CHAPTER 11

Designing Data Structures for Persistent Memory

Taking advantage of the unique characteristics of persistent memory, such as byte addressability, persistence, and update in place, allows us to build data structures that are much faster than any data structure requiring serialization or flushing to a disk. However, this comes at a cost. Algorithms must be carefully designed to properly persist data by flushing CPU caches or using non-temporal stores and memory barriers to maintain data consistency. This chapter describes how to design such data structures and algorithms and shows what properties they should have.

Contiguous Data Structures and Fragmentation

Fragmentation is one of the most critical factors to consider when designing a data structure for persistent memory due to the length of heap life. A persistent heap can live for years with different versions of an application. In volatile use cases, the heap is destroyed when the application exits. The life of the heap is usually measured in hours, days, or weeks.

Using file-backed pages for memory allocation makes it difficult to take advantage of the operating system–provided mechanisms for minimizing fragmentation, such as presenting discontinuous physical memory as a contiguous virtual region. It is possible to manually manage virtual memory at a low granularity, producing a page-level defragmentation mechanism for objects in user space. But this mechanism could lead to complete fragmentation of physical memory and an inability to take advantage of huge pages. This can cause an increased number of translation lookaside buffer (TLB) misses, which significantly slows down the entire application. To make effective use of persistent memory, you should design data structures in a way that minimizes fragmentation.

© The Author(s) 2020
S. Scargall, *Programming Persistent Memory*, https://doi.org/10.1007/978-1-4842-4932-1_11

Internal and External Fragmentation

Internal fragmentation refers to space that is overprovisioned inside allocated blocks. An allocator always returns memory in fixed-sized *chunks* or *buckets*. The allocator must determine what size each bucket is and how many different sized buckets it provides. If the size of the memory allocation request does not exactly match a predefined bucket size, the allocator will return a larger memory bucket. For example, if the application requests a memory allocation of 200KiB, but the allocator has bucket sizes of 128KiB and 256KiB, the request is allocated from an available 256KiB bucket. The allocator must usually return a memory chunk with a size divisible by 16 due to its internal alignment requirements.

External fragmentation occurs when free memory is scattered in small blocks. For example, imagine using up the entire memory with 4KiB allocations. If we then free every other allocation, we have half of the memory available; however, we cannot allocate more than 4KiB at once because that is the maximum size of any contiguous free space. Figure 11-1 illustrates this fragmentation, where the red cells represent allocated space and the white cells represent free space.

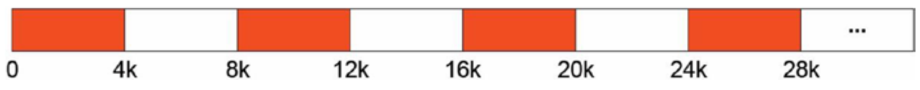

***Figure 11-1.** External fragmentation*

When storing a sequence of elements in persistent memory, several possible data structures can be used:

- Linked list: Each node is allocated from persistent memory.
- Dynamic array (vector): A data structure that pre-allocates memory in bigger chunks. If there is no free space for new elements, it allocates a new array with bigger capacity and moves all elements from the old array to the new one.
- Segment vector: A list of fixed-size arrays. If there is no free space left in any segment, a new one is allocated.

Consider fragmentation for each of those data structures:

- For linked lists, fragmentation efficiency depends on the node size. If it is small enough, then high internal fragmentation can be expected. During node allocation, every allocator will return memory with a certain alignment that will likely be different than the node size.
- Using dynamic array results in fewer memory allocations, but every allocation will have a different size (most implementations double the previous one), which results in a higher external fragmentation.
- Using a segment vector, the size of a segment is fixed, so every allocation has the same size. This practically eliminates external fragmentation because we can allocate a new one for each freed segment.[1]

Atomicity and Consistency

Guaranteeing consistency requires the proper ordering of stores and making sure data is stored persistently. To make an atomic store bigger than 8 bytes, you must use some additional mechanisms. This section describes several mechanisms and discusses their memory and time overheads. For the time overhead, the focus is on analyzing the number of flushes and memory barriers used because they have the biggest impact on performance.

Transactions

One way to guarantee atomicity and consistency is to use transactions (described in detail in Chapter 7). Here we focus on how to design a data structure to use transactions efficiently. An example data structure that uses transactions is described in the "Sorted Array with Versioning" section later in this chapter.

