DULL: A Fast Scalable Detectable Unrolled Lock-Based Linked List

Ahmed Fahmy
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Abstract

Persistent memory (PM) has emerged as a promising technology that enables data structures to preserve their consistent state after recovering from system failures. Detectable data structures have been proposed to detect the response of the last operation of a crashed process. Various lock-free detectable and recoverable concurrent data structures have been developed in the literature. However, designing detectable lock-based structures is challenging due to the need to preserve the correctness properties of the underlying locks, such as mutual exclusion and deadlock-freedom, across failures. Therefore, lock-based detectable and persistent data structures are not as common as lock-free structures. In this work, we introduce DULL: a fast, scalable and Detectable Unrolled Lock-based Linked list. This paper presents the design and implementation of DULL, along with an evaluation of its recoverability and scalability. Experimental Results show that DULL is several-fold faster than the competition in all workloads that involve updates. Moreover, as opposed to some of the previous works, our algorithm is scalable when the multiprocessor is oversubscribed. DULL is a demonstration of the feasibility of using lock-based data structures with detectability in PM environments. We believe that DULL opens up new research directions for designing and analyzing detectable lock-based data structures.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Software and its engineering → Mutual exclusion; Computer systems organization → Reliability

Keywords and phrases detectability, lock-based, mutual exclusion, linked list, fault-tolerance, persistent memory, concurrency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2024.6

Funding This research is supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council of Canada (NSERC).

Acknowledgements We would like to thank the OPODIS review committee for their valuable comments and helpful suggestions.

1 Introduction

Non-volatile random access memory, or *persistent memory* (PM), has been a research focus for the past few years. Such a technology enables computer systems to maintain data in memory across transient failures, such as power outages. The evolution of PM blurs the line in current computer architectures between the primary *dynamic random-access memory* (DRAM) and the secondary long-term storage (e.g., solid-state drives). In lieu of utilizing the DRAM for high-speed memory access and the secondary storage for data recovery, the PM promises the best of the two worlds: performance and durability. PM is byte-addressable like the DRAM, allowing computer programs to directly recover in-memory data structures without transferring data from the block oriented secondary storage. The durability of PM created the opportunity for designing data structures that are aware of the underlying memory hardware to leverage its recovery feature.

With the emergence of PM, data structure algorithms are potentially able to preserve their consistent state after recovering from a crash failure. Nowadays, most multiprocessors use the *cache-coherent* (CC) memory model, where each processor has a local cache while all processors share memory modules that can be accessed through some interconnect. Designing recoverable, or persistent, data structures for CC systems is challenging. In such systems, caches constitute a volatile level in the memory hierarchy that falls between the processing units and the PM. If a data structure changed its state based on data in a volatile cache that is not yet written back to memory, a subsequent system-wide failure erases those data, and recovering from that crash naively might lead to inconsistency. Therefore, persistent data structures should be designed carefully to establish consistency. For example, instructions for writing back cache lines and memory fences can impose order in which data are written into memory. Intel has introduced a new technology on the 3rd generation Xeon processors, called the extended asynchronous DRAM Refresh (eADR), that includes CPU caches into the system-wide failure protected domain [33]. Such a technology removes the need for cache line write-back instructions. However, the protected domain excludes the CPU volatile registers. Hence, memory fences and recovery protocols are still required to ensure consistency.

Many recoverable concurrent data structures have been presented [52, 17, 6, 35, 19, 22, 51, 3]. The strongest correctness property for such structures is *strict linearizability* [1] as it requires crashed operations to either take effect before the crash or never take effect. Some data structures are said to be detectable [5, 7, 8, 22], which enables detecting the response of a crashed operation. Published lock-based detectable and persistent data structures [13, 31, 45, 49] are not as common or practical as lock-free structures. For example, some algorithms [13, 31] use logging, which degrades performance. Others [45, 49] adopt privileged persistence instructions (e.g. flush-the-whole-cache) and specific techniques that limit scalability, such as flat-combining.

Designing detectable lock-based data structures is challenging. The correctness properties of the underlying locks, such as mutual exclusion and deadlock-freedom, must be preserved across failures when adding persistence or detectability. In this work, we introduce DULL: a fast, scalable and Detectable Unrolled Lock-based Linked list.

2 Literature Review

Various recoverable concurrent data structures have been presented in the literature [52, 17, 6, 35, 19, 22, 51, 3]. Researchers have established recoverability with different techniques. (see Appendix A for more details). Herlihy and Wing's *linearizability* [30], a widely used correctness property for concurrent data structures, does not consider crash failures explicitly. Therefore, other variants of the correctness property have been proposed to accommodate failures [1, 27, 10, 9]. *Strict linearizability* [1] is the strongest among the other variants as it requires crashed operations to either take effect before the crash or never take effect. In addition, it is compatible with different process naming schemes, including the notion that a process is resurrected after a failure and attempts to detect the outcome of its last operation. We adopt this correctness property in our work.

Some data structures are said to be detectable [22, 5, 7, 8], which is an additional feature on top of recoverability that enables detecting the response of a crashed operation. In, [5] and [8], Attiya et al. and Ben-David et al. presented a detectable persistent CAS synchronization primitive that uniquely identifies each operation. One of the queues proposed by Friedman in [22] offered detectability at the expense of using bookkeeping logs. The lower and upper bound of the space complexity associated with achieving detectability was investigated by Ben-Baruch et al. [7]. They also presented a detectable CAS with bounded space complexity. Until recently, there was no formal definition for detectability. Attiya et al. in [5] defined detectability through a correctness property they have proposed, *nesting-safe recoverable linearizability* (NRL). NRL assumes that a crashed process can be restarted from its innermost recoverable operation, which requires some kind of persistent call stack. Additionally, it does not consider the possibility that a crash failure may occur outside of any recoverable operation. The applicability of such assumptions is not clear in a practical setting. Li and Golab [42] formalized a more robust definition by providing a systematic approach of obtaining a *detectable sequential specification* (DSS) of an object type from its non-detectable sequential specification. The resultant sequential specification has to be paired with a correctness property, like strict linearizability, as opposed to other detectability definitions [44]. Using DSS, [42] presented a detectable lock- and log-free queue. Based on Li and Golab's work, Moridi et al. [44] proposed a new correctness definition, called *unified detectable sequential specification* (UDSS), that has a simpler interface at the small expense of performance. In addition, they presented a black box approach that transforms a DSS object into a UDSS one.

Researchers are currently active in developing persistent and detectable lock-free algorithms [3, 4, 56, 34, 52, 21, 23]. Attiya et al. built on top of the work in [3] and presented in [4] a technique to obtain a detectable persistent implementation of lock-free data structures. Zuriel et al. [56] designed an efficient persistent concurrent set. Izadpanah et al. [34] proposed PETRA, an optimization to *persistent transactional memory* (PTM) introduced by Wang et al. [54]. NVTraverse [21] is a general transformation by Friedman et al. that automatically makes a lock-free data structure persistent. However, it requires changing its representation into *a traversal form*, which needs programming expertise and potentially degrades performance [23]. Friedman et al. [23] presented another transformation, called Mirror, where a copy of the base data structure is stored in each of the DRAM and PM, resulting in a faster lock-free recoverable data structures than NVTraverse. It uses atomic operations and sequence numbers to ensure consistency and durable linearizability. Mirror is specifically designed for lock-free algorithms and is not suitable for lock-based structures.

Published lock-based detectable and persistent data structures are less common and practical compared to lock-free structures. Atlas [13], by Chakrabarti et al., identifies failure-atomic regions using locks in lock-based data structures to simplify recovery, but the intensive use of logs in critical sections degrades both performance and recovery time. Ching-Hsiang et al. [31] use mutexes to coarsely divide the code into synchronization points of consistent states, recording buffered writes between these points to reduce logging overhead. Nawab's Dali hashmap [45] is lock-based but relies on the privileged x86 instruction to flush the entire cache, often flushing and invalidating useful cache lines. A detectable lock-based flat-combining technique [49] was proposed by Rusanovsky et al., providing a generic method to build detectable lock-based data structures. Flat-combining involves a single thread performing operations submitted by other threads, utilizing the cache more efficiently but limiting scalability. Appendix A provides further discussion of related works.

3 System Model

This section describes the system model adopted in this work. Our focus is on the *cache-coherent* (CC) model where each process stores a local copy of the variable in its local cache for faster subsequent access to the data. Data in volatile caches are not guaranteed to be written back to the memory unless explicit cash-line write-back instructions are used. In addition, the order of writing back cache lines to the memory is not ensured. The order can be enforced by using memory fences. In current computer architectures, a flush instruction in C/C++ is implemented as a cache-line write-back intrinsic followed by a store fence.

An algorithm for a data structure can be executed concurrently by N asynchronous processes, or threads, $(p_1, p_2, ..., p_N)$ prone to crash failures. A process executing an algorithm can take three types of *steps*: an ordinary step, a flush step or a (system-wide) crash step. An ordinary step is an operation in which a process accesses a shared variable. A flush step is a persistence operation that writes the contents of a cache line back to memory and then imposes a memory fence and may (or may not) invalidate the current cache state. A crash step is a system-wide failure that resets all processes to their initial state and clears all volatile caches.

We use the notion of *unified detectable sequential specification* (UDSS) [44] to define detectable object semantics. A UDSS object is a (base) object that is equipped with an auxiliary procedure to *resolve* each of the object operations. Specifically, for each operation *op* of the base object, a procedure **Detectable-*op*** is defined to perform *op* on the object in a detectable manner. For convenience, we use *op* to represent **Detectable-*op*** in this work. In addition, the auxiliary procedure **Resolve** is defined to resolve a process' last *detectable* operation, if it exists. The auxiliary procedures enable recovering detectable objects to resolve operations interrupted by crash failures and potentially complete the operations.

