BOPL: Batch Optimized Persistent List

Paulo Martins  
paulo.r.martins@tecnico.ulisboa.pt  
Instituto Superior Técnico, Lisbon, Portugal  
May 2019

Abstract

Due to the increase of the number of operations in the databases, and with the improvement of both performance and capacity of the DRAM, the use of In Memory Database (IMDB) has become feasible instead of the current Disk Based Database (DBDB). The emergence of more efficient memories, such as Optane SSD and NVRAM, may lead to replacing the disks in the IMDB and making them more efficient. The use of Non-Volatile Random Access Memory (NVRAM) imposes a modification on the structures that are used in DBDB, because in case of a system failure, the data in NVRAM may be incoherent. A number of new structures have been developed with the aim of making the data consistent in NVRAM, but it was found that these structures tend to perform poorly, due to the fact that each modification is required to persist. Batch-Optimized Persistent List (BOPL) aims to achieve better performance compared to existing solutions, leaving the structures always consistent in both architectures that have persistent memory such as Intel Optane and/or NVRAM. This type of performance is obtained in BOPL, because persistence is not guaranteed during the critical path of operations, but in the background.

Keywords: Non-Volatile Random Access Memory (NVRAM), Intel Optane SSD, Disk Based Database (DBDB), In Memory Database (IMDB), key-values structures

1. Introduction

Many companies have for several databases. These have to be able to process numerous operations, daily, in real time [13]. DBDB, although widely used, have performance/scalability limitations due to the bottleneck that the disk usage implies [13]. This is due to the fact that, in this type of database, all information is on disk, so when it is not in cache, it is necessary to access it, which generates a large latency [13]. With the development of the Dynamic Random Access Memory (DRAM) memories, both, in terms of capacity and speed, became feasible to create and use IMDB. The IMDB, compared to DBDB, has better performance because access and manipulation of information is much faster in DRAM than in secondary memories, like Solid State Disk (SSD) and Hard Drive Disk (HDD) [13] [22].

However, IMDB has a major disadvantage which is related to the fact that the DRAM is volatile and the IMDB requires secondary memories for storing information. This because that in case of a system failure, the data stored in the secondary memories can be recovered [20].

Recently, the development of new memory related technologies makes it expected that new types of memory will emerge, one that presents more significant gains in comparison to the HDD, SSD, and Optane DC SSD [10] and the other can integrate a persistent alternative to DRAM in modern architectures, in particular NVRAM [13].

1.1. Problem

The use of NVRAM has a problem: the transmission of information between the CPU and the NVRAM is not linear [11], which means, if the developer does not force the flush from the cache line of the CPU, it can reorder the instructions. If there is a system failure, this reordering may lead to the inconsistency of the information stored in NVRAM. In the figure 1 it’s illustrated the insertion of the value 3 into an ordered structure. The process consists in moving a slot the values 4 and 5, inserting the value 3 and increasing the number of elements. During insertion of value 3, if there is a system failure, three types of inconsistencies may occur:

- Inconsistency 1: Value 4 and 5 advance a slot and there is a system failure;
- Inconsistency 2: Value 4 and 5 advance a slot, value 3 is inserted, but due to a system failure, the increment in number of elements is not performed;
- Inconsistency 3: The operation is all performed in cache, but after writing (flush) the first line of cache the system fails.
1.2. Hypothesis
In order to solve the problem described above, multiple solutions use flush in each operation, which ensures that the information is persisted in memory. However, constant use of it, make the solutions worst in terms of performance, both in the perspective of DBDB, and in IMDB. The BOPL was designed to decrease the additional latency created by the use of the flush, reducing the number of flushes without compromising the consistency of the persisted information in memory (both in NVRAM, and in Optane). To do this, BOPL will use the concept of memory zones (pages) to persist and store the information. The information will be placed in pages automatically by the CPU and, when a page is completely filled, it will be flushed, thus ensuring that all is persisted. This mechanism will guaranty better performance than current solutions and will ensure consistency of the data in memory, even in presence of failures. In this initial proposal, BOPL uses, as indexing structure, a linked list, however it is believed that with small adaptations, in the future can be endowed with more complex structures, such as trees, thus having a significant impact on the performance of databases.

