ResPCT: Fast Checkpointing in Non-volatile Memory for Multi-threaded Applications

Ana Khorguani
ana.khorguani@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Thomas Ropars
thomas.ropars@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Noel De Palma
noel.depalma@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Abstract

Non-volatile memory (NVMM) technologies are a great opportunity to build fast fault-tolerant programs, as they provide persistent storage in main memory. However, since the processor caches remain volatile, solutions are needed to recover a consistent state from NVMM after a crash. This paper presents ResPCT, a checkpointing approach to make multi-threaded programs fault tolerant, by flushing persistent data structures to NVMM periodically. ResPCT uses In-Cache-Line logging to efficiently track modifications during failure-free execution, and to restore a consistent state after a crash. The ResPCT API enables programmers to position restart points in their program, which simplifies the identification of the persistent program state and can also help improving performance. Experiments with representative benchmarks and applications, show that ResPCT can outperform state-of-the-art solutions by up to 2.7×, and that its overhead can be as low as 4% at large core count.

Keywords: non-volatile memory, fault tolerance, checkpointing, multi-threaded applications

1 Introduction

Emerging Non-Volatile Main Memory (NVMM) modules [25] can preserve data across system reboots or crashes, while achieving performance close to DRAM. NVMM modules are byte-addressable. They can be plugged directly on the memory bus, to be used as main memory and be accessed directly through load and store instructions.

NVMM is an opportunity to build fast resilient programs. Using it as main memory allows recovering the state of programs after a crash, without having to flush data to a *slow* persistent I/O device (SSD or HDD) during failure-free execution. However, the following problems need to be solved.

First, even if NVMM is used as main memory, several levels of volatile caches are present between the processor and the memory to improve memory access performance. Data written by programs are first stored in cache before being written back to NVMM on cache line eviction. Consequently, if no specific action is taken, parts of the program state might still be lost in a crash if some data were only stored in the volatile caches and not written yet to NVMM.

Second, the state stored in NVMM might be inconsistent because data might be written back from cache to NVMM in an order that differs from program order. To ensure that an application can restart from a consistent state after a crash, solutions need to be designed to guarantee either that the state of data structures in NVMM is always consistent [30, 47] or that enough information is available on restart to recover a consistent state [9, 33].

Controlling data movement between the cache and the main memory requires using cache line flush and memory fence instructions [42], but using these instructions has a performance cost as it forces the processor to synchronize with main memory and imposes ordering on some memory accesses. This cost is amplified by the fact that although NVMM provides high throughput and low latency, its performance is not on par with DRAM [49].

Transaction-based solutions [8, 13, 14, 20, 32, 37, 40, 44, 48] or other comparable approaches [6, 23, 30, 33] allow building fault tolerant programs on top of NVMM and provide *durable linearizability* [27]. In this consistency model, once an operation has been fully executed, it is never rolled back. It implies that all updates executed during a transaction have to reach NVMM before the transaction can commit.

Checkpointing is an attractive alternative that allows smoothing the cost of synchronization with NVMM by adapting the duration of the checkpointing period [2, 9, 36, 45, 47]. Checkpointing approaches divide the execution of the program into epochs and only flush modified data structures to NVMM at the end of each epoch. The achieved consistency model is *buffered durable linearizability* [27]: all operations executed after the last checkpoint are lost in a crash. Such a consistency criterion is sufficient for many use cases [31], especially when epochs can be as short as a few milliseconds.

Efficient checkpointing faces two main challenges: (i) A *consistency* challenge, *i.e.*, recovering the program in a consistent state after a crash, despite partial modifications that

EuroSys '22, April 5–8, 2022, RENNES, France
The definitive Version of Record was published in *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France*, https://doi.org/10.1145/3492321.3519590.

might have reached NVMM during the crashed epoch; (ii) A *tracking* challenge, *i.e.*, identifying the memory locations that have been modified, and thus, that must be flushed to NVMM at checkpoint time [47].