4 Persistent Objects

Before getting into the algorithmic details of DULL, three persistent objects are defined to be used as building blocks: *PMptr* is a pointer designed to work with *persistent memory mapped files*, *PMpair* is a tuple to store a key-value pair atomically in the PM, and *descriptor* is an object that stores information related to the last operation initiated, which may or may not be completed, by a process to facilitate list detectability.

4.1 Persistent Pointers

A persistent pointer, PMptr, handles two problems: *address space layout randomization* (ASLR) [32] and efficiently persisting pointers. As a result of ASLR, dereferencing a pointer on PM that stores the address of a persistent object after recovering from a system failure might result in accessing an invalid memory location. The PMptr compensates for ASLR by storing each pointer as an offset from the base address of the persistent memory-mapped file and calculating the pointer value on every access. Persisting the offset upon every read and write is a conservative approach to ensure correctness. However, this might be inefficient since each update needs to be flushed only once to the PM for a subsequent recovery. PMptr persists the offset more efficiently. The idea is borrowed from the work of Wang et al. in [53]. The *offset* variable is a tuple comprising the offset value intended to be persisted and a "dirty bit" that indicates whether the current offset value has ever been persisted (0) or not (1). A PMptr object can be accessed through two operations: *Set and Persist* (**SaP**) and *Get and Persist* (**GaP**). See Appendix B for PMptr implementation details.

4.2 Persistent Pairs

A persistent key-value pair, PMpair, comprises a single variable and three associated functions. The variable *pair* is a tuple $\langle key, value, flag \rangle$, where *key* and *value* are variables that store a key and its corresponding data, respectively, and *flag* is a dirty bit set to 1 if *key* and *value* are not persisted and 0 otherwise. Accessing *pair* is governed by three functions: **Set**, **Get** and **Persist**. The function **Set** stores a key-value pair, given as an argument, into *pair* and marks the tuple as not persisted. The key and value stored in *pair* are retrieved by the procedure **Get**, ignoring the dirty bit state. Neither function flushes the key-value pair object. Persisting *pair* is delegated to a separate procedure, **Persist**, for performance purposes. See Appendix C for more implementation details of the PMpair object.

4.3 Descriptors

A descriptor object is defined in Algorithm 1. To achieve detectability, each process accessing an instance of DULL has an auxiliary descriptor object. At the beginning of a detectable operation, a process shall store some operation-related information in its descriptor, i.e. *prepare* the operation, through the function **Prep-***op*. Preparing an operation is designed such that all changes to the descriptor variables *appear to be atomic*. That is, either all changes take effect or none at all. To achieve atomicity, each descriptor object consists of two instances of its variables, called *units*, and a Boolean variable, *curr*, that points to the *current unit*, where the other instance is called the *next unit*.

Algorithm 1 Variables and functions of a *descriptor* object.

```
Define Unit
    op: int ∈ {Insert, Remove, ⊥}, init ⊥
    args: tuple ⟨key, value⟩, init ⟨⊥, ⊥⟩
    result: int ∈ {true, false, ⊥}, init ⊥
    offset: pointer of a node, init 0

Define Descriptor
    units[0, 1]: array of struct Unit
    curr: int ∈ {0, 1}, init 0

Procedure SetResult (res)
1   units[curr].result := res                          // Set result
2   Flush (units[curr].result)                         // Persist to PM

Procedure SetNode (ptr)
3   units[curr].offset := ptr − BaseAddress
4   Flush (units[curr].offset)                         // Persist to PM

Procedure Prep-op (key, value)
5   next := 1 − curr
6   units[next].op := op                               // init next unit
7   units[next].args := ⟨key, value⟩
8   units[next].result := ⊥
9   units[next].offset := 0
10  Flush (units[next])                                // Persist next unit
11  curr := next                                       // make next unit current
12  Flush (curr)                                       // Persist curr to PM

Procedure Resolve
    // get resolved state
13  op := units[curr].op
14  ⟨key, value⟩ := units[curr].args
15  result := units[curr].result
16  return ⟨op, key, value, result⟩
```

In **Prep-***op*, the next unit variables are initialized with the appropriate parameters (Lines 6-9), then *curr* is flipped to switch the next and current units (Line 11). Note that, all the descriptor variables in the next unit are persisted to PM at Lines 10 and 12 before the next unit becomes the current one. The preparation process is marked as completed only if *curr* is updated. If a crash occurs before persisting *curr*, then re-invoking the function **Prep-***op* upon recovery repeats the whole procedure, making it failure-atomic. The function **SetNode** is used for check-pointing (more in Section 4.6), and **SetResult** marks the end of the prepared operation (*op*) by storing the result in the descriptor. Furthermore, a process can resolve its last operation at any point in time by invoking the idempotent **Resolve** procedure of its descriptor object.

This section presents the implementation details of DULL. Our detectable implementation is based on the unrolled linked list by Platz et al. [46]. Although our focus is on persistence and detectability, a full description of the proposed DULL algorithm, including overlaps with the algorithm in [46], is provided in this section for completeness. In the search, insertion, and removal operations, the lines of added or modified steps are highlighted in red, and corresponding line numbers are underlined to clearly distinguish our contributions from the original work. The remaining operations are key components of our contribution.

4.4 Linked List Structure

In designing DULL, persistent unrolled nodes are used as building blocks. Node unrolling, initially proposed by Shao et al. [50], involves the storage of multiple elements in an array within each node. Each node comprises the following variables:

- *count*: the number of key-value pairs stored in the node.
- *marked*: a Boolean indicating whether the node has been logically removed from the list.
- *m*: a volatile mutex lock used only during insertions and removals in the list.
- *anchor*: the key that uniquely characterizes a node such that if a key exists in a node, then the key $\geq$ the *anchor* of that node and $<$ the *anchor* of the successor node. Nodes are thus sorted in ascending order by their anchor keys.
- *next*: a persistent pointer (PMptr) to the successor node.
- *pairs*$[0..K-1]$: an array of up to K unordered variables of type PMpair, where K is the maximum number of pairs in a node.

In addition to persistent nodes, a DULL instance has an array $D[1..N]$ of N descriptor objects, where N is the number of processes. Moreover, a DULL object has at least three persistent nodes at all times: the head and tail sentinel nodes and one non-sentinel node. Sentinel nodes do not store key-value pairs. The head and the non-sentinel node anchor keys are $-\infty$, and the tail anchor key is ∞, where $-\infty$ is some key $<$ the minimum key in the key space, and ∞ is some key $>$ the maximum key in the key space.

4.5 Traversal and Search

Traversing a DULL is separated into two functions: **Scan** and **Seek**. Both procedures are presented in Algorithm 2. The function **Scan** searches the list for a target *key* starting from the first non-sentinel node in line 18, while tracking the predecessor (*pred*) and successor (*succ*) of each (*curr*) node. The search ends by reaching a node, *curr*, where the target key might exist, i.e., the successor's anchor key is greater than the target key (Line 20). Finally, the function **Scan** returns the three tracked pointers at Line 24. The while-loop at Line 20 is guaranteed to terminate since the tail's anchor key is greater than all keys in the key space. The procedure **Seek** takes a pointer to some node *curr* and a target key, and it linearly searches for the key in the array *pairs* of *curr* (Line 25). If the target key is found, the function returns the index of its key-value pair at Line 29, otherwise, the procedure returns $\perp$ at Line 30, indicating that the key is absent in *curr*.

A **Contains** operation, presented in Algorithm 2, searches for the node, *curr*, that potentially contains the argument target key by invoking the procedure **Scan** at Line 31. The array *pairs* of the node *curr* is searched for the target key by calling the function **Seek** at Line 32. If the key is found, **Seek** returns a valid index (Line 33) of the corresponding pair, which is atomically read and then persisted at Lines 34 and 35, respectively. The key of the read pair is checked at Line 36 before returning its value at Line 37. If **Seek** does not find *key* and returns $\perp$, or the check at Line 36 fails, the operation returns $\perp$ at Line 38, indicating that the list does not contain the target key. Note that all traversed pointers are flushed at least once through the function **GaP** at Line 23, and each pair read is persisted through the **Persist** function at Line 27. Additionally, the function **Contains** persists the found pair (Line 35) after the atomic read at Line 34. All persistence operations occur before returning from the corresponding procedures to preserve strict linearizability.

Algorithm 2 List traversal.

Procedure Scan (key)

```
17 $pred := head.$GaP()                                // Skip head
18 $curr := pred.next.$GaP()
19 $succ := curr.next.$GaP()
20 while $succ.anchor \leq key$ do
21     $pred := curr$
22     $curr := succ$
23     $succ := succ.next.$GaP()
24 return $(pred, curr, succ)$
```

Procedure Seek $(curr, key)$

```
25 for $j := 0$ to $K - 1$ do
       // Read pair atomically
26     $\langle k, v \rangle := curr.pairs[j].$Get()
27     $curr.pairs[j].$Persist()
28     if $k = key$ then
29         return $j$                                  // Return index if found
30 return $\bot$                                       // Otherwise return Nil
```

Procedure Contains (key)

```
31 $(\_, curr, \_) :=$ Scan$(key)$                     // Get the containing node
32 $j :=$ Seek$(curr, key)$                            // Get the index of key in curr
33 if $j \neq \bot$ then                               // If key is found
34     $\langle k, v \rangle := curr.pairs[j].$Get()   // Read pair atomically
35     $curr.pairs[j].$Persist()                       // Persist pair to PM
36     if $k = key$ then                               // Return value if key still in pairs
37         return $v$
38 return $\bot$                                       // Otherwise return Nil
```

4.6 Insertion

Given a pair, *key* and *value*, an **Insert** operation, shown in Algorithm 3, calls **Prep-Insert** of the executing process' descriptor to initialize it at Line 39. Subsequently, the operation finds the appropriate node (*curr*) to store the pair, and its predecessor (*pred*), by calling the function **Scan** at Line 40. The predecessor lock is acquired at Line 41 to prevent any simultaneous update to *curr*, *pred* and the linking pointer, *pred.next*. The validation in [29] is then applied to check that *pred.next* still points to *curr* and that neither node is marked as removed (Line 42). If the validation fails, then the predecessor's mutex is released and the insertion operation is restarted. Otherwise, Line 45 calls the function **Seek**, searching for *key* in *curr*'s array. If the returned index (j) from **Seek** is valid, then the function marks the insert operation as completed, unlocks the predecessor's lock and returns false (Lines 47-49), indicating that the key already exists.[1] Otherwise, an empty slot in the array *pairs* is searched for by the function **Seek** at Line 51. Consequently, there are two possible scenarios: (1) There is an empty slot in *pairs* and **Seek** returns its index, or (2) the array is full and **Seek** returns $\bot$. If (1), then the empty slot in *pairs* is set to *key* and *value* and is persisted at Lines 53 and 54, respectively, and the counter of *curr* is incremented at Line 55. If (2), then *curr* is split into two nodes, new_1 and new_2, using the helper procedure **Split**, called at Line 57. The **Split** function creates two new nodes (new_1 and new_2) to replace *curr*, where each stores roughly half of the pairs in *curr*. The implementation details of the **Split** function are presented in Appendix E.