Transactions are the simplest solution for guaranteeing consistency. While using transactions can easily make most operations atomic, two items must be kept in mind. First, transactions that use logging always introduce memory and time overheads. Second, in the case of undo logging, the memory overhead is proportional to the size of data you modify, while the time overhead depends on the number of snapshots. Each snapshot must be persisted prior to the modification of snapshotted data.

[1]Using the `libpmemobj` allocator, it is also possible to easily lower internal fragmentation by using allocation classes (see Chapter 7).

It is recommended to use a data-oriented approach when designing a data structure for persistent memory. The idea is to store data in such a way that its processing by the CPU is cache friendly. Imagine having to store a sequence of 1000 records that consist of 2 integer values. This has two approaches: Either use two arrays of integers as shown in Listing 11-1, or use one array of pairs as shown in Listing 11-2. The first approach is SoA (Structure of Arrays), and the second is AoS (Array of Structures).

Listing 11-1. SoA layout approach to store data

```
struct soa {
    int a[1000];
    int b[1000];
};
```

Listing 11-2. AoS layout approach to store data

```
std::pair<int, int> aos_records[1000];
```

Depending on the access pattern to the data, you may prefer one solution over the other. If the program frequently updates both fields of an element, then the AoS solution is better. However, if the program only updates the first variable of all elements, then the SoA solution works best.

For applications that use volatile memory, the main concerns are usually cache misses and optimizations for single instruction, multiple data (SIMD) processing. SIMD is a class of parallel computers in Flynn's taxonomy,[2] which describes computers with multiple processing elements that simultaneously perform the same operation on multiple data points. Such machines exploit data-level parallelism, but not concurrency: There are simultaneous (parallel) computations but only a single process (instruction) at a given moment.

While those are still valid concerns for persistent memory, developers must consider snapshotting performance when transactions are used. Snapshotting one contiguous memory region is always better then snapshotting several smaller regions, mainly due to the smaller overhead incurred by using less metadata. Efficient data structure layout that takes these considerations into account is imperative for avoiding future problems when migrating data from DRAM-based implementations to persistent memory.

[2]For a full definition of SIMD, see `https://en.wikipedia.org/wiki/SIMD`.

Listing 11-3 presents both approaches; in this example, we want to increase the first integer by one.

Listing 11-3. Layout and snapshotting performance

```
struct soa {
  int a[1000];
  int b[1000];
};

struct root {
  soa soa_records;
  std::pair<int, int aos_records[1000];
};

int main()
{
  try {
    auto pop = pmem::obj::pool<root>::create("/daxfs/pmpool",
             "data_oriented", PMEMOBJ_MIN_POOL, 0666);

  auto root = pop.root();

  pmem::obj::transaction::run(pop, [&]{
    pmem::obj::transaction::snapshot(&root->soa_records);
    for (int i = 0; i < 1000; i++) {
      root->soa_records.a[i]++;
    }

    for (int i = 0; i < 1000; i++) {
      pmem::obj::transaction::snapshot(
                      &root->aos_records[i].first);
      root->aos_records[i].first++;
    }
  });

68   pop.close();
69   } catch (std::exception &e) {
70      std::cerr << e.what() << std::endl;
71   }
72 }
```

- Lines 37-45: We define two different data structures to store records of integers. The first one is SoA – where we store integers in two separate arrays. Line 44 shows a single array of pairs – AoS.
- Lines 56-59: We take advantage of the SoA layout by snapshotting the entire array at once. Then we can safely modify each element.
- Lines 61-65: When using AoS, we are forced to snapshot data in every iteration – elements we want to modify are not contiguous in memory.

Examples of data structures that use transactions are shown in the "Hash Table with Transactions" and "Hash Table with Transactions and Selective Persistence" sections, later in this chapter.

Copy-on-Write and Versioning

Another way to maintain consistency is the copy-on-write (CoW) technique. In this approach, every modification creates a new version at a new location whenever you want to modify some part of a persistent data structure. For example, a node in a linked list can use the CoW approach as described in the following:

1. Create a copy of the element in the list. If a copy is dynamically allocated in persistent memory, you should also save the pointer in persistent memory to avoid a memory leak. If you fail to do that and the application crashes after the allocation, then on the application restart, newly allocated memory will be unreachable.
2. Modify the copy and persist the changes.
3. Atomically change the original element with the copy and persist the changes, then free the original node if needed. After this step successfully completes, the element is updated and is in a consistent state. If a crash occurs before this step, the original element is untouched.

Although using this approach compared to transactions can be faster, it is significantly harder to implement because you must manually persist data.

