx.Log(&tail.next) // <-
14 } // <-
15 tail.next = n
16 tx.End() // <-

```

Listing 3: Compiler generated code for listing 1.

We point out a couple of limitations that our locking model introduces -

1. Multiple `lock()` and `unlock()` operations on the same lock do not work inside a `txn` block.
2. Holding all locks taken inside a transaction until the end of the transaction can make lock-based critical sections in `go-pmem` slower than other models such as Atlas [27] that allow dependent transactions to run concurrently (as soon as required locks get unlocked). In order to provide isolation, they capture transaction dependency between multiple threads in their logs.

Listing 3 shows the compiler generated code of the `txn{}` code block in listing 1. Additional code added by the compiler to ensure transactional semantics is highlighted. Line 8 of listing 3 shows how a mutex unlock is delayed until the end of the transaction. `go-pmem` also allows function calls within a transaction, as explained further in §5.2.2.

4.6 No Persistent Data Types

`go-pmem` intentionally does not introduce new data types. Previous works have often introduced data types like `pmem`, `p<int>` or `persistent int`. They usually do this because they offer a library implementation and overload the assignment operator [18] or they want type-safety, reference counting etc. for `pmem` resident data [32]. We believe this complicates the programming model and instead rely on Go's in-built typesafety and garbage collection.

5 Implementation

`go-pmem` adds about 4000 lines of code and removes 300 lines of code from Go runtime, excluding code documentation. This does not include the code in the `CPUID` package from Intel [10] that is used by runtime to identify CPU features for flushing CPU caches. The two Go packages (`pmem` and `transaction`) took close to 2300 lines of Go code. These changes were made on top of Go 1.11 release and don't include the code for testing these changes. Our implementation currently works only for 64-bit Linux 4.15 and above.

5.1 Runtime Details

5.1.1 Data Structure Support

To support persistent memory allocations, runtime datastructures such as the `mspan`, `mcache`, `mheap`, `mcentral` were extended to store persistent memory metadata. The `mspan` structure was augmented to identify if this is a `pmem` span or volatile memory span. Similarly, `mcache`, `mcentral`, and `mheap` were doubled in size to store `pmem` spans separately. It should be noted that no Go runtime data-structures are stored in `pmem`. Instead, minimal additional metadata is logged in `pmem` arena header to capture `pmem` state.

5.1.2 Memory Allocation

A persistent memory allocation request results in the following workflow - if a cached `pmem` span is available in the `mcache` with a free slot, it is used to satisfy the allocation request. Otherwise, a new `pmem` span is requested from the `mcentral/mheap`. If no `pmem` span is available, a new `pmem` arena is mapped to memory. The required span is then carved out from the `pmem` arena to satisfy the allocation request. Any required metadata is also logged (§5.1.3). An advantage of the `go-pmem` allocator is that these steps can be done without invoking transactions. This is because any `pmem` leaks are plugged by the GC during heap recovery (§5.1.4).

5.1.3 Metadata Logging

The metadata that is logged is the minimum amount of information that the runtime needs to reconstruct the memory allocator and garbage collector state on the subsequent execution of the application. The benefits of keeping the metadata to a minimum are twofold: we do not need to introduce complex transactions in the runtime to maintain consistency and the allocator performance is only slightly affected due to the additional logging.

Two kinds of runtime metadata are logged. The first is the GC heap type bits. Whenever the allocator sets heap type bits for a pmem object, it is also logged in the pmem arena header section. Like its volatile memory representation, heap type bits occupy 2 bits for every 8 bytes of heap data. In §5.1.5 we talk about an optimization that helps avoid heap type bitmap logging for frequently allocated data-types. The second is the span table. The span table captures information such as which spans are in use and the size class of the span. Span table logging happens when a new span is created by the allocator or freed up by the GC. Span table reserves 32 bits for every pmem page in the arena (Go uses a page size of 8KB). For a 64MB pmem arena, the arena header section occupies 2024KB, and 63512KB is available for the allocator. The arena section includes 80 bytes for the header, 31756 bytes for the span table and 1984.75KB for the heap bitmap, making the pmem arena memory overhead as 3.09%.