Subsequently, the given pair is inserted into new_1 if the pair's key is strictly less than the anchor key of new_2. Otherwise, the pair is inserted into new_2. In both cases, the counter of the node where the pair is inserted is incremented. Note that if the target key is not found at Line 45, it will eventually be inserted regardless of whether case 1 or 2 occurs. To maintain such an effect in the presence of a crash failure, the pointer *pred* is stored in the descriptor $D[i]$ as a checkpoint marker at Line 50. That is, if a process p_i crashes after persisting $D[i].offset$ (Line 4), the operation can be resumed during recovery (Section 4.8).

[1] Instead, the value in a found pair can be updated and the algorithm remains correct.

Algorithm 3 Key-value pair insertion.

Procedure Insert $(key,\ value)$

```
39  D[i].Prep-Insert(key, value)
    while true do
40      (pred, curr, _) := Scan(key)
41      pred.m.Lock()        // Lock the predecessor node
        /* Validate curr is the pred of next and that
           both are not deleted                              */
42      if pred.marked ∨ curr.marked ∨ pred.next.GaP() ≠ curr then
            // If not valid, unlock pred and retry
43          pred.m.Unlock()
44          continue
45      j := Seek(curr, key)     // Get the index of key
46      if j ≠ ⊥ then             // If the key is found
            // Mark op as completed
47          D[i].SetResult(false)
48          pred.m.Unlock()
49          return false
        // Set pred in descriptor
50      D[i].SetNode(pred)
51      j := Seek(curr, ⊥)              // Get empty slot
52      if j ≠ ⊥ then                   // If found
            // Set key-value pair
53          curr.pairs[j].Set(key, value)
54          curr.pairs[j].Persist()
55          curr.count := curr.count + 1
        else
56          curr.m.Lock()            // lock curr if full
            /* Split into new1, new2 and insert pair
               into the appropriate node                    */
57          (new_1, new_2) := Split(curr)
58          if key < new_2.anchor then
59              new_1.pairs[new_1.count].Set(key, value)
60              new_1.count := new_1.count + 1
            else
61              new_2.pairs[new_2.count].Set(key, value)
62              new_2.count := new_2.count + 1
            // Persist and logically insert new nodes
63          new_1.PersistNode()
64          new_2.PersistNode()
65          pred.next.SaP(new_1)
66          curr.marked := true
67          curr.m.Unlock()
        // Mark op as completed
68      D[i].SetResult(true)
69      pred.m.Unlock()
70      return true
```

Finally, *curr* is marked to prevent erroneous subsequent updates to the removed node (Line 66), the locks in *curr* and *pred* are released (Lines 67 and 69), and the value **true** is returned (Line 70) to indicate that the pair is successfully inserted into the list. Before the procedure **Insert** returns, p_i marks its operation as completed by setting the result field of its descriptor object through calling **SetResult** at Line 68.

To ensure recoverability, the next pointers of the new replacement nodes are persisted in the **Split** function. The other variables are persisted using **PersistNode** (Lines 63 and 64) before the nodes are inserted and accessible by other threads (Line 65), maintaining atomicity in the split process. **PersistNode** sequentially issues a cache-line write-back for each node variable, followed by a store fence. If the system fails before replacing *curr* with the new nodes (Line 65), then the incomplete split can be, optionally, re-invoked upon recovery since it has not taken effect. If a failure occurs after Line 65 takes effect, then the recovery protocol (Section 4.8) recovers the new nodes.

4.7 Removal

Algorithm 4 shows the procedure of removing a key-value pair from a DULL instance. First, the descriptor of the executing process is initialized at Line 71. Given a target key, invoking **Remove** finds the node (*curr*), and its predecessor (*pred*), that potentially stores the key via the function **Scan** at Line 72. The lock in *pred* is acquired at Line 73 to prevent any simultaneous update to the *curr* and *pred* nodes. The algorithm validates that the next pointer or *pred* still points to *curr* and that both nodes are not marked as removed (Line 74). If the validation fails, then the mutex of *pred* is unlocked and the removal operation is restarted. If the validation succeeds, the function **Seek** searches for the target key at Line 77. If the key is not found, then the function marks the operation as completed, unlocks the lock of *pred* and returns **false** (Lines 79-80). Otherwise, p_i stores the offset of *pred* in its descriptor object at Line 81 to detect and optionally resume the operation upon recovery (Section 4.8). Then, the pair containing the target key is removed and persisted at Lines 81 and 82, respectively, and the node's counter is decremented at Line 82.

Algorithm 4 Key-value pair removal.

Procedure Remove (key)

```
71  D[i].Prep-Remove(key, $\bot$)
    while true do
72      $(pred, curr, \_) :=$ Scan(key)
73      pred.m.Lock()
        /* Validate curr is the pred of next and that
           both are not deleted                                */
74      if $pred.marked \lor curr.marked \lor pred.next.$GaP() $\neq curr$ then
75          pred.m.Unlock()
76          continue
77      $curr :=$ Seek(curr, key)
78      if $curr = \bot$ then
            // mark op as completed
79          D[i].SetResult(false)
            pred.m.Unlock()
80          return false
        // Set pred in descriptor
81      D[i].SetNode(pred)
        curr.pairs[curr].Set($\bot, \bot$)          // Remove pair
82      curr.pairs[curr].Persist()
        $curr.count := curr.count - 1$
83      if $curr.count <$ MinFull then
84          curr.m.Lock()
85          $succ := curr.next.$GaP()
86          if $curr.count = 0 \land curr.anchor \neq -\infty$ then
                // If empty and not last non-sentinel
87              pred.next.SaP(succ)               // Remove curr
88              $curr.marked :=$ true
89          else if $succ.anchor \neq +\infty$ then
                // otherwise, merge or redistribute
90              succ.m.Lock()
91              if $curr.count + succ.count <$ MaxMerge then
92                  $new_1 :=$ Merge(curr, succ)
                else
93                  $(new_1, new_2) :=$ Redist(curr, succ)
94                  $new_2$.PersistNode()
95              $new_1$.PersistNode()
96              pred.next.SaP($new_1$)
97              $curr.marked :=$ true
98              $succ.marked :=$ true
99              succ.m.Unlock()
100         curr.m.Unlock()
        // mark op as completed
101     D[i].SetResult(true)
102     pred.m.Unlock()
103     return true
```

Consequently, if the counter is less than a predefined constant (MinFull $\leq K/2$), some actions are taken to reduce the variation in the number of pairs across all nodes. In the general case where *curr* is not the only non-sentinel node and its successor (*succ*) is not the tail, there are three possible scenarios: (1) *curr* is empty, (2) *curr* is not empty, and the sum of the counters of *curr* and *succ* is less than some predefined constant (MaxMerge), or (3) *curr* is not empty, and the sum of the counters of *curr* and *succ* is greater than or equal to MaxMerge. If (1), then *curr* is logically removed (Line 87) from the list and marked to prevent invalid subsequent changes to the node. If (2), then *curr* and *succ* are merged into a new node by invoking the helper function **Merge** at Line 92. If (3), then the pairs stored in *curr* and *succ* are redistributed into two new replacement nodes, by calling the helper procedure **Redist** at Line 93, such that both nodes contain roughly the same number of pairs. Due to space constraints, the steps for the helper subroutines **Merge** and **Redist** are moved to Appendix F. In cases (2) and (3), the new nodes are persisted and logically inserted into the list (Lines 94-96), and *curr* and *succ* are marked at Lines 97 and 98, respectively. Finally, at Lines 99-103, all acquired locks are released, the operation is marked as completed, and the **Remove** procedure returns true, indicating that the pair was successfully deleted.

4.8 Recovery

The recovery protocol, defined in Algorithm 5, is a procedure that is invoked, by a single thread, after a crash failure. First, D, *head* and *tail* are initialized to the addresses of the descriptors array, head node and tail node (Lines 104-106), respectively (refer to Appendix D for details about the memory mapped file layout). Then, the recovered head and tail are used to traverse the list (Line 108) and recover the variables of each node.

Adding persistent nodes to the list – which may occur when inserting or removing a key – comprises creating new nodes and persisting their *anchor*, *pairs* and *next* variables before the new nodes are logically inserted into the list. Throughout a node life-cycle, *anchor* does not change after it is first initialized. As for the *next* pointers, traversing the list (Section 4.5) is designed such that the value of a *next* pointer is persisted by the **GaP** function upon reading. In addition, each element in the array *pairs* is persisted on update. As a result,

Algorithm 5 Recovery protocol.