This paper presents ResPCT, a solution based on checkpointing to build fault-tolerant multi-threaded programs on top of NVMM. ResPCT relies on In-Cache-Line Logging (InCLL), proposed by related work [9, 40]. InCLL implements highly efficient undo logging, by storing the log of a datum in the same cache line as the datum itself. This leverages the persistency semantics of the Intel-x86 architecture [39] to guarantee that an undo log will reach NVMM before the corresponding modification, without using any flush or fence instructions. In previous works, InCLL was used to solve the *consistency* challenge in a specific data structure [9] or in a transactional system [40]. This paper uses InCLL to implement a general-purpose checkpointing solution for programs running on top of NVMM. We describe how InCLL can be used not only to efficiently address the *consistency* challenge introduced above, but also the *tracking* challenge.

Our other innovation identifies the program states in which checkpoints can be taken as *Restart Points* (RPs). The programmer positions RPs in the application code. All threads need to reach an RP before a checkpoint can be taken. This approach has several advantages. First, we argue that it simplifies the work of the programmer in the definition of the recovery procedure. Indeed, providing an explicit API to locate RPs allows us to define simple rules to identify data structures that need to be stored in NVMM and restored during recovery. Second, choosing the position of RPs can help reducing the size of the state saved in NVMM, and thus improve performance as some of our experiments show.

We have implemented a prototype of ResPCT and tested it with basic concurrent lock-based data structures (queue, hash table), benchmarks from multi-threaded benchmark suites (Parsec [4], Phoenix [41]), and a popular in-memory key-value store (Memcached [34]), on real NVMM hardware with a 32-core processor. When checkpointing every 64 ms, our results show that the overhead of ResPCT is below 20% in most cases, and can even be as low as 4%, compared to the non-fault-tolerant version of the program executed on DRAM. Its speedup compared to the best existing solutions is up to 2.7×.

To summarize, this paper presents a new approach to checkpoint multi-threaded applications to NVMM, and makes the following contributions:

- A solution based on InCLL to solve both the consistency and the tracking challenges associated with checkpointing in NVMM.
- An API for locating restart points in programs with benefits in terms of performance and simplicity.
- An extended evaluation using both micro-benchmarks and real applications, that shows that ResPCT can significantly outperform all existing fault tolerant techniques targeting NVMM, including other checkpointing techniques.

The rest of the paper is organized as follows. Section 2 presents the system model considered in this paper and describes the related work. Section 3 describes our algorithm. Section 4 provides a correctness proof. Finally, Section 5 presents our experimental evaluation.

2 Background

2.1 System model and assumptions

Our system model, illustrated in Figure 1, captures the main characteristics of existing NVMM hardware when used with multicore x86 Intel processors [25]. It corresponds to the *App Direct* mode with Intel DCPMM memory modules [25]. NVMM DIMMs are plugged on the memory bus, are byte-addressable, and are directly accessible through load and store instructions. The system might also include DRAM side-by-side with NVMM. A program can explicitly choose to store data in NVMM or DRAM.

Our goal is to ensure that a program can restart from a consistent state after a crash, and to minimize the computation lost due to the crash. We target race-free multi-threaded programs where synchronization between threads is managed by mutex locks or condition variables. The synchronization mechanisms induce a partial ordering, *i.e.*, *happens-before* relations, between the load and store instructions executed by the different threads, as defined in the C++ memory model [5]. A consistent state can be defined as one that complies with this partial ordering.

Since we assume race-free multi-threaded programs, the algorithm presented in Section 3 relies on the fact that concurrent writes to the same variable cannot occur. Furthermore, it does not support programs that use atomic read-modify-write instructions to modify shared variables. Its design is based on the assumption that any thread modifying a shared variable holds the lock that gives exclusive access to this variable.[1] We name *critical section*, a section of code protected by a mutex lock. Our algorithm is not designed to support lock-free and wait-free algorithms.