5.1.4 Reconstruction

All runtime data structures related to memory allocator and garbage collector are stored in volatile memory. So, if an application crashes, all state information is lost. *Reconstruction* is the term used in our project to denote the process of bringing back the state of the memory allocator and garbage collector related to persistent memory as it was before the crash. The additional metadata stored in pmem aids in this *reconstruction* process. No persistent memory data is modified during reconstruction. Hence the reconstruction process is resilient to any crashes that happen while it is ongoing. Briefly, *reconstruction* works as following:

1. GC is disabled until *reconstruction* finishes so that it does not interfere with the *reconstruction* process.
2. Spans are recreated using the logged span metadata table.
3. The logged heap type bitmap is copied as-is to the arena metadata in volatile memory.
4. Pointers are swizzled, if necessary
5. GC is re-enabled

GC walks the *reconstructed* pmem heap starting from the named root objects. It marks all reachable objects as being in-use, and makes any leaked pmem objects available for reuse. This GC walk runs in the background, making the *reconstruction* process execute quickly.

5.1.5 Using Go's Typesystem to Optimize Pmem Allocations

An allocation of an object that has pointers in it results in the allocator setting the heap type bits for it. If this is a pmem object, these type bits are also logged in the pmem arena header. If a datatype is heuristically found to be a frequently allocated type, then that type is promoted to be cached specially in mcache to speed up allocations of such objects. The heuristic used is the following - the number of allocations of such an object has exceeded the number of slots available in a span corresponding to this object sizeclass and its allocation frequency is greater than 100 objects per second. A span specially so cached is used only to allocate objects of one type. This makes it possible that the heap type bits be logged only for the first object allocated, making further allocations from this span very fast. We employ the typemap (§4.3.1) in the global header to store what types have been specially promoted. Our design supports promoting up to 50 types to be specially cached. In our experience, the maximum number of types frequently allocated by various pmem applications were far fewer. Each type is represented in the typemap section in the pmem header using an 8-byte identifier, making typemap occupy a total of 400 bytes.

5.1.6 Undo Log Implementation

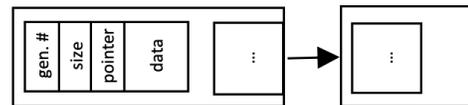


Figure 4: Undo log design

Undo logs are stored within a linked list of Go's byte arrays. Each undo log entry has the layout as shown in figure 4. *gen.#* stores current undo log generation number. On a successful abort/commit, the *generation number* is bumped up to mark all entries as no longer valid. *size* stores the size of the data logged. *pointer* is the address at which this data originally resided. *data* contains the logged copy of the data. Since we do not anticipate a transaction abort in the common case, log entries are populated using *movnt* instructions so that data is directly moved to the pmem device bypassing the processor cache.

We also employ a number of optimizations to make logging fast in go-pmem:

1. Empty transactions do not incur any runtime overhead as no cache flushes or memory fences are issued.
2. Logging the same object multiple times incur minimal overhead. We maintain a map to track what objects have already been logged.
3. As byte arrays are very common in Go, we specially optimize logging 1-byte data. We try to pack consecutive 1-byte data objects in a single log entry rather than create separate entries for each.

5.1.7 Working Around Go's GC to Optimize Undo Logging

Go uses a mark-and-sweep GC. It scans the heap in a breadth-first fashion, traversing the heap through the live pointers it finds. The scan starts from the pointers found in goroutine stacks and global variables. We want objects pointed at from the logged data to be kept alive until the transaction completes. But since data is logged in a byte-array, runtime no longer has the type information of each data item logged. Our initial logging library used Go's reflect package which gave a poor performance, so we decided to use Go's byte arrays. To identify pointers in the byte array, whenever a data is logged, all pointers within this data item is stored separately in an array of pointers residing in volatile memory. This ensures that GC finds these pointers while traversing the heap.

5.1.8 Cache Flushing

Data written to persistent memory can be guaranteed to be persistent only after they are flushed from the processor cache to the persistent memory media. The runtime provides the `PersistRange` API to flush the processor cache over the address range passed to it.

```
func PersistRange(addr unsafe.Pointer, len int)
```

If the persistent memory device supports direct-access, this function takes care of executing the most optimized cache flush instruction supported on the processor (such as `clwb`, `clflushopt`, or `clflush`) and any necessary memory barriers [42]. If the device does not support direct-access, then `PersistRange` invokes the `msync` system call to flush data at a page size granularity. Transactions in `go-pmem` automatically call into the runtime to flush data from the caches, freeing the programmer from having to do it manually.