1.3. Contributions
The contributions of this work are:
- **Review of related work on technologies in the context of consistency techniques:** This study reports different types of technological approaches associated with consistency techniques. It also describes the advantages and disadvantages of each approach, regarding the use of different persistence models, and programming models.
- **Design and implementation of the algorithm:** Implementation of the algorithm, that consist on, the implementation of a normal and generic case and, the BOPL per say. The generic algorithm it’s used only offer testing purpose.
- **Evaluations and results of the implemented systems:** It was made a set of evaluations for the performance of the algorithms and the use of this type of solutions in key-value structures. These assessments allows to identify advantages and disadvantages of this approach. These results have proven which solutions are most relevant in terms of performance.

1.4. Document structure
The rest of the report is organized as follows. In section 2 it is presented the background related to databases, indexing structures and types of memory. Section 3 describes the state of the art regarding different types of technological approaches in consistency techniques, persistancy models and programming models for NVRAM. Section 4 reports the implementation of the BOPL. In the Section 5 it’s detailed the experimental evaluations more specifically, the metrics, the hardware and the BOPL results. Finally, in the Section 6 it’s discussed the results, and the last section concludes the report.

2. Background
Before investigating the related work, it is necessary to understand some basic concepts about databases and indexation structures, more specifically, the B+ Tree and the Radix Tree.

2.1. Disk Based Database (DBDB) & In Memory Database (IMDB)
There are two types of databases, the DBDB and IMDB. The DBDB differ from the IMDB because, when a query is processed, if the result isn’t in the memory pool, the database has to get it from the disk, that in some cases creates a bottleneck between the memory pool and the disks. With the increase in terms of performance and capacity of DRAM, it became reliable the use of IMDB. These are better in performance compared with DBDB, but they have a huge problem, when they start all the information on disk, they must go to the memory pool and, if the system has a power failure, all the information in DRAM it’s lost.

2.2. Indexing Structures
**B+ tree:** Is a balanced tree ie, separating the tree in the middle by the root, in the left part would be the nodes, that have the key smaller than the one of the root and, at the right, we have the keys with bigger keys to the one of the root.
This tree has a vector with the capacity to contain N keys, N being the size of the vector. A disadvantage of the B+ Tree it's that needs to be re-balanced periodically and, with the increase of information, it tends to consume a substantial amount of memory [15].

Radix Tree: This tree allows the operations to be done in $O(k)$, where k is the dimension of the keys, instead of $O(\log(n))$. That is, the performance of the operations is not influenced by the size of the tree, but by the size of the key [3]. An adaptation of the Radix tree allows to improve the efficiency in terms of memory usage. This adaptation is titled Adaptive Radix Tree (ART). The difference between ART and the Radix tree, is that ART contains four types of nodes. Each component is a vector with predefined size. The NODE4, NODE16, NODE48, and NODE256 are able to contain 4, 16, 48 and 256 pointers, respectively.

2.3. Types of Memory
Two types of non-volatile memory, HDD and SSD are currently used in most systems, but more recently there has been another non-volatile memory, Intel Optane [1] and, it has a better performance than HDD and SSD [2] [18]. Also, in recent studies, appeared a new type of memory, called NVRAM. This memory such as DRAM, can be addressed at the byte level directly through machine instructions store and load. According to the researchers, NVRAM will ensure that 8 bytes are atomically modified [8] [9][16].

3. Related Work
In this section, it will be described an overview of the related work, where it's shown the different approaches of consistency techniques.

3.1. Persistence Models
One possible way to combat the consistency problem, is to provide the CPU with the ability to order the execution of the instructions, thus ensuring that they occur in the intended order. For this, there are three main persistence models proposed that define how the CPU should act, and in what order the information is persisted [17]:

- **Strict Persistency**: This model ensures that the instruction flow is the same, both in DRAM and NVRAM. The information is persisted in memory in the order of execution of the instructions. Using this model, while offering complete assurance in ordering, it has a major negative impact on the performance of the program [17];

- **Epoch Persistency**: This does not guarantee that the instruction flow is the same between DRAM and persistent memory, but allows the use of regions, called epochs. It ensures the ordering of the instructions between two different epochs, ie the instructions associated with an epoch are executed only after the previous epoch its entirety executed. Epoch Persistency performs better than Strict Persistency, since it provides ordering between different epochs, thus not compromising the remaining memory [8] [17] [19].