Note that our design assumes that memory management, as well as all accesses to shared data, are under the control of the programmer, which implies that programs that rely on a garbage collector are not supported.

CPU registers and caches are volatile. In contrast, we assume that the memory controllers belong to the persistent

[1] This assumption allows us to have a very efficient solution. ResPCT could support atomic operations but that would imply introducing locks in the implementation, and degrading performance significantly. We prefer requiring the programmers to replace atomic operations by mutex locks in their program rather than reducing the performance of ResPCT.

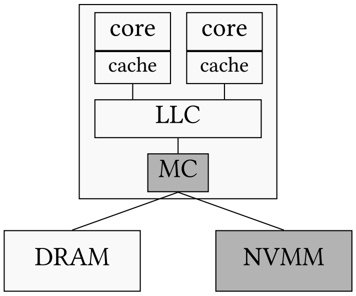

Figure 1. Multicore processor with NVMM side-by-side with DRAM. The persistent domain is in gray. The memory controller (MC) belongs to the persistent domain while the private cache and the shared last-level cache (LLC) are volatile.

domain, as with Intel *App Direct* mode [50]. A datum is persistent once it has been flushed from cache. The ordering of write operations in NVMM follows the Persistent Cache Store Order model (PCSO) [10], as implemented by x86 architectures. In PCSO, a cache line is written back according to an (unknown) cache line replacement policy implemented in hardware. Hence, after a store instruction, a datum might reside in cache for some time and be lost in the event of a crash. Furthermore, writes might reach NVMM in an order not consistent with program order.

Some guarantees can be obtained on the order in which updates reach NVMM. First, our model assumes that data movement can be controlled using dedicated instructions. A pwb instruction initiates write-backs at cache-line granularity. This instruction is asynchronous. A psync instruction should be issued to ensure that all preceding pwb instructions issued by the current thread are completed. On modern x86 architectures, pwb is implemented using the *clwb* instruction and psync using *sfence* [26]. Second, in the case of multiple writes to the same cache line, PCSO guarantees that a write to a cache line never reaches NVMM before any preceding writes issued by any thread to the same cache line [10]. In this case, no dedicated instruction needs to be used.

2.2 Related work

NVMM modules, such as Intel Optane DCPMM modules [25], are available since a few years. They are persistent but much faster than traditional storage devices such as Solid-State Drives (SSDs) [49]. However, fault-tolerant algorithms must be designed with care to achieve good performance because, although NVMM can be used as a replacement of DRAM, it is not as fast as DRAM. Namely, read latency is between 2x and 3x higher than with DRAM. The peak read and write bandwidth is also significantly lower [49].

Several fault-tolerant approaches built on top of NVMM guarantee *durable linearizability* [27]. Durable linearizability ensures that each operation persists in a single atomic step between its call and its return. It implies that the state available after a crash must include all operations that completed before the crash. To this end, each operation must include at least one call to *clwb* and one call to *sfence* to ensure that modified data have reached NVMM before continuing [11, 40]. General solutions have been proposed to provide durable linearizability for concurrent programs, either through persistent transactional memory [8, 13, 14, 20, 32, 37, 40, 44, 48] or through failure-atomic sections for critical sections [6, 19, 23, 30, 33, 47]. These approaches add *clwb* and *sfence* instructions on the critical path of programs, and thus, can induce a significant performance overhead.

An alternative is to provide *buffered durable linearizability* [27]. It ensures that the state recovered after a crash includes a consistent subset of the operations that completed before the crash. Systems implementing this consistency model are typically based on checkpoints [9, 19, 29, 36, 45, 47]: they divide the program execution into epochs and flush all updates to NVMM at the end of each epoch. Flushing updates periodically amortizes the cost of synchronizing with NVMM. Hence, such approaches aim at providing a different trade-off between performance and consistency.