5.2 LogStore SSA Pass

The LogStore SSA pass can automatically interpose stores to persistent memory and redirect this to an undo transaction. The user can wrap any codeblock with a `txn("undo")` keyword and engage this new SSA pass. In the absence of any `txn` block, this SSA pass does not do anything. Because we wanted to keep the changes to a minimum, this SSA pass can be plugged into/out of the Go compiler's usual workflow.

5.2.1 Handling Volatile Memory Access Inside Transactions

Go uses escape analysis [28] to figure out life time of variables, and thereby avoids unnecessary memory allocations to volatile heap. We extend this to avoid unnecessary allocations to persistent heap and track pointers in volatile heap. With this static analysis, we know the probable location of an update within the transaction. If this update can be proven to be a location in volatile memory, we do not do anything. Otherwise we store the current value of the data in a persistent log which will be replayed in case of a crash.

5.2.2 Handling Function Calls Inside Transactions

Go compiles each function independently and cannot know if this method will be called from a transaction, or a non-transactional code at compile time. Instead of cloning the function for transactional access, we maintain a per Go-routine handle and ask the user to wrap any potentially transactional code within a `txn` code block. Based on whether a transaction is already ongoing or not, we intelligently start a new transaction or continue the same transaction at runtime. In case this function is called from a non-transactional code, the function simply operates on volatile data without any side effects. Using the techniques mentioned in §5.2.1 we try to keep the performance overhead to minimum in this case.

5.3 Implementing Go-redis-pmem

We implement a multithreaded feature-poor redis server called `go-redis-pmem` written using `go-pmem`. `go-redis-pmem` currently supports storing/retrieving string KV pairs in `pmem` and can run traffic generated from `memtier` benchmark. It currently has 6800 lines of code across 11 files and 360 functions. As we were implementing `go-redis-pmem`, we realized that as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory. Our desire to support function calls and ability to reuse functions for data in volatile and persistent memory was driven by the implementation of `go-redis-pmem` and the ease this offered to the programmer.

5.4 Limitations

`go-pmem` is a work in progress and will continue to evolve as we gain more experience in programming applications for persistent memory. Below, we enumerate the current limitations of `go-pmem`:

1. No support for shrinking persistent heap file after they have grown. We believe an offline application working like a compacting garbage collector can fix this.
2. No support for allocating/transactionally using Go's maps or channels in `pmem`. `go-pmem` currently supports basic Go types, structs, and Go slices.
3. No support for working with multiple persistent heaps
4. No support for operating systems other than 64-bit Linux.
5. No support for redo/custom implementation of transactions with the `txn` code block.
6. We do not handle the case when there are traditional I/O operations (like network, display, etc.) inside a transaction and then there is a crash. Previous works have handled this by throwing an exception [34] and by providing safe IO [44]. We can do something similar.