- **Strand Persistency**: This model is similar to Epoch Persistency. While the previous has possibility of creating an epoch, it offers the opportunity to create strands. Strand is an execution interval of a thread, which establish the order of the execution of all instructions that are within strand. In order to be able to sort this model, it is necessary to analyze the persistent memory after each write, to identify the position of the next strand [17].

The CPU currently offers only a totally relaxed model [17], ie the CPU has no ability to sort instructions. If it is necessary, it is up to the developer to flush the information that he wants to persist.

3.2. Consistency Techniques

**In-place Update**: This technique uses the fact that NVRAM atomically writes 8 bytes of information, in order to modify or insert small information (less than or equal to 8 bytes). As this assures atomicity, it does not allow for inconsistencies in NVRAM where, either it is all written/modified or no type of modification happens.

**In-place Append**: This technique refers to put a new structure in NVRAM attached to an existing one. For example, while putting an Element B next to an Element A, first the developer must persist B and, only when B it's safely persisted, he can use the In-place Update technique to modify the next of A.

**Shadow Copy**: This technique is useful if it is necessary to modify more than 8 bytes. This consists of making a copy of the node that needs to be modified, changing the information of the copied node, persisting the modified node, and finally changing the pointer of the previous node to reference the modified.

**Redo Logging**: This is applied in critical operations such as insertions and deletions of elements in a structure. This method consists in putting information about the operations that will be performed in a persistent register (log). When a system failure occurs the algorithm can beware, of what operations were in progress before the failure and reproduce them. Although effective, the use of logs reduces the performance of programs that use them [5], since they need to flush the information at two different locations in the log and in the structure.
3.3. Programming Models

File System for NVRAM

File systems are used by many systems today, but normally it’s applied to non-volatile memories based on disks, Hard Drive Disk (HDD) or Solid State Disk (SSD). With the origin of NVRAM, researchers have implemented Byte-Addressable Persistent File System (BPFS) [8].

BPFS uses a tree to store information about files, which is made up of 4KB fixed-size nodes. The BPFS consists of three types of nodes:

- Inode: contains the metadata of the file, it is only considered valid if it matches a directory that is also valid.
- Directory: contains a vector of pointers, for the corresponding file, which is only considered valid if the file exists.
- File: This data structure contains the information about the file itself.

This system is based on a hybrid architecture, namely DRAM memory and NVRAM. To keep NVRAM consistent, BPFS uses the following techniques: In-place Update, In-place Append and Partial Shadow-Copy.

Persistent Heaps

Several software such as Mnemosyne [21], Recovery Write-Ahead System for In-Memory on Volatile Data-Structures (REWIND) [4] and Non-Volatile Heap (NV-Heap) [7], were created in such a way as to facilitate the programming of systems in NVRAM. All of the above systems have an interface, which makes it easy for developers to use NVRAM. These have the ability to map zones in NVRAM (heaps), giving developers functions to maintain persistent data structures, with the support of atomic transactions.

REWIND [4] gives programmers the ability to, implement an atomic persistent section. For that to be possible, REWIND uses a log composed by a set of linked lists.

The NV-Heap [7] gives the developers the ability to use persistence while programming with objects, thus writing them in NVRAM. To store the objects on the heap, a tree is used as shown in Figure 3. All objects can be reached from the root node and, in order to allocate and deallocate space, a reference count is made for each object and, if it reaches zero, the object is removed from the tree [7].

To ensure the consistency of the structures, NV-Heap also provides several types of pointers. There are four types of pointers, from Volatile to Volatile (V - V), Non - Volatile to Non - Volatile (NV - NV), V - NV and NV - V. When an object is created, NV - V pointers are not allowed. NV-Heap also uses a log, so that if a crash occurs and the system reboots, it can make rollback of the information.

Mnemosyne [21] provides persistent variables, sections, heaps, and transitions functions. To provide all of the above functionality, and to guaranty that the information is stored in a consistent manner, Mnemosyne uses all of the consistency techniques.

Non Volatile Memories-aware structures

There are data structures that are implemented in order to maximize the operation on NVRAM. These structures tend to be key-value indexing, like trees, such as Fingerprinting Persistent Tree (FP-Tree) [16], Persistent B+ tree [6], Write Optimal Radix Tree (WORT) [14], Non-Volatile Tree (NV-Tree) [23] and wB+-Tree [5].