Procedure Recover

```
104 $D$ := BaseAddress + 8                          // Recover descriptors
    // Recover head
105 $head := D + N \times$ SizeOf(Descriptor)
106 $tail := head +$ SizeOf(Node)                   // Recover tail
107 $ptr := head$                     // Recover nodes from head to tail
108 while $ptr \neq$ BaseAddress do                 // While $ptr \neq \bot$
109     $count := 0$                 // Recover the current node counter
110     for $i := 0$ to $K - 1$ do
111         if $ptr.pairs[i]$.Get() $\neq \langle \bot, \bot \rangle$ then
112             $count := count + 1$
113     $ptr.count := count$
114     $ptr.m :=$ new Mutex                        // Reset the node lock
115     $ptr := ptr.next$.GaP()
116 $incompleteOp :=$ new Map                       // Map nodes to PIDs
117 for $i := 1$ to $N$ do                          // For every process i
118     $tmp := D[i].units[D[i].curr]$              // Get current unit
119     $pred := tmp.offset +$ BaseAddress          // Get pred
120     $\langle op, key, value, res \rangle := D[i]$.Resolve()
121     if $res = \bot \wedge op \neq \bot \wedge pred \neq$ BaseAddress then
122         $incompleteOp[pred] := i$               // Map pred to PID
123     else if $res = \bot \wedge op \neq \bot$ then   // no effect
124         $D[i].curr := 1 - D[i].curr$            // clean descriptor
125 foreach $pred \in$ list of nodes from tail to head do
126     if $pred \in incompleteOp$ then
127         $curr := pred.next$.GaP()
128         $succ := curr.next$.GaP()
129         $i := incompleteOp[pred]$               // Get PID
130         $\langle op, key, value, res \rangle := D[i]$.Resolve()
131         $curr :=$ Seek($curr, key$)             // find key in curr
132         if $op =$ Insert then
133             if $succ.anchor < key \vee curr \neq \bot$ then
                    // If split is done or key in curr
134                 $D[i]$.SetResult(true)
                else                                // Resume insertion
135                 Execute Lines 51-70 of Insert except for
                    (un)locking mutexes
            else
136             if $index \neq \bot$ then           // If found, remove pair
                    Execute Lines 81-82 of Remove
                // Resume merge/redistribute if needed
137             Execute Lines 83-103 of Remove except for
                (un)locking mutexes
```

a node is simply recovered by resetting its mutex (Line 114) and deriving the value of the variable *count*. A node's *count* is inferred by traversing the array *pairs* and counting all valid key-value pairs (Lines 109-113).

After initializing the list variables from PM, the recovery protocol uses the recovered descriptors to resolve and resume interrupted operations. Locks properties should not be violated during recovery. That is, resuming incomplete operations in arbitrary order violates strict linearizability. For example, consider the following problematic scenario: Suppose there exist two processes, p_i and p_j, that are performing an insertion on two consecutive fully-filled nodes such that the *curr* node of p_i is the *pred* of p_j. Since the *curr* nodes of p_i and p_j are full of key-value pairs, both processes split their *curr* nodes (Line 52), or crash. During a failure-free execution, p_i would split its *curr* node only after p_j finishes splitting its *curr* node and releases its *pred*'s lock (Line 69), which is *curr.m* of p_i. Such a sequential order of execution, imposed by the mutex locks, prevents invalid states from occurring. Now, suppose a crash failure occurs after both processes store and persist their *pred* pointer values into their descriptors (Line 50) but before linking their split nodes at Line 65. Upon recovery, if p_i splits its node first, deleting its *curr* node (i.e. p_j's *pred* node), then p_j would create two nodes, new_1 and new_2, while splitting its *curr* and erroneously set the next pointer of the removed *pred* node to new_1 (Line 65). Therefore, the sequential execution imposed by the locks must be preserved to avoid inconsistencies. To solve this problem, we observe that locks of consecutive nodes are always acquired in one direction, from head to tail, and always released in the opposite direction. Hence, resuming incomplete operations in the reverse order of the corresponding *pred* nodes position in the linked list (Lines 117-137), from tail to head, guarantees that all locks are released before being acquired by other processes.

The for-loop at Line 117 iterates through each descriptor object and reads its current unit (Line 118). From each descriptor unit, the last operation of each process is resolved (Line 120) and then checked if it needs to be resumed (Line 121). Incomplete operations (i.e. its $result = \bot$) of which the corresponding variable *offset* is initialized with a valid node address shall be resumed. Such operations are logged in an associative array such that each predecessor node address is mapped to the ID of the process by which the operation was being executed when the crash failure occurred. Subsequently, the for-each-loop at Line 125 resolves (Line 130) and resumes the interrupted operations (Lines 132-137) in the reverse order of the associated predecessor nodes in the linked list. For each insertion operation, if the successor's anchor key is less than the key to be inserted, then the node split, and the key insertion, must have been done before the crash. If the key has been found in *curr*, then the key has been inserted without splitting the node. In both cases, the recovery protocol just marks the operation as completed (Line 134). Otherwise, the **Insert** procedure is resumed starting from Line 51 without accessing the locks. As for interrupted key removals, the key is removed at Line 135 if it is found upon recovery (Line 136). Then, Lines 83-103 of the function **Remove** is executed to merge or redistribute if necessary. Similarly to insertion operations, steps for acquiring and releasing locks are skipped when resuming key removals. Note that executing Lines 83-103 is idempotent. That is, the checks at Lines 83 and 91 ensure that the nodes are redistributed or merged at most once.

4.9 Analysis

▶ **Theorem 1.** *DULL is strictly-linearizable and deadlock-free.*

The proof is omitted due to space consideration.

5 Experimental Evaluation

The recoverability of the proposed algorithm is tested using a strict-linearizability checker, and its performance is evaluated against the state-of-the-art lock-free and lock-based solutions [18, 48, 49, 28], as well as its non-recoverable variant [46]. The constants MinFull and MaxMerge are set to $K/4$ and $3\times K/4$, respectively, and $K = 8$ is used in all our experiments. All lists are implemented in C++ with the `-funroll-loops` optimization flag using gcc 9.3.0 on Ubuntu 20.04.1. A flush instruction is implemented as a CLWB (`_mm_clwb`) compiler intrinsic followed by a memory store fence (`_mm_sfence`). The server used in our experiments has a 2.1 GHz Intel Xeon processor that has 20 physical cores, in which CLWB instructions are supported. Turbo boost is disabled to obtain consistent evaluation results. Moreover, the server has 6 Intel Optane modules of 128GB, PM is accessed using the PMDK 1.7 library through memory-mapped files in App-Direct mode. The performance of the following list implementations is evaluated:

- **Vanilla-ULL-NONE-PM**: the original, non-recoverable, unrolled linked list [46], runs on PM with no memory reclamation.
- **DULL-DEBRA-PM**: our proposed algorithm runs on PM and uses Brown's distributed EBR variant, DEBRA [12], for node reclamation, with a custom bitmap allocator. Upon recovery, the allocator is reset and reinitialized by traversing the list to prevent memory leaks from crashes. We also replace our hand-tuned persistence mechanism (based on [53]) with the FliT library [55] and compare its performance to our custom solution.
- **DFC-ULL-NONE-PM**: a detectable unrolled linked list implemented using the flat-combining technique proposed in [49]. In DFC-ULL, insertions and removals are submitted to a global array and then handled by a combiner thread. DFC-ULL runs on PM with no memory reclamation.
- **OneFile-LL-PM**: a recoverable singly-linked list implemented using a lock-free persistent transactional memory (PTM) algorithm [48], where updates are performed using transactions, each has a unique ID. Inserting, or removing, an item and updating the transaction ID occurs atomically using *double-word-compare-and-swap* (DCAS). OneFile-LL runs on PM and uses transactions to reclaim nodes.

- **Romulus-LL-PM**: A recoverable singly-linked list uses a lock-based PTM algorithm [18]. Romulus maintains two copies of the list. Updates modify the primary copy first, with the other updated afterward. In case of a crash, the consistent copy is used for recovery. Romulus-LL runs on PM and uses transactions for node reclamation.
- **Harris-LL-Hazard-DRAM**: the non-recoverable Harris' lock-free list, runs on DRAM and uses Michael's lock-free Hazard pointers [43] to reclaim nodes.
- **Harris-LL-DEBRA-DRAM**: Harris' lock-free linked list, runs on DRAM and uses Brown's distributed variant of EBR, DEBRA [12], to reclaim nodes.
- **PMDK-LL-PM**: a recoverable singly-linked list that is based on the undo-logging algorithm provided by the PMDK object library (`libpmemobj`). PMDK runs on PM and uses transactions to reclaim nodes.