6 Evaluation

We use the benchmarks from Computer Language Benchmarks Game (CLBG) [5] as the microbenchmarks to compare the performance of our memory allocator and transactions.

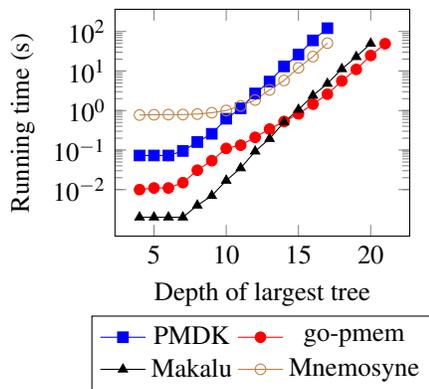


Figure 5: Runtime as depth of largest tree changes

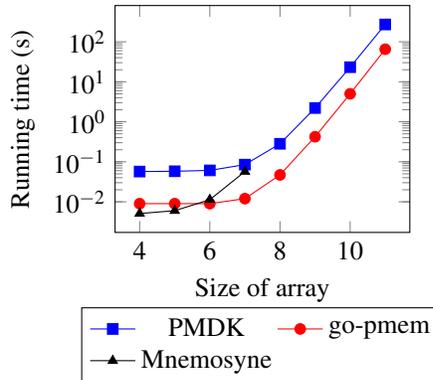


Figure 6: Runtime as num. of permutations in fannkuch-redux changes

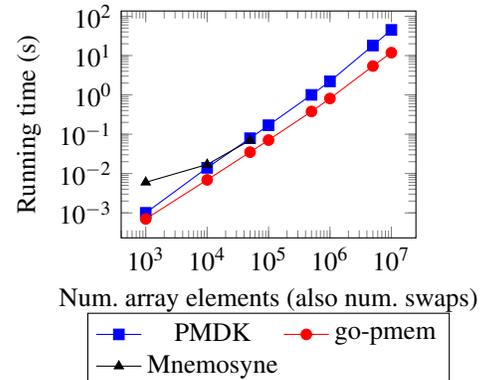


Figure 7: SPS Benchmark: Overhead of transactions

CLBG has gained wide attention for comparing the performance of different programming languages [39]. Go has also used some of these benchmarks to optimize their implementation in the past [23]. We extend some of these benchmarks to test how they perform when run on pmem, and don't focus on others as they test other language features such as arithmetic precision, hashtable performance etc. which is not our focus here. To test multicore scalability of go-pmem, we use the microbenchmarks from the Phoenix suite [41]. The phoenix suite was originally written to evaluate the MapReduce model for multi-core systems. We use the *pthread* version of these benchmarks to port to various pmem libraries. We also compare the performance of *go-redis-pmem* with other Redis implementations. In these evaluations, we ensure there is no remote persistent memory traffic by keeping only one CPU socket on. We compare our work against PMDK stable version (1.7 release commit *bc5e30948*) and we write code in C++ using C++ bindings from PMDK (stable version 1.8 commit *ab4ff69b7*) [22]. The Mnemosyne examples build on implementation code from [7] and Makalu examples run on implementation code from [16].

6.1 Experimental Setup

Our system is a 24-core Intel Cascade Lake machine with hyperthreading disabled and only one socket to avoid remote pmem traffic. It has 4 Intel[®] Optane[™] DC Persistent Memory Module, each of 128GB, and 64GB of DRAM. In all the runs, the deviation observed across runs was <2%. We report the average runtime across 3 runs.

6.2 Change in Compile Time

The compilation time of Go source code increased by 3.4% (from 42.71s to 44.16s) because of all our changes to Go compiler. In the compilation of *go-redis-pmem* (which has 6800 lines of code and 11 files), we did not see any noticeable difference in the compile time with and without the new ssa pass. The difference was <1% (0.71s vs 0.713s). These numbers were obtained for a fresh compilation with go's build

cache cleaned. With the build cache enabled, the observed difference was even smaller (less than 1ms).

6.3 Memory Allocator Performance

The Binary Tree allocator microbenchmark from CLBG stresses the pmem allocator by creating several perfect binary trees. One of these stresses memory to see maximum memory available. One is a long-lived binary tree and there are several short-lived trees which are created and then deallocated. Figure 5 compares the performance of go-pmem's pmem allocator to PMDK, Mnemosyne and Makalu. We note that because go-pmem can garbage collect pmem we did not have to free any tree nodes. Makalu and go-pmem do not write to pmem for each new allocation and so are at least an order of magnitude faster than PMDK and Mnemosyne. PMDK must do all allocations and deallocations within a transaction and performs the worst.