These trees use different types of strategies, both in the process of searching for values, inserting, removing and modifying node values. They also vary in the architecture they adopt, that is, some structures only use memory NVRAM, while others operate on a hybrid architecture.

Trees that are based in memory-only architectures, based on NVRAM, for example wB+-Tree [5], don’t need to have functions with the purpose of tree reconstruction, since it is already in NVRAM. In hybrid architectures, such as FP-Tree [16], the leaf nodes are stored in NVRAM and the internal nodes are in DRAM so, when a fault occurs, all internal nodes have to be rebuilt from the leaf nodes.

The main functions associated with the trees are: insertion; search; removal; modification of values; separation and recovery.

Insertion: Usually it is done using free spaces, if there is free space in the reference vector the node it’s stored. If there is no space, it will be necessary to separate the references or to allocate a new vector. In order to ensure that the insertion is done correctly, In-place Append technique it’s used together with the logs.

Search: This function is one of the most important, because it is one of most used operations in database, which consists of the process of finding a value associated with given key. Because leaf nodes tend not to be ordered, which means that there is no overwriting in memory [6], it is necessary to sort them on the internal nodes, in order
to make the search more efficient. For this, some trees use insertion bitmaps, because they are ordered, and it is simpler to organize the internal nodes. [16].

Delete: This function can be done using in-place update, i.e., it can be performed by placing a bitmap entry with zero [16]. It is also possible to flag the node in question to brand it for removal [23].

Update: This operation consists of changing values of a node, which can be done by using In-place Update for small information. If the information is greater than 8 bytes, it is necessary to apply Shadow Copy.

Split: This function allows the tree to be scalable. When the reference vector gets full, it will be essential to separate the nodes and allocate another vector of any size, in order to store/reference the remaining nodes. This split is different depending on the used base structure. If the base structure is a B+-Tree, a vector of equal size is allocated, if it is a Radix Tree, no separation is required.

In the case of ART, what happens is that occurs a reallocation of the reference vector i.e., the size of the vector increases for example, a NODE4 when fully populated becomes an NODE16.

Recovery: Recovery is done when a system crashes and restarts. Recovery can be categorized in two ways: reconstruction of volatile nodes, or consistent recovery of non-persistent operations at the time of failure.

The rebuilding of the tree, as the name implies, is the process of reassembling the tree into DRAM, but this operation is very costly in terms of performance [16].

The continuity of operations is achieved because of the existence of diaries, which contain information of the operations before the failure. After the system is restarted, the records are analyzed and, if there is any operation that has not been completed, it is redone with the node in question.

3.4. Discussion

If the programmer chooses other advantages in exchange of poorer performance, he may use a persistent file system, or even heaps. The File System model, out of the three models, performs poorly because it is necessary to make system calls, in order to interact with the files. These calls cause a high latency, due to the need of dependency of the operating system. The Persistent Heaps model, compared to the File System, is better, since it does not need to make system calls to put information into NVRAM. The NVM-aware structures are developed with focus on a specific structure, where the number of flushes that are performed in the operations are minimized to the maximum. This characteristics causes NVM-aware structures to have greater performance potential than Persistent Heaps.

Both File System and Persistent Heaps provide consistency for any type of information, whereas NVM-aware structures only guarantee consistency of information regarding the structure. With this information, it is better to choose an NVM-aware approach when building a list for NVRAM.

Structures in hybrid architectures need to rebuild the tree, influencing negatively the performance of the program. The different dimensions of keys are an important feature in indexing frameworks since, a structure can be used to index multiple keys, regardless the size. Considering these two characteristics, we can assume that wB+-Tree [5] and WORT [14] are more efficient than the others.

However, by looking to the investigations [5] [6] [14] [16] [23], it can be assumed that WORT performs better than the rest. This is due to the fact that it has a Radix tree as a base structure, this way it becomes possible to write the keys, using only the In-place Update and In-place Append.

It can also be verified that structures that use records to ensure the consistency of the information tend to perform, on average, more flushes instructions than the others. It can also be inferred that structures which use the In-place Update technique tend to perform better.