Because the hardware might write a modification back from cache to memory *at any moment*, all the solutions listed above, including the systems implementing durable linearizability, need to support rollbacks of *partial updates* after a crash. The two main approaches are undo logging [8] or redo logging [18, 44]. Both have performance drawbacks [29, 50]: Undo logging requires extra synchronization with NVMM to write the log; Redo logging imposes read redirection. Some techniques have been proposed by durably linearizable systems to improve performance. Pronto [33] implements operation logging, whereas Clobber-NVM [48] significantly reduces the size of the undo log, by logging only the variables that are both in the read set and in the write set of a transaction. We include a comparison with Clobber-NVM in our experimental evaluation. The results show that ResPCT can be significantly faster.[2]

To avoid the overhead of undo logging with checkpointing solutions [19], alternative approaches targeting specific data structures have been suggested [9, 36]. Cohen *et al.* [9] introduce In-Cache-Line Logging as an extremely efficient undo log, for a Masstree data structure. InCLL locates the undo log of a datum in the same cache line as the datum itself. It takes advantage of the PCSO semantics for writes to the same cache line, to log without using *clwb* or *sfence* instructions. Using InCLL, Cohen et al. achieve unprecedented performance for this specific data structure. But the algorithm they propose is tailored to Masstree. To the best of our knowledge, ResPCT is the first general-purpose checkpointing algorithm

[2] We also evaluated Pronto, but because its performance is not as good as Clobber-NVM, it is not included in this paper.

using InCLL that ensures buffered durable linearizability for an arbitrary lock-based concurrent program.

InCLL has also been studied in the implementation of a persistent transactional memory [40]. The two proposed algorithms, Trinity and Quadra, achieve very high performance and guarantee durable linearizability. ResPCT is designed for a different trade-off: higher performance but with buffered durable linearizability.

Since they are also general approaches that implement checkpointing, PMThreads [47] and Montage [45] are the two systems the most similar to ResPCT. To avoid the cost of logging, PMThreads maintains a shadow copy of the persistent data structures in DRAM. During an epoch, updates are written only to DRAM. All modifications are copied to NVMM at the end of the epoch. Working with data in DRAM can be beneficial since DRAM is faster. However, PMThreads has to track the modifications made during an epoch, to be able to copy them to NVMM when the epoch ends. PMThreads implements two solutions for this tracking [47]: (i) intercepting store instructions, and (ii) relying on the OS page protection mechanism. Both have a non-negligible overhead. ResPCT also needs to track modifications, but it uses a strategy that takes advantage of InCLL to implement very efficient tracking.

To avoid the cost of logging, Montage [45] uses Copy-on-Write (CoW): any update in NVMM requires allocating a new object. This strategy also eases tracking: Montage provides a dedicated API for memory allocation in NVMM, and modification tracking is implemented as part of the allocation function. To avoid a cascade of updates for data structures based on pointers, pointers are only maintained in DRAM but sufficient metadata are stored in NVMM to reconstruct indexes after a crash.[3] As our evaluation shows, the main difference between Montage and ResPCT is that ResPCT puts less stress on the memory allocator, and does not introduce extra metadata.

In checkpointing approaches, another important design choice is to decide when checkpoints can be taken. To ensure correctness, checkpoints should be allowed only in states where happens-before relations can be maintained even after a crash. The simplest way to ensure this, is to guarantee that checkpoints cannot occur as long as a thread is in a critical section [19]. Beyond this constraint, existing approaches do not define any specific rules regarding the states in which checkpoints can be taken. For instance, with PMThreads, the state of a thread can be checkpointed at the end of any critical section [47].

ResPCT opts for a different strategy. It introduces an API that allows programmers to explicitly position *Restart Points* (RPs) in their code to identify states where checkpoints can occur. Such an approach is inspired from solutions that implement application-level checkpointing in parallel distributed systems [3]. As discussed in more details in Section 3.3, we think that positioning RPs in the program code is a simple task that ultimately simplifies the job of a programmer in making her program fault tolerant. As demonstrated by our evaluation, it also helps improving performance by reducing the size of the persistent state.