6.4 Performance of Transactions

6.4.1 Long Running Transactions

We use *fannkuch* from CLBG and *sps* to model programs with a long-running transaction. *Fannkuch* takes a byte array of size n and shuffles around elements for all $n!$ permutations of this array. We model all this to happen inside a single transaction. This lends it the behavior of a long running transaction. We also noticed that both PMDK and go-pmem use undo logs and maintain minimal state in pmem (only the oldest value of the array) whereas Mnemosyne uses redo log and quickly starts to use a lot of pmem. The *sps* microbenchmark has been used previously [30, 31] to report throughput of transactions. *sps* randomly performs swaps between entries of an integer array. The number of these swaps is equal to the number of elements. We change the number of elements in this array from 1k to 10 million and do all the swaps within one single transaction. So unlike *Fannkuch*, the size of data being operated on also increases as the transaction becomes longer. Figures 6 and 7 compare the running time of *Fannkuch*

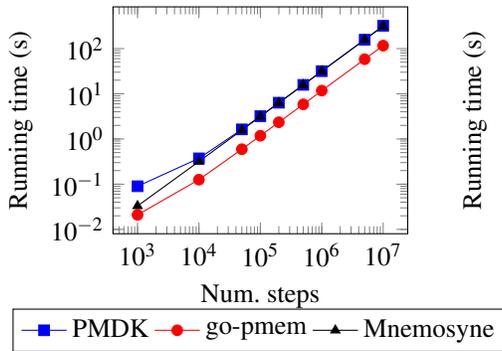


Figure 8: n-body: Runtime as num. of steps a planet moves varies

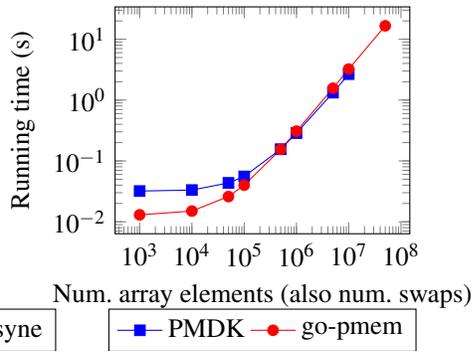


Figure 9: Tx recovery time after abort at the last swap

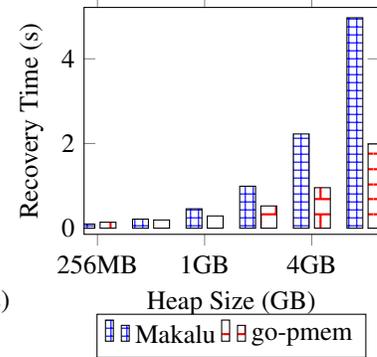


Figure 10: Restart time comparison using fill-heap

and *sps* for PMDK, Mnemosyne and go-pmem. We note that Mnemosyne curves stop early as the public implementation does not support large amounts of data within a transaction. Similar results for Mnemosyne were reported in [31]. The PMDK curves stop early as the default PMDK implementation does not allow creating a pmem file of 2GB or larger. For both these cases, we consistently outperform PMDK by 3-4x as the transactions become larger.

6.4.2 How Much Data do Applications Store to pmem?

We also want to highlight that the existing programming models are inept for long-running applications like the ones we model above. PMDK asks the user to specify the pmem file size at the beginning. It crashes if this size is exceeded. Mnemosyne always creates a pmem file of fixed size. We think it is very difficult to predetermine the pmem capacity that commercial applications will use. With the ability to grow pmem heaps (§4.3.1), our programming model is more flexible.

6.4.3 Several Short Transactions

We use *n-body* microbenchmark from CLBG to model a program with several short transactions. For short transactions, the overhead in setting up the transactions is not amortized, and we try to capture this overhead here. *n-body* models the orbits of planets using an algorithm. The input is the number of steps that the planets move from a starting point and the output is the new coordinates of the planets. We change one step movement of the planets to be one transaction. Figure 8 shows the running time as we vary the number of small transactions. go-pmem consistently performs 2-3x faster than PMDK and Mnemosyne even as the number of small transactions in the application increase to 10 million.

6.5 Multicore Scalability

We run all 7 benchmarks from the Phoenix 2.0 suite [41] to evaluate how go-pmem scales on multithreaded benchmarks. We modified these benchmarks to keep the data manipulated by each thread in pmem. Figure 13 captures the relative run-

ning time of PMDK compared to go-pmem. All benchmarks use 24 threads and run on the largest input configuration provided by Phoenix. Unlike PMDK and go-pmem, Mnemosyne stores more data in its redo logs and we were not able to get it running on any of these benchmarks. go-pmem scales much better than PMDK on all benchmarks other than *linear regression*. *Linear regression* stores very little data on pmem and incurs minimal transactional overheads. The benchmark *kmeans* uses 2D arrays in pmem and is significantly slower for PMDK. PMDK uses fat pointers and accessing 2D arrays requires multiple indirections. Although figure 13 shows results with 24 threads, we ran the benchmarks varying the number of threads from 2 to 24. The results with different thread counts are similar to the results shown in figure 13.

6.6 Restart Time Comparison

6.6.1 Undo Transaction Recovery Time After a Crash

We reuse *sps* benchmark from §6.4.1 to measure time spent by undo transactions in PMDK and go-pmem to revert back to consistent application state. We change the number of integers swapped in *sps* and crash at the last integer swap. Figure 9 shows go-pmem performs at par with PMDK when the amount of data to recover is less but gets slower by 20% as the amount of data to recover increases. We have not optimized the recovery path too much as this is not the common path. One obvious optimization is to store application data in cache-line aligned chunks. This will reduce the number of writes to pmem when we write back the consistent data during restart. Optimizing this path without affecting the performance of common cases remains on our future agenda.