4. Implementation

In the following chapter it will be described the mechanisms, functions, as well as the structures used in BOPL. For standardization purposes, during the description of the algorithms and structures, data types that have the prefix perm., will be considered permanent. Those that have the prefix vol., will be considered volatile, that is, stored in DRAM. It is called “secure writing,” the one that is, assuredly, persistent and consistent. On the other hand, “unsecured writing” offers no guarantee of persistence and/or consistency. Finally, it is called mapped zone to the region of virtual memory that mirrors the memory of persistence.

4.1. First Implementation

In the first instance, a naive solution was created, by not having any type of optimization, where the goal was to implement all the functionality. This implementation is not focused on the optimization but rather on the ensuring the consistency of information.

BOPL is a library that provides the programmer with a key-value data structure. Those elements are represented in the data structure [1]. In order to provide a good performance, it’s crucial to have some global variables. These are intended to provide information on the details of the list.

The variables that will be used are the following:
Data Structure 1: Description of data structure elements

```c
typedef struct perm_Elem {
    long key;
    size_t value_size;
    char* value;
    Elem* next;
} Elem;
```

- `perm_Elem* buffer`: is a pointer that points to the start of the mapped area;
- `perm_Elem* workingPointer`: is responsible for pointing to the first free area of NVRAM, which corresponds to placement of a new element;
- `perm_Elem* headerPointer`: is a pointer that points to the head of the list.
- `perm_Elem* tailPointer`: It is a pointer that points to the last element of the list.

Using these variables, it was possible to implement the functions that will be described in the following paragraphs.

**Function bopl_init**

The algorithm is intended to initialize the data structure that the system uses. Pointers that point to the data structure are initialized with information in memory (rows 2 to 5). If there is no information, the condition in line 6 is confirmed, which then means that the structure was not initialized. In line 7, structure will be initialized and the `workingPointer`, `headerPointer` and `tailPointer` will have the value of `buffer` (line 7 to 10).

```
Function bopl_init (int sizeOfBuffer) {
    Elem* buffer = getBuffer();
    Elem* workingPointer = getWorkingPointer();
    Elem* tailPointer = getTailPointer();
    Elem* headerPointer = getHeaderPointer();
    if (buffer == NULL) then
        buffer = createBuffer(sizeOfBuffer);
        workingPointer = buffer;
        headerPointer = buffer;
        tailPointer = buffer;
    end
```

**Function bopl_insert**

In the algorithm 3, the implementation of the `bopl_insert` function is illustrated. In line 2, a new element is created using the arguments of the function, from line 3 to 6 the created element is flushed, that is, it is persisted. Finally, in line 7 the new element is associated with the next of the `tailPointer`, in line 8, the next of `tailPointer` is persisted and, on lines 9 and 10, the `tailPointer` becomes the new element and it’s flushed.

```
Function bopl_insert (vol_long key, vol_size_t sizeOfValue, vol_void* value) {
    vol_Elem* newElement = generateElement(key, sizeOfValue, value, workingPointer);
    while newElement < workingPointer do
        flush(newElement);
        newElement += WORD_BYTE;
    end
    tailPointer -> next = newElement;
    flush(tailPointer -> next);
    tailPointer = newElement;
    flush(tailPointer);
```

**Algorithm 3: Description of function bopl_insert**

**Function bopl_inplace_insert**

This function is similar to the previous one. The only difference is that this one has a parameter that identifies the key of the element that will stay on the left. With that being said, this implementation must search for the left element, persist the new element with its next updated and persists the next of the left element (Algorithm 4).

**Bopl_update**

This function uses the technique of Shadow Copy, which creates a new element with the new values. The algorithm searches the left of the element in question where, if the key of the element exists, then the new element is persisted with all its values modified. After this, the next of the left element is securely written using the In-place-Update technique.

**Bopl_remove**

This method only uses the In-place-Update technique. This function searches the left element (leftElement), the one that is next to the one to be removed, if the element exists on the list then, the leftElement->next becomes leftElement->next->next.

**Bopl_search**

Given a key, it loops on all elements of the list and, if it encounters the element with the correct key, returns the correspondent value.

**Bopl_close**

The main purpose of this function it’s to deallocate the memory in NVRAM.