Copy-on-write usually works well in multithreaded systems where mechanisms like reference counting or garbage collection are used to free copies that are no longer used. Although such systems are beyond the scope of this book, Chapter 14 describes concurrency in multithreaded applications.

Versioning is a very similar concept to copy-on-write. The difference is that here you hold more than one version of a data field. Each modification creates a new version of the field and stores information about the current one. The example presented in “Sorted Array with Versioning” later in this chapter shows this technique in an implementation of the insert operation for a sorted array. In the preceding example, only two versions of a variable are kept, the old and current one as a two-element array. The insert operations alternately write data to the first and second element of this array.

Selective Persistence

Persistent memory is faster than disk storage but potentially slower than DRAM. Hybrid data structures, where some parts are stored in DRAM and some parts are in persistent memory, can be implemented to accelerate performance. Caching previously computed values or frequently accessed parts of a data structure in DRAM can improve access latency and improve overall performance.

Data does not always need to be stored in persistent memory. Instead, it can be rebuilt during the restart of an application to provide a performance improvement during runtime given that it accesses data from DRAM and does not require transactions. An example of this approach appears in “Hash Table with Transactions and Selective Persistence.”

Example Data Structures

This section presents several data structure examples that were designed using the previously described methods for guaranteeing consistency. The code is written in C++ and uses `libpmemobj-cpp`. See Chapter 8 for more information about this library.

Hash Table with Transactions

We present an example of a hash table implemented using transactions and containers using `libpmemobj-cpp`.

As a quick primer to some, and a refresher to other readers, a hash table is a data structure that maps keys to values and guarantees O(1) lookup time. It is usually implemented as an array of buckets (a bucket is a data structure that can hold one or more key-value pairs). When inserting a new element to the hash table, a hash function is applied to the element's key. The resulting value is treated as an index of a bucket to which the element is inserted. It is possible that the result of the hash function for different keys will be the same; this is called a *collision*. One method for resolving collisions is to use separate chaining. This approach stores multiple key-value pairs in one bucket; the example in Listing 11-4 uses this method.

For simplicity, the hash table in Listing 11-4 only provides the `const Value& get(const std::string &key)` and `void put(const std::string &key, const Value &value)` methods. It also has a fixed number of buckets. Extending this data structure to support the remove operation and to have a dynamic number of buckets is left as an exercise to you.

Listing 11-4. Implementation of a hash table using transactions

```
#include <functional>
#include <libpmemobj++/p.hpp>
#include <libpmemobj++/persistent_ptr.hpp>
#include <libpmemobj++/pext.hpp>
#include <libpmemobj++/pool.hpp>
#include <libpmemobj++/transaction.hpp>
#include <libpmemobj++/utils.hpp>
#include <stdexcept>
#include <string>

#include "libpmemobj++/array.hpp"
#include "libpmemobj++/string.hpp"
#include "libpmemobj++/vector.hpp"

```

```
/**
 * Value - type of the value stored in hashmap
 * N - number of buckets in hashmap
 */
template <typename Value, std::size_t N>
class simple_kv {
private:
  using key_type = pmem::obj::string;
  using bucket_type = pmem::obj::vector<
      std::pair<key_type, std::size_t>>;
  using bucket_array_type = pmem::obj::array<bucket_type, N>;
  using value_vector = pmem::obj::vector<Value>;

  bucket_array_type buckets;
  value_vector values;

public:
  simple_kv() = default;

  const Value &
  get(const std::string &key) const
 {
  auto index = std::hash<std::string>{}(key) % N;

  for (const auto &e : buckets[index]) {
   if (e.first == key)
     return values[e.second];
 }

 throw std::out_of_range("no entry in simplekv");
}

84    void
85    put(const std::string &key, const Value &val)
86    {
87     auto index = std::hash<std::string>{}(key) % N;
88
89     /* get pool on which this simple_kv resides */
90     auto pop = pmem::obj::pool_by_vptr(this);
91
92     /* search for element with specified key - if found
93      * update its value in a transaction*/
94     for (const auto &e : buckets[index]) {
95       if (e.first == key) {
96         pmem::obj::transaction::run(
97           pop, [&] { values[e.second] = val; });
98
99         return;
100       }
101     }
102
103     /* if there is no element with specified key, insert
104      * new value to the end of values vector and put
105      * reference in proper bucket */
106      pmem::obj::transaction::run(pop, [&] {
107       values.emplace_back(val);
108       buckets[index].emplace_back(key, values.size() - 1);
109        });
110      }
111   };
```