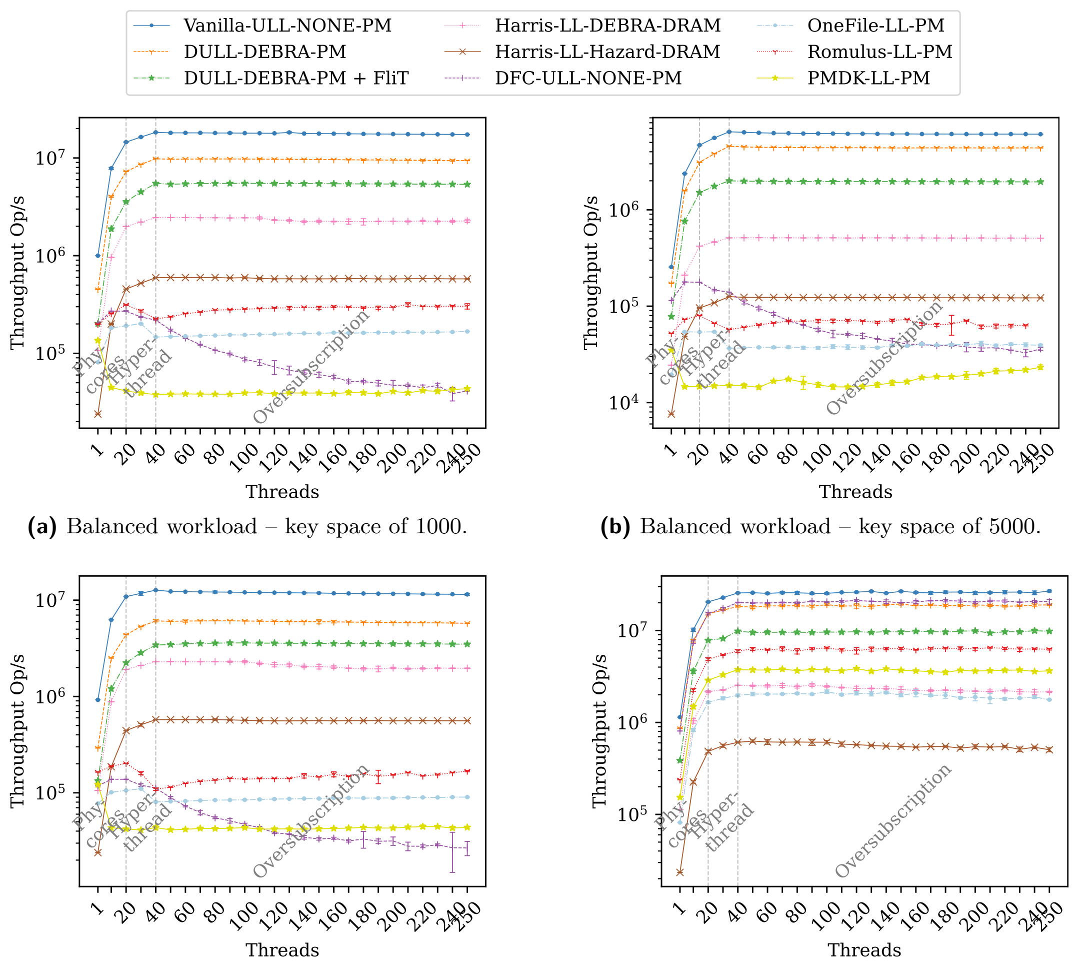
The source code for OneFile, Romulus, and PMDK was obtained from [47], and the source code for FliT was acquired from [16]. Note that only DULL and DFC-ULL are detectable, while all lists are recoverable except for the Vanilla-ULL and Harris lists. Although the Vanilla-ULL is not recoverable, its variables are stored in PM to assess the overhead of DULL. Our evaluation includes experiments conducted under various workloads: an all-read workload, comprising 100% search operations; a balanced workload, with 25% insertions, 25% removals, and 50% searches; and an all-write workload, consisting of 50% insertions and 50% removals. In all experiments, the lists are pre-filled with random key-value pairs to reach their *expected size*. The expected size of a list is estimated based on the key space and the ratio of insertions to removals. Mathematically, the expected size is $\frac{\text{Insertions}(\%)}{\text{Removals}(\%)+\text{Insertions}(\%)} \times$ Key Space. Additionally, various key spaces are used to evaluate list performance.

5.1 Results

When evaluating performance we focus on crash-free executions[2] as applications are expected to run free of failures most of the time. Flush instructions are used in our failure-free experiments to reflect the lists' practical performance. Each run involves multiple threads that continuously insert, remove and lookup key-value pairs for a specific period of time. The metric used for evaluating throughput is the total number of operations per second over a varying number of threads. Experiments engage up to 250 threads in three different modes: The first mode uses only physical cores where up to 20 threads are one-to-one mapped to processor cores. The second mode utilizes hyper-threading in addition to physical cores and goes up to 40 threads. The third mode over-subscribes the 40 logical cores with up to 250 threads. The first two modes evaluate the scalability and performance of each list under consideration. The goal of the third mode is to assess system behaviour under high thread preemption. Empirical results are presented, in Figure 1, as graphs in which each point denotes the median of three consecutive runs of five seconds. For each point, there is an error bar that represents ± one standard deviation.

In all-write and balanced workloads, Figures 1a-1c, DULL outperforms all the other lists by several-fold in all of the three modes when considering a key-space of 1000 and 5000. The FliT-based DULL algorithm performed second best in terms of overall performance. FliT, designed as a general-purpose transformation tool, provides persistence by utilizing a custom persistent atomic type in place of the standard atomic library, making it particularly effective for persisting word-sized variables like pointers [55]. However, we found that FliT's generalized approach is not as efficient when persisting complex objects containing multiple

[2] We also experimentally tested the recoverability of DULL. See Appendix G for more details.



(a) Balanced workload – key space of 1000.

(b) Balanced workload – key space of 5000.

(c) All-write workload – key space of 1000.

(d) All-read workload – key space of 1000.

Figure 1 Throughput of state-of-the-art lists under different workloads and key spaces.

variables, such as our descriptors. Specifically, our custom method, which issues a cache-line write-back instruction for all descriptor variables followed by a single store fence, achieves faster performance. This optimization is valid when the object variables are exclusively read and written by a single thread in the absence of failures, with other processes accessing them only during recovery after a crash. While FliT's generalization does not take advantage of such specific optimizations, it offers broader applicability, which limits its ability to maximize performance in our case. The two non-recoverable Harris lists are faster than the other algorithms at high concurrency levels. Note that Harris lists are faster than other lists because, unlike other algorithms, they run on DRAM and are not recoverable. Nevertheless, Harris lists are surpassed by our proposed algorithm.

In the first mode, where threads are one-to-one mapped to physical cores, the DFC-ULL algorithm comes in second place when comparing recoverable solutions in most of the workloads, except for the all-write workload when the key space is 1000 as Romulus-LL has higher throughput. In addition, Romulus-LL surpasses OneFile-LL and PMDK-LL in all scenarios. PMDK-LL is dominated by all algorithms except in all-read workloads, Figure 1d, where it yields higher throughput than OneFile-LL. The reason is, for every node while traversing the list, OneFile-LL performs a more complex load *interposition* that involves two acquire-locks and may search the write-set [48]. On the other hand, DULL, DFC-ULL, Romulus-LL and PMDK-LL employ simpler loads.

In hyper-threading mode, the throughput of DULL increases in high concurrency levels, unlike the other recoverable algorithms in all-write and balanced workloads. This shows the scalability of our proposed algorithm. The performance of DFC-ULL, Romulus-LL and OneFile-LL degrades when more hyper-threaded cores are used in update workloads. The DFC-ULL and Romulus-LL are flat-combining techniques. Therefore, more concurrent updates raise the global combining overhead, which prolongs blocking other threads. The OneFile-LL behaviour is expected as PTMs tend not to scale in scenarios of high degree of simultaneous updates. On the other hand, our proposed algorithm does not require adding locks, or transactions, beyond the ones required for the correctness of the non-recoverable variant. In DULL, we utilize the fact that the unrolled linked list is already lock-based to achieve detectability. Therefore, in high levels of concurrency (>20 threads), DULL is significantly more scalable than the competition.

In over-subscription mode, the performance of DULL, Romulus-LL and OneFile-LL is shown to be stable and saturated as the number of threads increases in all-write and balanced workloads. The DFC-ULL performance, in contrast, drops significantly when the concurrency level rises. It was interesting at first to see the DFC-ULL algorithm collapses in scalability compared to the Romulus-LL, although both algorithms are based on flat-combining. This observation can be explained as a result of the *reduction* step in DFC-ULL. In DFC, the combiner first examines the submitted requests by other threads and attempts to reduce the number of operations by *elimination*. For example, the combiner may resolve the result of each simultaneous pair of opposite operations (an insertion and a removal of the same key) without modifying the data structure. Such a technique is mostly advantageous when used in implementing *abstract data types* (ADTs) that have a parameterless remove procedure, like stacks, queues and deques, where a removal can be matched and eliminated with any concurrent insertion. In contrast, for a correct set ADT implementation, like a linked list, a key removal can only be eliminated by a simultaneous insertion of the same key.

The results considering all-read workloads are demonstrated in Figure 1d. The overhead of adding recoverability while traversing DULL is found to be small when compared to the Vanilla-ULL, thanks to the efficient flush-on-read implemented as part of the $PMptr$ and $PMpair$ persistent objects. The same is noticeable for the DFC-ULL as well. In fact, DFC-ULL is slightly faster than DULL since it does not add flush steps to the original traversing algorithm. The Romulus-LL, OneFile-LL and PMDK-LL are slower compared to the DULL and DFC-ULL in all-read workloads as a consequence of using the unrolled list as the underlying list algorithm.

Figure 1b illustrates the throughput considering a balanced workload and a key space of 5000 to assess and compare the list algorithms for a larger list size, roughly 2500 key-value pairs. When compared to balanced workloads with key space of 1000, Figure 1a, a similar pattern with a consistent lower performance is observable in all algorithms, except for DFC-ULL. That is, the DFC-ULL is impacted the least when increasing the key space and list size. The fact that the degradation in performance of the DFC-ULL algorithm is barely noticeable shows that the DFC component is the limiting factor of the list scalability. Specifically, although a larger list slows down traversal, the flat-combining on updates is relatively more complex and, hence, is the performance bottleneck in the DFC-ULL algorithm. The results for all-write and all-read workloads for key space of 5000 are very similar to the results for key space of 1000. Therefore, such outcomes are omitted due to lack of space.

As Vanilla-ULL demonstrates the fastest performance among all the algorithms due to its unrolled nodes and lack of recoverability mechanisms, we used it as a baseline to evaluate the performance of each algorithm over different list sizes. Figure 2 illustrates the relative throughput of each list compared to Vanilla-ULL using 20 threads, a balanced workload consisting of 50% reads and 50% updates, and varying key spaces.