3 Algorithm

This section starts by providing an overview of our algorithm in Section 3.1. Section 3.2 details the mechanisms used to ensure consistency and Section 3.3 presents Restart Points.

3.1 Design overview

ResPCT is a solution based on checkpointing to make multi-threaded programs fault tolerant. It divides the execution of the program into epochs. During an epoch, the algorithm does not constraint the order in which updates reach memory. For instance, assume the insertion of an element in a list. ResPCT allows the pointers of the list to be updated in NVMM (due to cache line eviction) without the new element being persisted yet. ResPCT does not execute any cache line flush or memory fence instructions during normal execution to order writes to NVMM.

At the end of each epoch, ResPCT executes a checkpoint procedure that flushes the updates made during the epoch to NVMM. To ensure that this state is consistent, threads may not progress while the checkpoint is being taken.

ResPCT has to solve a *consistency* and a *tracking* challenge. The consistency challenge is to be able to roll back any partial updates that have reached NVMM during an epoch that crashes. To solve this issue, ResPCT adopts In-Cache-Line Logging [9]. It simply locates the undo log for a variable in the same cache line as the variable itself, which guarantees that if a variable update reaches NVMM, the corresponding undo log is written to NVMM as well. Hence, ResPCT does not execute any flush or fence instruction to implement logging. Such an approach is key to achieve high failure-free performance on NVMM.

The tracking challenge is to identify the memory locations that have been modified during the current epoch, to be able to flush all of them at checkpoint time. To do this efficiently, ResPCT takes advantage of InCLL. Since InCLL stores the epoch in which a variable has last been modified next to the variable, ResPCT uses this information to implement efficient tracking.

Two more inter-dependent points need to be addressed to make a program fault tolerant using checkpointing: (i) identifying program states in which a checkpoint may be taken; (ii) identifying the subpart of the program state that needs to be made persistent to be able to restart after a crash.

[3] For instance, to implement a FIFO queue, all items are identified by a global sequence number stored in NVMM. This sequence number is used during recovery to figure out the order of the elements in the queue.

To address these issues, ResPCT provides an API that enables programmers to explicitly position Restart Points in the program. At the end of the epoch, ResPCT waits until each thread reaches an RP before actually taking the checkpoint and moving to the next epoch. Requiring the programmer to position RPs might sound difficult or inconvenient. But our experiments show that it is a simple task.

Positioning RPs manually has several advantages that we detail in Section 3.3. The position of RPs in the code dictates which data belong to the persistent state, and which data require logging. Hence, by choosing where to place RPs, the programmer can reduce the size of the persistent state and/or the need for logging, and improve performance (see the detailed evaluation in Section 5.3).

We detail the ResPCT algorithm in the following. We start by describing how it ensures that a consistent state is recovered after a crash. Then, we present the concept of Restart Point, its implementation and its use in practice. The main data structures and variables used by the ResPCT algorithm are presented in Figure 2 and Figure 3, the algorithm itself is presented in Figure 4 and Figure 5. We summarize the API of ResPCT is Table 1, and describe it in more details as we present the algorithm.

3.2 Ensuring consistency

Our approach relies on periodic checkpoints that divide the program execution into successive epochs. A checkpoint flushes all the memory locations that belong to the persistent state and that have been modified during the current epoch by calling `flush_modified()` (Lines 14-18 in Figure 4). The checkpoint procedure increases the global variable `epoch` (the current epoch number) and flushes it by executing the `clwb` instruction followed by a fence to persist it in NVMM (Lines 56-58). Thus, if a failure occurs, a recovery procedure can easily identify the epoch that crashes.