6.6.2 Restart Time of Persistent Heaps

We use *fill-heap* benchmark from Makalu [26] to compare the restart time as the size of the persistent heap changes. *fill-heap* creates 64-byte objects to fill up a specified heap size. It makes half of these objects reachable from the pmem root objects. On restarting the *fill-heap* application, we measure the time taken by go-pmem, PMDK, and Makalu to recover

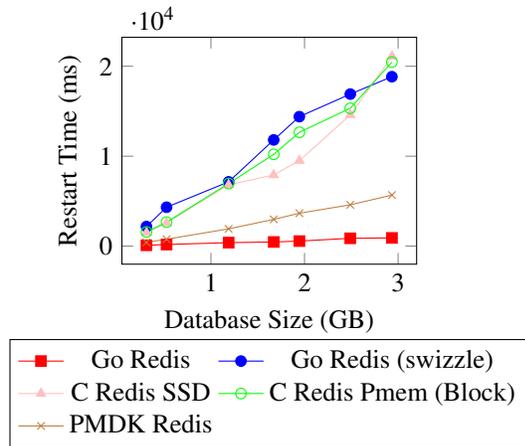


Figure 11: Comparing restart time of various Redis versions

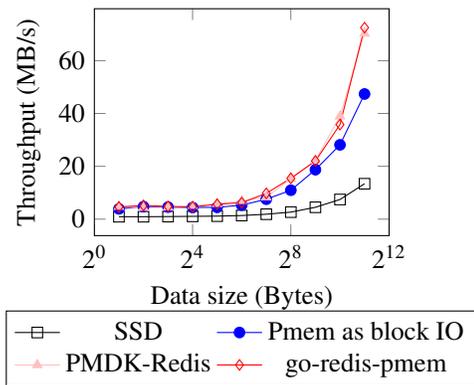


Figure 12: Redis throughput comparison against memtier benchmark

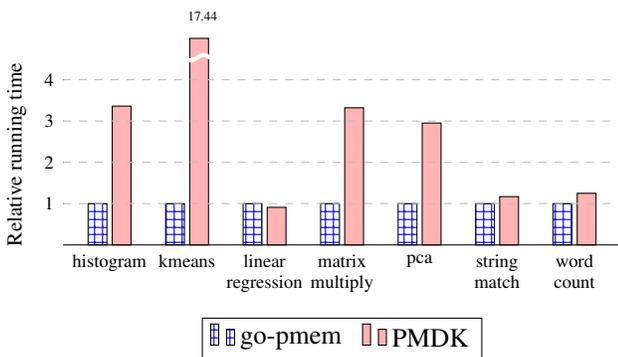


Figure 13: Relative comparison on Phoenix benchmarks

the pmem heap. As seen in figure 10, go-pmem recovers the pmem heap much faster than Makalu as Makalu has to go through an expensive offline GC phase. PMDK recovers almost instantaneously, because their allocator is inherently transactional, and hence incurs minimal startup cost.

6.6.3 Restart Time of Redis Variations

To measure how go-pmem heap’s recovery fares on a pmem application, we compared the restart time of go-redis-pmem against various other Redis configurations as shown in figure 11. Go Redis swizzle measures the cost of swizzling pointers by force mapping all arenas at a different address than where it was originally mapped. C Redis SSD persists its data as an AOF file on SSD, whereas C Redis Pmem block persists the AOF file on pmem used in block IO mode.

6.7 Go Benchmarks

We run a set of 4 macro-benchmarks used by Go community to monitor Go performance regressions as new features are added to the compiler [9]. These benchmarks stress the memory allocator, GC, compiler, etc. Table 2 compares perfor-

Benchmark	go-1.11	go-pmem	delta
Build-24	23.9s	23.9s	0%
Garbage-64	23.4ms	23.6ms	0.87%
JSON-24	84.0ms	84.3ms	0.44%
HTTP-24	73.9μs	75.5μs	2.15%

Table 2: Go benchmark comparison

mance of go-pmem versus Go-1.11 upon which our changes are based on. Our changes add little to no performance difference in these benchmarks.