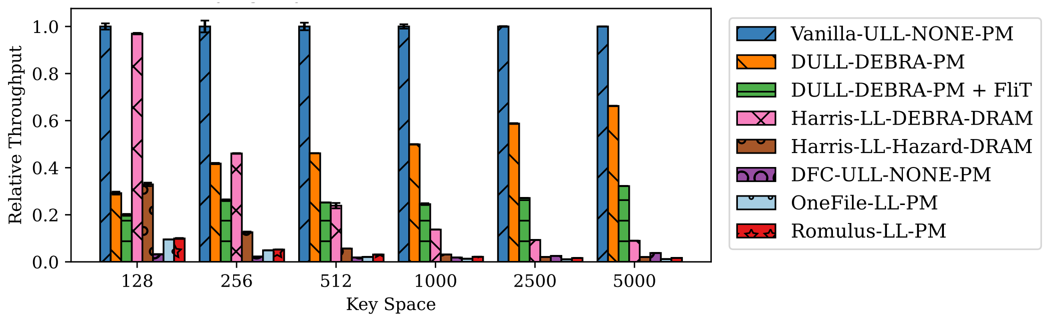


Figure 2 Throughput relative to the Vanilla-ULL – 20 threads – balanced workload.

Since key insertions and removals are uniformly distributed across the key space, the list sizes are approximately half the key space. Based on the experimental results presented in Figure 2, we observe that Harris-LL achieves throughput nearly identical to Vanilla-ULL for smaller list sizes, specifically when the key space is 128, corresponding to approximately 64 list elements. DULL follows, achieving about one-third of the throughput of Vanilla-ULL and Harris-LL, primarily due to the additional overhead introduced by the recoverability and detectability mechanisms in DULL.

As the key space – and consequently, the list size – increases, the relative performance of Harris-LL compared to Vanilla-ULL declines sharply, as does DULL's. This decline is attributable to lock contention: with smaller lists, lock-based algorithms experience higher contention overhead when accessing locks. However, this performance bottleneck diminishes as list sizes grow. Therefore, when the key space exceeds 512, DULL outperforms all other algorithms, demonstrating at least a 2x improvement in throughput compared to Harris-LL and several-fold improvements over the remaining recoverable competitors.

6 Conclusion and Future Work

The emergence of PM allows data structures to preserve their state after a crash failure. Designing detectable and recoverable lock-based data structures is challenging because they must deal with both concurrency and failures. In this work we introduced DULL: a fast, scalable and Detectable Unrolled Lock-based Linked list. Outcomes demonstrate that DULL outperforms all competing recoverable algorithms as it is faster by more than one order of magnitude in all workloads that involve updates. Moreover, our proposed algorithm dominates most of the other algorithms in read-only workloads.

As of future work, we are investigating the challenges and potential advantages of using *recoverable* (RME) locks [26] in the context of detectability. Additionally, we are studying the algorithmic differences between DULL implementations in system-wide and independent crash failure models. Moreover, we are working on a generic transformation based on our detectability technique used in designing DULL to transform lock-based data structures into detectable ones.

References

1 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, 2003.

2 Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, pages 1–16, 2010.

3 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover: Recoverable lock-free data structures. *ArXiv*, 2019. doi:10.48550/arXiv.1905.13600.

4 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 262–277, 2022. doi:10.1145/3503221.3508444.

5 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 7–16, 2018. doi:10.1145/3212734.3212753.

6 Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings VLDB Endow.*, 10(4):409–420, 2016.

7 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 11–20, 2020. doi:10.1145/3382734.3405725.

8 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 253–264, 2019. doi:10.1145/3323165.3323187.

9 Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions, 2022. doi:10.48550/arXiv.2208.11114.

10 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–17, 2016. doi:10.4230/LIPICS.OPODIS.2015.20.

11 Vibhor Bhatt and Prasad Jayanti. Specification and constant rmr algorithm for phase-fair reader-writer lock. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN '11, pages 119–130, 2011. doi:10.1007/978-3-642-17679-1_11.

12 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, 2015. doi:10.1145/2767386.2767436.

13 Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 433–452, 2014.

14 David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized rmr complexity from standard primitives. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*, PODC '20, pages 181–190, 2020. doi:10.1145/3382734.3405736.

15 David Yu Cheng Chan and Philipp Woelfel. Tight lower bound for the rmr complexity of recoverable mutual exclusion. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC '21, pages 533–543, 2021. doi:10.1145/3465084.3467938.

16 CMU Parlay Group. Source Code for FliT: A Library for Simple and Efficient Persistent Algorithms. https://github.com/cmuparlay/flit, 2024.

17 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. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, 2011. doi:10.1145/1950365.1950380.

18 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*, SPAA '18, pages 271–282, 2018. doi:10.1145/3210377.3210392.

19 Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 373–385, 2018. URL: https://www.usenix.org/conference/atc18/presentation/david.

20 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*, PODC '20, pages 1–10, 2020. doi:10.1145/3382734.3405739.

21 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 377–392, 2020. doi:10.1145/3385412.3386031.

22 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40, 2018. doi:10.1145/3178487.3178490.

23 Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1218–1232, 2021. doi:10.1145/3453483.3454105.

24 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 211–220, 2017. doi:10.1145/3087801.3087819.

25 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 17–26, 2018. doi:10.1145/3212734.3212755.

26 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing*, 32(6):535–564, 2019. doi:10.1007/S00446-019-00364-0.

27 R. Guerraoui and R.R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407, 2004.

28 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, 2001. doi:10.1007/3-540-45414-4_21.

29 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, pages 3–16, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

30 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

31 Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, K. Keeton, and Patrick Th. Eugster. Nvthreads: Practical persistence for multi-threaded applications. *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

32 IBM. Overview: Address space layout randomization. `https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization`, 2023. Accessed: 2024-03-22.

33 Intel. eadr: New opportunities for persistent memory applications, 2021. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html`.

34 Ramin Izadpanah, Christina Peterson, Yan Solihin, and Damian Dechev. Petra: Persistent transactional non-blocking linked data structures. *ACM Trans. Archit. Code Optim.*, 18(2), 2021. `doi:10.1145/3446391`.

35 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, 2016. `doi:10.1145/2872362.2872410`.

36 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing*, pages 313–327, 2016. `doi:10.1007/978-3-662-53426-7_23`.

37 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only fasas. In *Proceedings of the 6th International Conference on Networked Systems*, NETYS 2018, pages 191–206. Springer, 2019. `doi:10.1007/978-3-030-05529-5_13`.

38 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 177–186, 2019. `doi:10.1145/3293611.3331634`.

39 Prasad Jayanti and Anup Joshi. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *Proceedings of 31st International Symposium on Distributed Computing*, volume 91 of *DISC*, pages 30:1–30:15, 2017. `doi:10.4230/LIPICS.DISC.2017.30`.

40 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In *Proceedings of the 7th International Conference on Networked Systems*, NETYS 2019, pages 217–232. Springer, 2019. `doi:10.1007/978-3-030-31277-0_14`.

41 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic rmr complexity. In *Proceedings of the 24th International Conference on Principles of Distributed Systems*, volume 184 of *OPODIS 2020*, pages 15:1–15:16, 2021. `doi:10.4230/LIPICS.OPODIS.2020.15`.

42 Nan Li and Wojciech Golab. Brief announcement: Detectable sequential specifications for recoverable shared objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 557–560, 2021. `doi:10.1145/3465084.3467943`.

43 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. `doi:10.1109/TPDS.2004.8`.

44 Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. A closer look at detectable objects for persistent memory. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, ApPLIED '22, pages 56–64, 2022. `doi:10.1145/3524053.3542749`.

45 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 37:1–37:16, 2017. `doi:10.4230/LIPICS.DISC.2017.37`.

46 Kenneth Platz, Neeraj Mittal, and S. Venkatesan. Practical concurrent unrolled linked lists using lazy synchronization. *Journal of Parallel and Distributed Computing*, 139:110–134, 2020. `doi:10.1016/J.JPDC.2019.11.005`.

47 Tiago Pramalhe. Onefile: Concurrent programming simplified. `https://github.com/pramalhe/OneFile`, 2023. Accessed: 2024-03-22.

48 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019. `doi:10.1109/DSN.2019.00028`.

49 Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In Colette Johnen, Elad Michael Schiller, and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 505–509, Cham, 2021. Springer International Publishing. `doi:10.1007/978-3-030-91081-5_38`.

50 Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 185–195, 1994. `doi:10.1145/182409.182453`.

51 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, 2011.

52 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, 2011. `doi:10.1145/1950365.1950379`.

53 Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

54 Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1):58–61, 2015. `doi:10.1109/LCA.2014.2329832`.

55 Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: a library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 309–321, 2022. `doi:10.1145/3503221.3508436`.

56 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings ACM Programming Languages (OOPSLA)*, 3(128), 2019. `doi:10.1145/3360554`.

A More Related Works

Various recoverable concurrent data structures have been developed in the literature [52, 17, 6, 35, 19, 22, 51, 3]. Researchers have established recoverability with different techniques. Volos et al. [52] and Coburn et al. [17] employed redo and undo logging to track threads' progress prior to a crash. A persistent hybrid algorithm was proposed by Brown and Avni [6] that utilizes software and hardware transactional memory in PM. These approaches depend on logging, which comes with a significant performance overhead. An efficient logging technique, JUSTDO, presented by Izraelevitz et al. minimizes the log size via tracking the most recent store instruction only, enabling instant operation recovery after a crash without the need for rolling back [35]. JUSTDO assumes non-volatile caches, which requires additional specific hardware to flush caches on system-wide failures. Other researchers designed log-free recoverable concurrent data structures [19, 22, 51, 3]. David et al. [19] presented set of approaches to construct log-free persistent data structures from their volatile original algorithms. Friedman et al. [22] presented three novel concurrent lock-free queue algorithms with different persistence guarantees. Venkataraman et al. [51] developed a persistent concurrent B-Tree using a log-free versioning scheme that enables instant roll-back. Attiya et al. [3] provided a generic tracking technique aimed at lock-free data structures that achieves recoverability. Dali is a persistent hashmap proposed by Nawab et al. [45] that relies on periodic persistence rather than flushing every update to the PM. The idea is to adopt a weaker correctness property, buffered durable linearizability [36], for better overall performance. Some data structures are said to be detectable [22, 5, 7, 8], which is an additional feature on top of recoverability that enables detecting the response of a crashed operation. In, [5] and [8], Attiya et al. and Ben-David et al. presented a detectable persistent **CAS** synchronization primitive that uniquely identifies each operation. One of the queues proposed by Friedman in [22] offered detectability in the expense of using bookkeeping logs. The lower and upper bound of the space complexity associated with achieving detectability was investigated by Ben-Baruch et al. [7]. They also presented a detectable **CAS** with bounded space complexity.