**Restriction**

In the course of the description of the functions,
Algorithm 4: Description of bopl_inplace_insert

Function bopl_inplace_insert (vol_long leftKey, vol_long key, vol_size_t sizeOfValue, vol_void* value)
  vol_Element* newElement = generateElement(key, sizeOfValue, value, workingPointer);
  if leftKey == 0 then
    newElement->next = headerPointer;
    while newElement < workingPointer do
      flush(newElement);
      newElement += WORD_BYTE;
    end
    headerPointer = newElement;
    flush(headerPointer);
  end
  else
    vol_Element* leftElement = findElement(leftKey);
    newElement->next = leftElement->next;
    while newElement < workingPointer do
      flush(newElement);
      newElement += WORD_BYTE;
    end
    leftElement->next = newElement;
    flush(leftElement->next);
    if newElement->next == NULL then
      tailPointer = newElement;
      FLUSH(tailPointer);
    end
  end

it can be verified that all algorithms that aim at the manipulation of the list, there is at least one forced flush. The forcing of flushes can incur on a performance decrease.

4.2. Optimization Technique
To address the problem encountered above, an optimization mechanism has been implemented that uses pagination, in order to hide flushes, thus removing them from the critical path. In order to implement this optimization technique, it was necessary to add the following variables:

- **perm_Element* safePointer**: flags the end of the last page that was written in a secure way;
- **vol_semaphore flushPageSemaphore**: Activates or block the thread that write the information securely;
- **vol_bool workDone**: It’s true if all instructions were executed, otherwise it’s false.

To implement this optimization technique a thread it’s created, and it’s that thread that guarantees the consistency. Some were created and some modified.

The bopl_init was modified in order to initialize the semaphore and the thread. The bopl_close function was changed in order to activate the variable workDone, and thus wait for the thread.

When the algorithm thread is executed, this forces the flush at page level to execute the flushRange function (algorithm 5). If the page(s) have already been filled with elements (line 5), or if the instructions have already finished (executed bopl_close) and there are unsafe elements (line 9) the pages(s) is/are persisted.

In the flushRange function, (Algorithm 5) more specifically on line 3, all modified bytes (function argument) are persisted (line 4). At line 9, the safePointer is updated and flushed to ensure that persistence occurred safely.

Algorithm 5: Pseudo-code for flushRange

Function flushRange (vol_long bytesToFlush)
  vol_Element* toFlush = safePointer;
  while bytesToFlush > 0 do
    flush(toFlush);
    toFlush += WORD_BYTE;
  end
  flush(tailPointer);
  safePointer = toFlush;
  flush(safePointer);

Algorithm 6: Description forcePageFlush

Function forcePageFlush ()
  while !sem_wait(workingSemaphore) do
    vol_int modifiedBytes = workingPointer - safePointer;
    if modifiedBytes >= PAGE_SIZE then
      flushRange(PAGE_SIZE);
    else
      if workDone workingPointer >= safePointer then
        flushRange(modifiedBytes);
      end
    end
  end

Optimized Insertion
The optimization technique allows element creation to be performed without any flush in the critical path. Before the creation of the new element, the function checkThreshold (algorithm 7) is executed.
Algorithm 7: Description of the function checkThreshold

Function checkThreshold (vol_size_t sizeOfNewElement)

// getPointerPage it’s a function that returns the page of a pointer
int lastPage =
  getPointerPage(workingPointer);
int nextPage =
  getPointerPage(workingPointer + sizeOfNewElement);
if lastPage < nextPage then
  // getLeftToFillPage returns what’s left to fill the page
  workingPointer +=
    getLeftToFillPage(workingPointer);
  sem_post(&workingSemaphore);
end

Restriction

This optimization technique is crucial for a safe creation of elements, however, although it works correctly for insertion, the same does not apply to the remaining functions of BOPL. This works correctly, because of the continuous allocation of pages, which means that operations that modify the elements already created, are not possible to be persisted in a secure way.

4.3. Second and Third Implementation

In order to overcome the restriction mentioned above, two solutions were implemented. The first solution is to have as an additional structure a journal (structure 8) that, in case of a system failure, BOPL can return to a state of consistency. The second solution consists of having a structure, a hashmap (structure 9), in volatile memory, which stores all the modifications.