6.8 Go-redis-pmem

Figure 12 shows the throughput of go-redis-pmem on the same memtier benchmark used in figure 1. Even though go-redis-pmem is multithreaded this configuration uses one client thread for a fair comparison. We can see that go-redis-pmem matches the performance of pmem-redis reconfirming our observation that if applications use persistent memory in byte addressable mode, they will perform the best. In data not shown here, we did see that go-redis-pmem performed better than pmem-redis and redis-3.2 when there are multiple client threads because it is multithreaded.

7 Conclusion

In this work, we presented go-pmem, an opensource extension to the Go language that allows programmers to develop pmem applications in Go. go-pmem is a natural extension of Go for pmem support, following the normal idioms of the Go language. We present a simple programming model and demonstrate its effectiveness by developing a feature-poor Redis equivalent key-value store in Go.

Acknowledgements

We thank the anonymous reviewers and our shepherd Changhee Jung for their invaluable feedback.

References

- [1] Available first on google cloud: Intel optane dc persistent memory, google cloud blog. <https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory>.
- [2] A black hole file system that behaves like /dev/null. <https://github.com/abbbi/nullfsvfs>.
- [3] A brief retrospective on transactional memory. <http://joeduffblog.com/2010/01/03/a-brief-retrospective-on-transactional-memory/>.
- [4] A call for a data persistence study group. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1026r0.pdf>.
- [5] The computer language benchmarks game. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.
- [6] Direct access for files, kernel.org. <https://www.kernel.org/doc/Documentation/filesystems/dax.txt>.
- [7] Gcc port of mnemosyne. <https://github.com/snalli/mnemosyne-gcc/tree/cfed43142cdcb5175f1b7c75cd6a922ce561060e>.
- [8] The go programming language, google. <https://golang.org>.
- [9] Golang benchmarks. <https://github.com/golang/benchmarks/>.
- [10] Intel corporation. cpuid library for go programming language. <https://github.com/intel-go/cpuid>.
- [11] Intel corporation. pmem-redis. <https://github.com/pmem/redis>.
- [12] Intel optane™ technology, intel. <https://www.intel.com/optane>.
- [13] Intel vtune profiler, intel. <https://software.intel.com/en-us/vtune>.
- [14] Introduction to the go compiler's ssa backend. <https://github.com/golang/go/tree/master/src/cmd/compile/internal/ssa>.
- [15] Jep 352: Non-volatile mapped byte buffers, openjdk. <https://openjdk.java.net/jeps/352>.
- [16] Makalu : Nvram memory allocator. https://github.com/HewlettPackard/Atlas/tree/makalu/makalu_alloc.
- [17] Persistent collections for java. <https://github.com/pmem/pcj>.
- [18] Persistent memory development kit. <https://pmem.io/pmdk/>.
- [19] Persistent memory documentation. <https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk>.
- [20] Persistent memory in linux, snia. https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Coughlan_Tom_PM_in_Linux.pdf.
- [21] Persistent memory wiki. <https://nvdimm.wiki.kernel.org/>.
- [22] Pmdk c++ bindings. <https://github.com/pmem/libpmemobj-cpp>.
- [23] Programs from "the computer language benchmarks game", used to be in the main go distribution in test/bench/shootout. <https://github.com/golang/exp/tree/master/shootout>.
- [24] Redis labs. <https://redis.io>.
- [25] Redis labs. redis and memcached traffic generation and benchmarking tool. https://github.com/RedisLabs/mentier_benchmark.
- [26] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016.
- [27] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.
- [28] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10):1–19, 1999.
- [29] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 197–212. ACM, 2013.
- [30] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 47(4):105–118, 2012.

- [31] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.
- [32] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 125–136. ACM, 2016.
- [33] SNIA NVM Programming Technical Working Group et al. Nvm programming model (version 1.2), 2017.
- [34] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Sigplan Notices*, volume 38, pages 388–402. ACM, 2003.
- [35] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, 2017.
- [36] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.
- [37] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dullloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.
- [38] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.
- [39] Stefan Marr, Benoit Daloz, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In *ACM SIGPLAN Notices*, volume 52, pages 120–131. ACM, 2016.
- [40] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.
- [41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.
- [42] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.
- [43] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–332. ACM, 2019.
- [44] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M Swift, and Adam Welc. xcalls: safe i/o in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 247–260. ACM, 2009.
- [45] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.
- [46] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 70–83. ACM, 2018.