- Lines 58-66: Define the layout of a hash map as a `pmem::obj::array` of buckets, where each bucket is a `pmem::obj::vector` of key and index pairs and `pmem::obj::vector` contains the values. The index in a bucket entry always specifies a position of the actual value stored in a separate vector. For snapshotting optimization, the value is not saved next to a key in a bucket. When obtaining a non-const reference to an element in `pmem::obj::vector`, the element is always snapshotted. To avoid snapshotting unnecessary data, for example, if the key is immutable, we split keys and values into separate vectors. This also helps in the case of updating several values in one transaction. Recall the discussion in the "Copy-on-Write and Versioning" section. The result could turn out to be next to each other in a vector, and there could be fewer bigger regions to snapshot.

- Line 74: Calculate hash in a table using standard library feature.
- Lines 76-79: Search for entry with specified key by iterating over all buckets stored in the table under `index`. Note that `e` is a const reference to the key-value pair. Because of the way `libpmemobj-cpp` containers work, this has a positive impact on performance when compared to non-const reference; obtaining non-const reference requires a snapshot, while a const reference does not.
- Line 90: Get the instance of the `pmemobj` pool object, which is used to manage the persistent memory pool where our data structure resides.
- Lines 94-95: Find the position of a value in the values vector by iterating over all the entries in the designated bucket.
- Lines 96-98: If an element with the specified key is found, update its value using a transaction.
- Lines 106-109: If there is no element with the specified key, insert a value into the values vector, and put a reference to this value in the proper bucket; that is, create key, index pair. Those two operations must be completed in a single atomic transaction because we want them both to either succeed or fail.

Hash Table with Transactions and Selective Persistence

This example shows how to modify a persistent data structure (hash table) by moving some data out of persistent memory. The data structure presented in Listing 11-5 is a modified version of the hash table in Listing 11-4 and contains the implementation of this hash table design. Here we store only the vector of keys and vector of values in persistent memory. On application startup, we build the buckets and store them in volatile memory for faster processing during runtime. The most noticeable performance gain would be in the `get()` method.

Listing 11-5. Implementation of hash table with transactions and selective persistence

```
#include <array>
#include <functional>
#include <libpmemobj++/p.hpp>
#include <libpmemobj++/persistent_ptr.hpp>
#include <libpmemobj++/pext.hpp>
#include <libpmemobj++/pool.hpp>
#include <libpmemobj++/transaction.hpp>
#include <libpmemobj++/utils.hpp>
#include <stdexcept>
#include <string>
#include <vector>

#include "libpmemobj++/array.hpp"
#include "libpmemobj++/string.hpp"
#include "libpmemobj++/vector.hpp"

template <typename Value, std::size_t N>
struct simple_kv_persistent;

/**
 * This class is runtime wrapper for simple_kv_peristent.
 * Value - type of the value stored in hashmap
 * N - number of buckets in hashmap
 */
template <typename Value, std::size_t N>
class simple_kv_runtime {
private:
  using volatile_key_type = std::string;
  using bucket_entry_type = std::pair<volatile_key_type, std::size_t>;
  using bucket_type = std::vector<bucket_entry_type>;
  using bucket_array_type = std::array<bucket_type, N>;

  bucket_array_type buckets;
  simple_kv_persistent<Value, N> *data;

public:
 simple_kv_runtime(simple_kv_persistent<Value, N> *data)
 {
  this->data = data;

  for (std::size_t i = 0; i < data->values.size(); i++) {
   auto volatile_key = std::string(data->keys[i].c_str(),
               data->keys[i].size());

   auto index = std::hash<std::string>{}(volatile_key)%N;
   buckets[index].emplace_back(
    bucket_entry_type{volatile_key, i});
   }
  }

  const Value &
  get(const std::string &key) const
  {
   auto index = std::hash<std::string>{}(key) % N;

   for (const auto &e : buckets[index]) {
    if (e.first == key)
     return data->values[e.second];
   }

   throw std::out_of_range("no entry in simplekv");
  }

  void
  put(const std::string &key, const Value &val)
  {
   auto index = std::hash<std::string>{}(key) % N;

  /* get pool on which persistent data resides */
     auto pop = pmem::obj::pool_by_vptr(data);

   /* search for element with specified key - if found
    * update its value in a transaction */
   for (const auto &e : buckets[index]) {
    if (e.first == key) {
      pmem::obj::transaction::run(pop, [&] {
        data->values[e.second] = val;
      });

     return;
    }
   }

  /* if there is no element with specified key, insert new value
   * to the end of values vector and key to keys vector
   * in a transaction */
   pmem::obj::transaction::run(pop, [&] {
    data->values.emplace_back(val);
    data->keys.emplace_back(key);
   });

   buckets[index].emplace_back(key, data->values.size() - 1);
 }
};

/**
 * Class which is stored on persistent memory.
 * Value - type of the value stored in hashmap
 * N - number of buckets in hashmap
 */
template <typename Value, std::size_t N>
struct simple_kv_persistent {
 using key_type = pmem::obj::string;
```

```
143  using value_vector = pmem::obj::vector<Value>;
144  using key_vector = pmem::obj::vector<key_type>;
145
146 /* values and keys are stored in separate vectors to optimize
147  * snapshotting. If they were stored as a pair in single vector
148  * entire pair would have to be snapshotted in case of value update */
149  value_vector values;
150  key_vector keys;
151
152  simple_kv_runtime<Value, N>
153  get_runtime()
154  {
155   return simple_kv_runtime<Value, N>(this);
156  }
157 };
```