Achieving detectability and recoverability when designing lock-based data structures is challenging. According to [23], "…lock-free data structures are a natural choice for being adapted to persistence." One of the reasons is the challenges associated pertaining locks' correctness properties (e.g. mutual exclusion and deadlock-freedom) when imposing persistence. This problem was recently formalized by Golab and Ramaraju as the novel *recoverable mutual exclusion* (RME) problem that considers crash failures [26]. Several RME algorithms have been published in the literature [25, 24, 39, 38, 20, 26, 14, 37, 40, 41]. In [24], Golab and Hendler's (GH) RME algorithm was proposed to solve the RME problem in CC systems with a sub-logarithmic *Remote memory reference* (RMR) complexity. RMR complexity is a standard complexity measure for RME algorithms [15]. An RMR is a costly memory operation that uses the interconnect between processor and memory [25]. The RMR complexity of an RME algorithm is the number of RMRs incurred by a single process while acquiring and releasing a lock once [11]. The GH [24] algorithm provides fault tolerance against failures of one process at a time. However, a new transformation of GH was introduced in [25] to accommodate simultaneous (system-wide) failures of processes. Another sub-logarithmic algorithm is Jayanti, Jayanti, and Joshi's (JJJ) [38], which guarantees the same complexity bound as GH in both *distributed shared memory* (DSM) and CC systems. Dhoked and Mittal [20] proposed an adaptive approach that improves performance, compared to previous works, in case of frequent failures. The idea is to use a (weaker) variant of the RME problem that violates the ME property, allowing multiple threads to be in their critical section simultaneously but in a controlled manner, and use that variant as a building block. A recent publication by Chan and Woelfel [15] proved that a deadlock-free n-process RME algorithm has a tight bound of sub-logarithmic RMR complexity, showing that GH [24] and JJJ [38] are optimal.

B Persistent Pointer

The persistent pointer object is defined in Algorithm 6. A persistent pointer consists of the variable *offset*, which is a tuple $\langle value, flag\rangle$ where *value* is an address offset from the PM base address, and *flag* is 1 if *offset* is persisted and 0 otherwise. Operating systems randomize the address space layout upon loading applications into the memory. This is known as *address space layout randomization* (ASLR). After recovering from a system failure, dereferencing a pointer on PM that stores the address of a persistent object might result in accessing an invalid memory location. The PMptrcompensates for ASLR by storing each pointer as an offset from the base address of the persistent memory-mapped file and calculating the pointer value on every access. BaseAddressis initialized every time the program is (re)started to ensure valid referencing when using PMptr pointers.

Algorithm 6 also handles persisting PMptr pointers. In some scenarios, values read from or written to persistent variables must be flushed before being used. Consider the following example: Assume there exists a concurrent singly-linked list that realizes an unordered set, and that the set only contains the value 5 at the head node. Consider two concurrent

Algorithm 6 Variables and functions of a *PMptr* object.

Variables:

- offset: **tuple** $\langle \textbf{pointer}, \textbf{Boolean} \rangle$, **init** $\langle 0, 0 \rangle$

Procedure SaP (ptr)

```
138 $value := ptr - \text{BaseAddress}$                        // Get the offset
    // Mark not persisted and store offset atomically
139 $offset := \langle value, 1 \rangle$
140 Flush($offset$)                                            // Persist to PM
141 CAS($offset, \langle value, 1 \rangle, \langle value, 0 \rangle$)   // Mark persisted
```

Procedure GaP

```
142 $\langle value, flag \rangle := offset$                   // Read the offset atomically
143 if $flag = 1$ then                                         // Check if offset is not persisted
144     Flush($offset$)                                        // Persist to PM
145     CAS($offset, \langle value, 1 \rangle, \langle value, 0 \rangle$)   // Mark persisted
146 return $\text{BaseAddress} + value$                        // Return the address
```

operations **Remove** (5) and **Contains** (5). Suppose that the former changed the head pointer of the list to $\perp$, then the latter dereferenced the head pointer and returned false, indicating that 5 is removed from the list. Now, if the system crashes before **Remove** (5) persists the value ($\perp$) of the head pointer, then, after recovery, a subsequent **Contains** (5) would return True which contradicts that 5 has been removed from the set. A possible solution is to persist the head pointer after reading it in **Contains** (5) before returning.

Persisting variables upon each read and write is a conservative approach to ensure correctness. However, this might be a performance overkill since each update needs to be flushed once to the PM for a subsequent recovery. Using PMptr enables persisting the offset more efficiently. The idea is borrowed from the work in [53]. The *offset* variable is a tuple of the offset value intended to be persisted and a "dirty bit" that indicates whether the current offset value was never persisted (1) or it was flushed at least once (0). As shown in Algorithm 6, a PMptr object can be accessed through two operations: *Set and Persist* (**SaP**) and *Get and Persist* (**GaP**).

The function **SaP** takes a pointer to a persistent object (ptr) and calculates the offset value ($value$) at Line 138. Then, *offset* is set to $\langle value, 1 \rangle$, setting its value to the calculated offset and marking the offset as not persisted. The **SaP** operation flushes *offset* (Line 140) and then attempts to mark it as persisted by resetting the dirty bit to 0 through a **CAS** operation (Line 141). The **CAS** at Line 141 is required to avoid erroneously overwriting a value set by another concurrent **SaP** operation.

The **GaP** operation reads the tuple *offset* (Line 142) and checks whether it was flushed or not by examining the dirty bit (Line 143). If the dirty bit is set to 1, then *offset* is persisted at Line 144, and the dirty bit is reset to 0 through a **CAS** operation at Line 145. Similar to **SaP**, the **CAS** at Line 145 is needed to prevent potential incorrect changes to updates occurred by a concurrent **SaP** operation.

C Persistent Pair

The persistent key-value pair object is defined in Algorithm 7. A persistent key-value pair object comprises a single variable and three associated functions. The variable *pair* is a tuple $\langle key, value, flag \rangle$, where *key* and *value* are variables that store a key and its corresponding data, respectively, and *flag* is a dirty bit set to 1 if *pair* is persisted and 0 otherwise.

Algorithm 7 Variables and functions of a *PMpair* object.

Variables:

- *pair*: **tuple** $\langle key, value, \textbf{Boolean}\rangle$, **init** $\langle\bot, \bot, 0\rangle$.

Procedure Set ($\langle key, value\rangle$)

```
    // Store the key-value pair and mark not persisted
147 $pair := \langle key, value, 1\rangle$
```

Procedure Get

```
148 $\langle key, value, \_\rangle := pair$                 // Read key-value pair
149 return $\langle key, value\rangle$
```

Procedure Persist

```
150 $\langle key, value, flag\rangle := pair$             // Read the pair atomically
151 if $flag = 1$ then                                     // Check if the pair is not persisted
152     Flush ($pair$)                                     // Persist to PM
153     CAS ($pair$, $\langle key, value, 1\rangle$ , $\langle key, value, 0\rangle$)   // Mark
```

Accessing *pair* is governed by three functions: **Set**, **Get** and **Persist**. The function **Set** stores a key-value pair, given as an argument, into *pair* and marks the tuple as not persisted. The key and value stored in *pair* are retrieved by the procedure **Get**, ignoring the dirty bit state. Both functions do not flush the key-value pair object. Persisting *pair* is delegated to a separate procedure, **Persist**, for performance purposes. This function reads the tuple *pair* (Line 150) and checks the dirty bit (Line 151). If the dirty bit is set to 1, then *pair* is not persisted, and the function flushes *pair* to the PM at Line 152. Finally, *pair* is marked as persisted by resetting the dirty bit to 0 through a **CAS** at Line 153. The **CAS** primitive at Line 153 is used for the same reason as at Lines 141 and 145.

D Memory Layout

The memory mapped file is organized as follows: The first 8 bytes are reserved for NIL pointer. Subsequently, N descriptor objects – one per process – are allocated. The next 2x node-size bytes are used for the head and tail nodes. The rest of the memory-mapped file is utilized as a pool for node allocation. The memory layout is illustrated in Figure 3. We implement and use a simple recoverable bitmap fixed-size memory allocator.

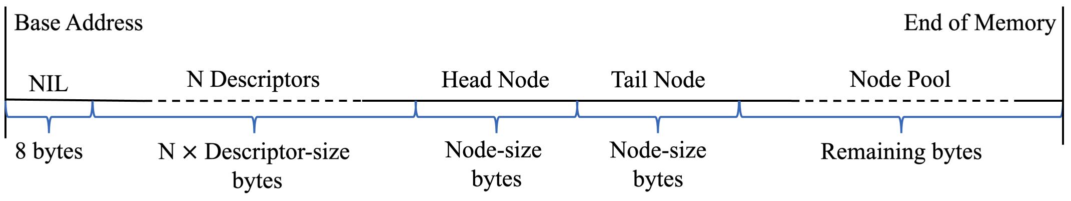


Figure 3 Memory layout for DULL.

E Split

Algorithm 8 defines the subroutine **Split**. This function creates two new nodes to replace *curr*, where each stores roughly half of the pairs in *curr*. The procedure starts by creating a local copy, *tmp*, of the *pairs* array in *curr* at Line 154 and sorting it by key at Line 155. New nodes, new_1 and new_2, are allocated (Lines 156 and 157), and the smallest share ($\lfloor K/2\rfloor$) of

Algorithm 8 Helper function **Split**.