Data Structure 9: Representation of Elements in hashmap

typedef vol_struct vol_modification

{ // epoch_k stores the page of when the modification occurred
  vol_int epoch_k;
  // leftElement pointer to left element of the one modified
  perm_Element* leftElement;
  // new Next Element pointer to the new next of leftElement
  vol_Element* new Next Element;
  vol_modification* nextModification;
}

On both solutions the modifications are associated with an epoch, which represents the working page. So, when a given page is persisted, all modifications associated will also be persisted.

Restrictions

Although the first implementation uses the optimization technique, and it’s able to implement all functions of BOPL, because it applies the Undo-Log technique, in certain operations executes at least one flush on the critical path. The second implementation doesn’t execute flush on the critical path but, because of all the modifications are stored in the hashmap, operations that perform searches need to, in every iteration, check on the hashmap if the element is updated.

4.4. System Failures and Recovery

The first implementation does not require any recovery mechanism because, the structure’s consistency is ensured by making the flush on a correct order. In the case of the second and third implementations, it is already necessary to have one or more recovery mechanisms.

In order to guaranty consistency in both solutions, a bitmap was created with information from pages that are possibly damaged (dirtyPages).

As can be seen in the algorithm 10 if the value of the safePointer is less than the value of the workingPointer (line 5) it means that a system failure has occurred, and the corrupted pages may exist, therefore it is necessary to flag them on dirtyPages (lines 8 to 12). Finally, for each bit in the dirtyPages, the value is checked and if returns 1, the associated page is blocked using the mprotect function (lines 14 to 18). Finally, in line 22 to 25 the tailPointer is updated. If an access to a protected page occurs during the loop of line 23, a function (signal_handler) that runs in parallel de-

http://man7.org/linux/man-pages/man2/mprotect.2.html
Algorithm 10: pseudo-code for the recover mechanism function

Function recoverMechanism ()

1. pem_bitmap dirtyPages = getDirtyPages();
2. safePointer = getSafePointer();
3. workingPointer = getWorkingPointer();
4. vol.Element* tempSafePointer = safePointer;
5. if safePointer < workingPointer then
6.   vol.int startPage = getPage(safePointer);
7.   vol.int endPage = getPage(workingPointer);
8.   while startPage != endPage do
9.     dirtyPages[startPage] |= 1;
10.    flush(dirtyPages[startPage]);
11.   end
12.   tempSafePointer += PAGE_SIZE;
13. end
14. for bit in dirtyPages do
15.   // for each bit that has the value 1 its blocked
16.   // o acesso a respetiva página
17.   if bit == 1 then
18.     mprotect(buffer + (pageSize * positionOf(bit)));
19. end
20. end
21. if safePointer < workingPointer then
22.   tailPointer = headPointer;
23.   while tailPointer->next != NULL do
24.     tailPointer = tailPointer->next;
25.   end
26.   safePointer = tempSafePointer;
27.   workingPointer = safePointer;
28. end

The code protects the signal and corrects the pointer next from tailPointer to NULL, stopping the access to the protected page.

In addition to the described function, rollback of modifications associated with pages that are possibly corrupted is also performed.

5. Experimental Evaluation

After the implementation of the BOPL, it was necessary to create tests, in order to guarantee the good functioning of the algorithm and to quantify possible gains or losses.

The experiments will be carried out focusing on the three implementations described. The first implementation will be indicated as Flush Always, the second one (solution with optimization technique using a persistent diary) will be referred to BOPL-Log and, finally, the third one (solution with optimization technique using a volatile hashmap), will be named BOPL-Hashmap.

To test the solutions, four types of scenarios were created that represent plausible cases and best illustrate the gains and losses of the solutions:

- 10% of Modifications and 90% of Searches;
- 50% of Modifications and 50% of Searches;
- 90% of Modifications and 10% of Searches;
- 100% of Insertions.

These scenarios will be cross-sectional in the tests, whether testing with Optane, NVRAM and with DRAM. In order to test the BOPL, it was necessary two machines:

Specifications Optane machine
- CPU: Intel Xeon E5-2648L v4 @ 1.8 GHz
- DRAM: 32GB DDR4 ECC @ 2.4GHz
- Optane SSD: 32GB M.2 PCIe 3.0
- Kernel: GNU/Linux 4.14.78
- Operating system: Ubuntu 16.04 LTS

Specifications NVRAM machine
- CPU: Intel Xeon Silver 4116 @ 2.1GHz
- DRAM: 64GB DDR4 ECC @ 3.3GHz
- Kernel: GNU/Linux 4.15.0-46 generic
- Operating system: Ubuntu 18.04.01 LTS

Because NVRAM is not yet available for public sale, it can be simulated by introducing latency into code zones that access DRAM.