As detailed in Section 3.3, ResPCT ensures that all program threads are blocked at restart points when the checkpoint is taken. This ensures that the state that is flushed is consistent. It also guarantees that the epoch number can be incremented safely in the checkpoint procedure.

During an epoch, as the program modifies data structures, it is possible that only partial updates reach NVMM. Updates that are only in the cache will be lost in the event of a failure. To be able to reconstruct a consistent state during recovery, ResPCT relies on undo logging implemented using In-Cache-Line Logging as described in the following.

3.2.1 In-Cache-Line Logging.

To implement undo logging, our algorithm logs the current value of a memory location before it is modified for the first time in each epoch. In the event of a failure, the recovery procedure can *undo* all the modifications made during the crashed epoch by replacing them with the logged values. If the same memory location is updated several times during an epoch, no action needs to be taken for the later modifications.

ResPCT API	Description
InCLL_data<T>	Template for InCLL variables
`init_InCLL`(InCLL_data<T> *l, T val)	Initializes InCLL for variable `l`
`update_InCLL`(InCLL_data<T> *l, T val)	Updates variable `l` and its log
`add_modified`(void *addr)	Registers the address of a modified variable
`RP`(int id)	Identifies a Restart Point
`checkpoint_allow/prevent`(...)	Allows/prevents checkpoint occurrence

Table 1. ResPCT API

To be correct, the algorithm must ensure that the log is updated in NVMM before the corresponding memory location. Undo logging is usually implemented as a separate data structure [8], which implies using costly flush and fence instructions. To avoid this cost, ResPCT leverages In-Cache-Line Logging proposed by Cohen *et al.* [9].

InCLL places a log entry in the same cache line as the memory location that requires logging. More specifically, Figure 2 presents the template used to create a variable controlled by InCLL. It includes three fields: (i) `record` contains the current value of the variable, (ii) `backup` is the logged value of the variable before it is modified at the beginning of the epoch, and (iii) `epoch_id` stores the identifier of the corresponding epoch. By storing the log entry in the same cache line as the variable itself, InCLL leverages the property of the PCSO model that states that two writes to the same cache line reach NVMM in program order. This guarantees that the updated value will not reach NVMM before the corresponding log entry, without any flush or fence instruction. Note that, to use InCLL for data structures with multiple fields, InCLL is simply applied to each field separately. No further modification is required under the assumption that the original program is race-free.

One could argue that including log entries next to the variables themselves will have a negative impact on memory and cache footprint. To avoid this problem, Cohen *et al.* [9] use only a small number of InCLL backup entries for multiple variables, and move back to traditional undo logging if all InCLL entries are used. To remain a generic solution, ResPCT has a backup entry per variable. Note however that memory footprint does not necessarily increase much. First, the number of memory locations that require logging is often small as it concerns only a subset of the persistent variables (as we explain in Section 3.3.2). Second, padding is often used in shared data structures to avoid false sharing, which provides empty space to store InCLL fields.

```
1  struct InCLL_data<T>:
2      T record;
3      T backup;
4      uint64_t epoch_id;
```

Figure 2. InCLL template

```
5  /* Global variables */
6  bool timer = False;
7  uint64_t epoch;
8  bool perThread_flag[NB_THREADS] = {F, F, ...};
9  list<void*> to_be_flushed;
10 /* Local variables */
11 InCLL_data<int> *RP_id;
```

Figure 3. Global and local variables

The InCLL API is based on two functions. The init_InCLL() function is called after allocating a variable to initialize its InCLL fields (Figure 4, Lines 19-23). The update_InCLL() function replaces the usual store instruction to update a variable with logging (Lines 24-29). It tests epoch_id to check whether it is the first update of the variable in the current epoch. If so, it copies the current value of record in backup, writes the current epoch number in epoch_id, and finally, updates record.

The code of update_InCLL() would not be correct if a situation where two threads would call it on the same variable concurrently, could occur. However, our design relies on the assumptions presented in Section 2.1 to avoid dealing with this case. It assumes that when a thread calls update_InCLL() on a shared variable, it necessarily holds the corresponding lock.