Procedure Split (*curr*)

```
154 $tmp := curr.pairs$                                  // Get local copy of the array pairs
155 Sort($tmp$)                                          // Sort pairs by key
156 $new_1$ := new Node                                  // Create two new nodes
157 $new_2$ := new Node
    // Copy half pairs to new1 and the rest to new2
158 for $j := 0$ to $K - 1$ do
159     $\langle k, v\rangle := tmp[j]$.Get()
160     if $j \leq \lfloor K/2 \rfloor$ then
161         $new_1.pairs[j]$.Set($k, v$)
162     else
163         $new_2.pairs[j]$.Set($k, v$)
164 $new_1.anchor := curr.anchor$                        // Set other node variables
165 $new_1.count := \lceil curr.count/2 \rceil$
166 $\langle k_{\min}, \_\rangle := new_2.pairs[0]$.Get()
167 $new_2.anchor := k_{\min}$
168 $new_2.count := \lfloor curr.count/2 \rfloor$
169 $new_1.next$.SaP($new_2$)                            // Link and persist the next pointers
170 $new_2.next$.SaP($curr.next$.GaP())
171 return ($new_1, new_2$)
```

the pairs in *tmp* are copied to new_1, and the rest of *tmp* is copied to new_2 (Lines 158-163). The anchor key of new_1 is set to the anchor key of *curr* at Line 164. The anchor key of new_2 is set to the key of the first pair (Line 167), which has the smallest key in new_2 since the node's array is sorted by key. The counter of each new node is set to the corresponding number of copied pairs from *curr* at Lines 165 and 168. Finally, the next pointers of new_1 and $new_2$are set to new_2 and the successor of *curr* at Lines 169 and 170, respectively, before returning to **Insert**.

F Merge and Redistribute

Algorithms 9 and 10 show the steps for the helper subroutines **Merge** and **Redist**, respectively.

Algorithm 9 Helper function **Merge**.

Procedure Merge (*curr*, *succ*)

```
172 $new_1$ := new Node                                  // Create a new node
173 $j := 0$                                             // Copy valid pairs from curr and succ to new_1
174 for $i := 0$ to $K - 1$ do
175     $\langle k_{curr}, v_{curr}\rangle := curr.pairs[i]$.Get()
176     $\langle k_{succ}, v_{succ}\rangle := succ.pairs[i]$.Get()
177     if $k_{curr} \neq \bot$ then
178         $new_1.pairs[j]$.Set($k_{curr}, v_{curr}$)
179         $j := j + 1$
180     if $k_{succ} \neq \bot$ then
181         $new_1.pairs[j]$.Set($k_{succ}, v_{succ}$)
182         $j := j + 1$
183 $new_1.anchor := curr.anchor$                        // Set other node variables
184 $new_1.count := curr.count + succ.count$
185 $new_1.next$.SaP($succ$.GaP())
186 return $new_1$
```

Merging the current node and its successor involves creating a new node at Line 172 and copying all valid pairs from the current and successor nodes to the newly created node at Lines 174-182. The merged node's anchor key is set to the current node's anchor key (Line 183) as it is less than or equal to all keys in the current and successor nodes. The counter of the new node is the sum of counters of the two nodes to be replaced (Line 184), the current

Algorithm 10 Helper function **Redist**.

Procedure Redist $(curr,\ succ)$

```
187 $new_1 :=$ new Node                                            // Create two new nodes
188 $new_2 :=$ new Node
189 for $i := 0$ to $K - 1$ do                                      // Copy valid pairs from curr to new_1
190     $\langle k, v\rangle := curr.pairs[i]$.Get()
191     if $k \neq \bot$ then
192         $new_1.pairs[new_1.count]$.Set$(k, v)$
193         $new_1.count := new_1.count + 1$
    // Number of pairs to move from succ to curr
194 $M := \left\lceil \frac{curr.count+succ.count}{2} \right\rceil - curr.count$  $tmp := succ.pairs$   // Get local copy of the array pairs
195 Sort $(tmp)$                                                    // Sort pairs by key
196 for $i := 0$ to $M - 1$ do
        // Copy the first M pairs in tmp to new_1 and the rest to new_2
197     $\langle k, v\rangle := tmp[i]$.Get()
198     $new_1.pairs[curr.count + i]$.Set$(k, v)$
199 for $i := M$ to $succ.count - 1$ do
200     $\langle k, v\rangle := tmp[i]$.Get()
201     $new_2.pairs[i - M]$.Set$(k, v)$
202 $new_1.anchor := curr.anchor$                                   // Set other node variables
203 $new_1.count := curr.count + M$
204 $\langle k_{\min}, \_\rangle := new_2.pairs[0]$.Get()
205 $new_2.anchor := k_{\min}$
206 $new_2.count := succ.count - M$
207 $new_1.next$.SaP$(new_2)$                                       // Link and persist the next pointers
208 $new_2.next$.SaP$(curr$.GaP$())$
209 return $(new_1, new_2)$
```

and successor nodes. The new node's next pointer is set to the next pointer of the successor node at Line 185 to replace both merged nodes when the predecessor's next pointer is set to the new node at Line 96 of **Remove**.

If merging the current and the successor nodes is not possible (Line 91), the pairs stored in these two nodes are roughly evenly redistributed across them by calling the helper function **Redist**, shown in Algorithm 10. The helper subroutine replaces a current node and its successor with two new nodes, new_1 and new_2, respectively. The current node's pair count is $<$ MinFull (Line 83), the total pair count of the current and successor nodes is $\geq$ MaxMerge (Line 91) $\geq (2 \times \text{MinFull}) - 1 > 2\times$ the current node's count. This implies that the current node pair count is $<$ the successor's. To evenly redistribute pairs among the two nodes, each node should store $\frac{curr.count+succ.count}{2}$ pairs. Since the current node has fewer pairs, $M := \lceil \frac{curr.count+succ.count}{2} \rceil - curr.count$ pairs should be moved from the successor to the current node (Line 194). The procedure **Redist** copies all pairs stored in the current node to new_1 in Lines 189-193. Subsequently, the function copies the smallest M pairs from the successor node to new_1 and the rest of pairs to new_2 at Lines 194-201. The anchor key of new_1 is set to the current node's anchor key (Line 202) as it is less than or equal to all keys in the current and successor nodes. The keys in new_2 are sorted since they are copied in order from the sorted array tmp. Hence, the anchor key of new_2 is set to the key of its first pair, which is the smallest key in new_2, at Line 205. The counter of new_1 is set to the pair count of the current node plus the number of pairs copied from the successor (M) at Line 203, and the counter of new_2 is set to the number of the remaining pairs $(succ.count - M)$ at Line 206. Finally, the function **Redist** links the new nodes together by setting the new_1's next pointer to new_2 (Line 207), and connects the two nodes to the list by setting the new_2's next pointer to the next node of the successor (Line 208). The next pointers of the two new replacing nodes are persisted by the **SaP** function, before they are returned to and used in the **Remove** procedure.

G Recoverability Testing

To test the recoverability of the proposed linked list, we have instrumented the code base such that each thread logs the invocation and response for each of the following operations: **Contains**, **Insert** and **Remove**. The result is a sub-history file per thread of sequential operations, which later are combined into one history file of interleaving operations. We implemented an extended variant of the algorithm proposed by Anderson et. al. [2] to examine the produced execution history for strict linearizability violations. When creating history files to test recoverability, the benchmark ensures that the assumptions in [2] are satisfied, including:

- All procedures are reduced to read and write operations.
  - **Contains** (key): a failed search is recorded as a read of the pair $(key, \perp)$[3] while a successful search is logged as a read of $(key, value)$, for some $value \neq \perp$.
  - **Insert** $(key, value)$: a failed insertion is a read of $(key, value)$, for some $value \neq \perp$, while a successful insertion is logged as a read of $(key, \perp)$ followed by a write of $(key, value)$.
  - **Remove** $(key, value)$: a failed removal is a read of $(key, \perp)$ while a successful insertion is logged as a read of $(key, value)$, for some $value \neq \perp$, followed by a write of $(key, \perp)$.
- Each key-value pair is unique.
  - **Insertion**: Insertions of all pairs with the same key are guaranteed to have unique values. Thus, a key can be reinserted but with different values.
  - **Removal**: Absence of a key is represented using distinct values. A key that was never inserted has the value $value = \text{INIT}$. A key's value is assigned to $\perp$ at most once, when it is firstly removed. Any subsequent successful removal of the same key only marks the value as deleted ($value$ to $value'$) without actually removing the key from the list. Marking a value is implemented as follows: Bit-wise shift all values to the left upon logging. Use the least significant bit as the marker (1 for deleted and 0 otherwise).

The linearizability checker in [2] is based on the original definition of linearizability [30], where failure-free executions are assumed. Therefore, we extended the code to accommodate strict linearizability. When the application is restarted after a crash, the last entry of each history file is examined for an incomplete operation *op*. If *op* is a read, then *op* is removed from the log file. If *op* is a write, then the PM content is examined to check whether the written value exists in memory. If the value is found in the PM, then the response of the write operation is appended. Otherwise, the write operation is removed from the log file. Removing an incomplete write operation $w(k, v)$ or appending its response based on the existence of the pair (k, v) in PM comes from the following intuition: If (k, v) is in PM, then a write operation $w'(k, v)$ must have written and persisted the pair (k, v). According to Algorithms 3 and 4, the thread that has started $w(k, v)$ must have locked the predecessor (Lines 41 and 73) of the node containing (k, v) and then found that k is not in PM (Lines 45 and 77). Therefore, $w(k, v)$ must be the operation that has written (k, v), i.e. $w'(k, v)$. Otherwise, $w(k, v)$ and $w'(k, v)$ would have simultaneously acquired the predecessor's lock, which contradicts the safety property of the mutex locks. The linearizability check is implemented in python and used for assessment.

We tested the algorithm initially by crashing the program (via `kill -9`), and then by repeatedly power-cycling the server to evaluate recoverability under system-wide failure.[4] The resultant history files were fed into the implemented linearizability checker, and no violations were reported.

[3] A unique value outside the key space is used to represent $\perp$.
[4] A test script was used to crash the server repeatedly for days.