5.1. Results

The figure 4 shows the execution time of the algorithm. The mean time for the Flush Always solution was 129.860 seconds (σ = 2.517s). As far as BOPL-Log, it was found that improved 92%, with a mean time of 12.262 seconds (σ = 0.008s). Finally, BOPL-Hashmap mode has an averaged of 162.226 seconds (σ = 4.170s), which shows that it’s worse than the traditional (Flush Always) mode. Due to these factors, it is possible to verify that, based on the statistic test One-way Analysis of variance (One-way ANOVA), there was a significant differences between the solutions ($F(2) = 14162, 668, p < 0.001$).

On the 100% insertions scenario, unlike the pre-
vious one, it is only necessary to test the solution **Flush Always** and the simple optimization mode (the BOPL). **Flush Always** had the mean time of 911.919 seconds ($\sigma = 0.517s$) and the BOPL had an average of 20.223 seconds ($\sigma = 0.072s$). Based on these factors, it can be concluded that in this scenario, BOPL presents gains of 98% and considering the statistical test, One-way ANOVA, it is possible to infer that they are also significantly different between them ($F(1) = 1104.550, p < 0.001$).

**Figure 5: Total execution time of 100% insert (Optane)**

On the scenario 100% insertion using NVRAM, like the previous one it is only necessary to compare the two solutions. In the context of PCM, the average time of the Flush Always solution is 73.648 seconds ($\sigma = 1.928s$), whereas for the BOPL is 23.357 seconds ($\sigma = 0.241s$). For the **Spin-Transfer Torque Memory** (STTM), the **Flush Always** solution had a mean time of 54.312 seconds ($\sigma = 1.486s$) and BOPL had 21.261 seconds ($\sigma = 0.101s$). For both the PCM and STTM, the one-way ANOVA test, proves that there are significant differences between the solutions ($F(2) = 582.388, p < 0.001$ and $F(2) = 365.160, p < 0.001$ respectively).

**Figure 6: Total execution time of 100% insert (NVRAM)**

### 6. Discussion
Based on the scenarios that have Optane, the results shows that the BOPL-Log presents better results than the others, thus showing that BOPL is superior in performance. In the NVRAM tests, the BOPL-Log solution only shows significant gains when there is a large percentage of insertions (approximately 100 %). This way, it is possible to conclude that BOPL could show more significant gains for NVRAM, if the data structure used had an optimized search operation, for example, a tree.

### 7. Conclusions
Due to the fact that there are many databases worldwide, they are used in a very large number of applications, both for modification and for research. This way, it was idealized an optimization technique, focusing on batch requests, which could provide better speeds for modifications. Consequently, it was engineered the **Batch-Optimized Persistent List** (BOPL), which allows the flush removal from critical path, improving the usage of the systems. As such, after the completion of this project, it was possible to achieve and identify different objectives. The first objective is related to a research of the state of art, in which it was verified that all the structures and mechanism focus on the persistence and the coherence of the structure in each operation, which implies a performance failure. The second goal was focused on the creation of BOPL and to identify other possible adaptations, since BOPL alone could not guarantee all possible operations. This led to the creation of **BOPL-Log** and **BOPL-Hashmap**. After the implementation of the three solutions (**Flush Always**, **BOPL-Log**, **BOPL-Hashmap**), it was necessary to analyze possible scenarios covering a total set of operations. After the tests were created, they were executed by BOPL in both Optane and NVRAM. This way, it was possible to obtain information about the total execution time, for each tests associated with each solution. Later, an analysis was performed at the time of execution of the tests, which served to conclude that the optimization technique has a very positive impact on the execution time, more specific the BOPL-Log solution.

### Acknowledgments
I would like to thank my supervisor, prof. João Barreto from Instituto Superior Técnico, to the PhD student João Neto, to my family and my dear friends, thank you so much for your support.

### References