- Line 67: We define the data types residing in volatile memory. These are very similar to the types used in the persistent version in "Hash Table with Transactions." The only difference is that here we use `std` containers instead of `pmem::obj`.
- Line 72: We declare the volatile buckets array.
- Line 73: We declare the pointer to persistent data (`simple_kv_persistent` structure).
- Lines 75-88: In the `simple_kv_runtime` constructor, we rebuild the bucket's array by iterating over keys and values in persistent memory. In volatile memory, we store both the keys, which are a copy of the persistent data and the index for the values vector in persistent memory.
- Lines 90-101: The `get()` function looks for an element reference in the volatile buckets array. There is only one reference to persistent memory when we read the actual value on line 97.
- Lines 113-121: Similar to the `get()` function, we search for an element using the volatile data structure and, when found, update the value in a transaction.

- Lines 126-129: When there is no element with the specified key in the hash table, we insert both a value and a key to their respective vectors in persistent memory in a transaction.
- Line 131: After inserting data to persistent memory, we update the state of the volatile data structure. Note that this operation does not have to be atomic. If a program crashes, the bucket array will be rebuilt on startup.
- Lines 149-150: We define the layout of the persistent data. Key and values are stored in separate `pmem::obj::vector`.
- Lines 153-156: We define a function that returns the runtime object of this hash table.

Sorted Array with Versioning

This section presents an overview of an algorithm for inserting elements into a sorted array and preserving the order of elements. This algorithm guarantees data consistency using the versioning technique.

First, we describe the layout of our sorted array. Figure 11-2 and Listing 11-6 show that there are two arrays of elements and two size fields. Additionally, one `current` field stores information about which array and size variable is currently used.

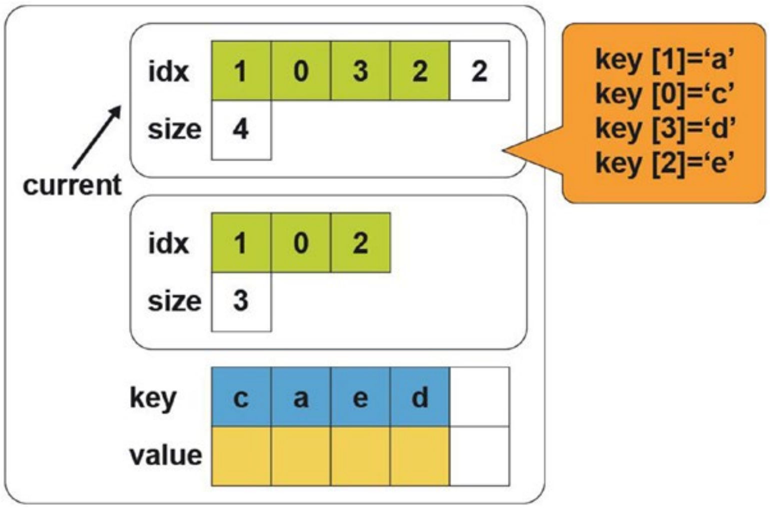

Figure 11-2. *Sorted array layout*

Listing 11-6. Sorted array layout

```
41  template <typename Value, uint64_t slots>
42  struct entries_t {
43    Value entries[slots];
44    size_t size;
45  };
46
47  template <typename Value, uint64_t slots>
48  class array {
49  public:
50   void insert(pmem::obj::pool_base &pop, const Value &);
51   void insert_element(pmem::obj::pool_base &pop, const Value&);
52
53   entries_t<Value, slots> v[2];
54   uint32_t current;
55  };
```

- Lines 41-45: We define the helper structure, which consists of an array of indexes and a size.
- Line 53: We define two elements array of `entries_t` structures. `entries_t` holds an array of elements (entries array) and the number of elements in the node as the `size` variable.
- Line 54: This variable determines which `entries_t` structure from line 53 is used. It can be only 0 or 1. Figure 11-2 shows the situation where the current is equal to 0 and points to the first element of the v array.

To understand why we need two versions of the `entries_t` structure and a current field, Figure 11-3 shows how the insert operation works, and the corresponding pseudocode appears in Listing 11-7.

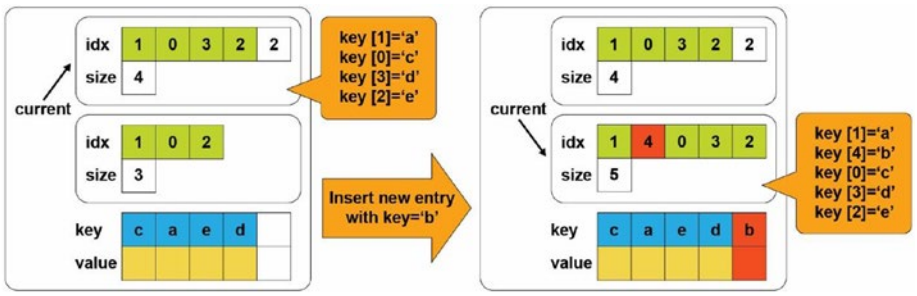

Figure 11-3. *Overview of a sorted tree insert operation*

Listing 11-7. Pseudocode of a sorted tree insert operation

```
template <typename Value, uint64_t slots>
void array<Value, slots>::insert_element(pmem::obj::pool_base &pop,
                   const Value &entry) {
  auto &working_copy = v[1 - current];
  auto &consistent_copy = v[current];

  auto consistent_insert_position = std::lower_bound(
   std::begin(consistent_copy.entries),
   std::begin(consistent_copy.entries) +
              consistent_copy.size, entry);
   auto working_insert_position =
       std::begin(working_copy.entries) +
       std::distance(std::begin(consistent_copy.entries),
       consistent_insert_position);

        std::copy(std::begin(consistent_copy.entries),
                  consistent_insert_position,
                  std::begin(working_copy.entries));

        *working_insert_position = entry;

        std::copy(consistent_insert_position,
                  std::begin(consistent_copy.entries) +
                     consistent_copy.size,
                  working_insert_position + 1);
80
81          working_copy.size = consistent_copy.size + 1;
82  }
83
84  template <typename V, uint64_t s>
85  void array<V,s>::insert(pmem::obj::pool_base &pop,
86                                   const Value &entry){
87   insert_element(pop, entry);
88   pop.persist(&(v[1 - current]), sizeof(entries_t<Value, slots>));
89
90   current = 1 - current;
91   pop.persist(&current, sizeof(current));
92 }
```

- Lines 60-61: We define references to the current version of entries array and to the working version.
- Line 63: We find the position in the current array where an entry should be inserted.
- Line 67: We create iterator to the `working` array.
- Line 71: We copy part of the current array to the working array (range from beginning of the current array to the place where a new element should be inserted).
- Line 75: We insert an entry to the working array.
- Line 77: We copy remaining elements from the current array to the working array after the element we just inserted.
- Line 81: We update the size of the working array to the size of the current array plus one, for the element inserted.
- Lines 87-88: We insert an element and persist the entire v[1-current] element.
- Lines 90-91: We update the current value and save it.

Let's analyze whether this approach guarantees data consistency. In the first step, we copy elements from the original array to a currently unused one, insert the new element, and persist it to make sure data goes to the persistence domain. The persist call also ensures that the next operation (updating the current value) is not reordered before any of the previous stores. Because of this, any interruption before or after issuing the instruction to update the current field would not corrupt data because the current variable always points to a valid version.

The memory overhead of using versioning for the insert operation is equal to a size of the entries array and the current field. In terms of time overhead, we issued only two persist operations.

Summary

This chapter shows how to design data structures for persistent memory, considering its characteristics and capabilities. We discuss fragmentation and why it is problematic in the case of persistent memory. We also present a few different methods of guaranteeing data consistency; using transactions is the simplest and least error-prone method. Other approaches, such as copy-on-write or versioning, can perform better, but they are significantly more difficult to implement correctly.