After a crash, the recovery procedure executed by ResPCT is simple (Figure 5, Lines 60-65). It starts by retrieving the value of the failed epoch from NVMM, and for every variable r stored in NVMM with InCLL, it verifies if it was modified during the failed epoch (Line 63). If so, the variable is re-initialized from backup. This is sufficient to ensure that all variables in NVMM will be restored to the state at the beginning of the crashed epoch.

3.2.2 Tracking modifications.

As mentioned earlier, at the end of each epoch, the state is flushed to NVMM. ResPCT aims to flush only the cache lines that correspond to memory locations modified during the epoch. Hence, ResPCT maintains a list (to_be_flushed) of updated memory locations. The algorithm should avoid inserting the same address multiple times if it is modified several times during an epoch. Furthermore, it should avoid iterating through the list to find duplicates, for obvious performance reasons.

Thanks to InCLL, ResPCT solves this problem simply. It knows if a persistent variable is updated for the first time in an epoch using the corresponding InCLL epoch_id field. Hence, in the algorithm, the add_modified() function, that adds an address to the to_be_flushed list, is called by update_InCLL() only for the first update of a variable in each epoch (Figure 4, Line 28).

```
12 add_modified(void *addr):
13     to_be_flushed.append(addr);

14 flush_modified():
15     for (l in to_be_flushed):
16         clwb(l);
17     sfence();
18     to_be_flushed.empty();

19 init_InCLL(InCLL_data<T> *l, T val):
20     l->record = val;
21     l->backup = val;
22     l->epoch_id = epoch;
23     add_modified(l);

24 update_InCLL(InCLL_data<T> *l, T val):
25     if(l->epoch_id != epoch):
26         l->backup = l->record;
27         l->epoch_id = epoch;
28         add_modified(l);
29     l->record = val;

30 checkpoint_allow():
31     perThread_flag[$T_i$] = True;

32 checkpoint_prevent(mutex_t * mutex):
33     perThread_flag[$T_i$] = False;
34     if(timer):
35         perThread_flag[$T_i$] = True;
36         unlock(mutex);
37         while(timer){;}
38         lock(mutex);
39         perThread_flag[$T_i$] = False;

40 RP(int id):
41     update_InCLL(RP_id, id);
42     if(timer):
43         perThread_flag[$T_i$] = True;
44         while(timer){;}
45         perThread_flag[$T_i$] = False;

46 checkpoint(): // called periodically
47     timer = True;
48     flag = True;
49     while(flag):
50         flag = False;
51         for(k in 0..NB_THREADS):
52             if(!perThread_flag[k]):
53                 flag = True;
54                 break;
55     flush_modified();
56     epoch++;
57     clwb(&epoch);
58     sfence();
59     timer=False;
```

Figure 4. Description of the algorithm for thread T_i

```
recovery():
   Load failed_epoch from NVMM
   for (l : variable in NVMM with InCLL):
      if(l.epoch_id == failed_epoch):
         l.record=l.backup;
   epoch = failed_epoch;
```

Figure 5. Recovery procedure

Note that some data stored in NVMM do not require logging, as detailed in Section 3.3.2. To persist such data, the programmer must position `add_modified()` directly in the program code, right after the write operation. In this case, the same assumptions as for the calls to `update_InCLL()` apply. Namely, concurrent calls to `add_modified()` for the same variable cannot occur by construction.

3.3 Restart Points and persistent state

Identifying the program states where a checkpoint is allowed to be taken is important. Indeed, the identification of the persistent state, *i.e.*, which variables must be stored in NVMM, depends on it. The end of a period may occur at arbitrary times. If a checkpoint could be executed without any constraints on the program counter of each thread, every variable would have to be part of the persistent state to enable recovery in all cases.

A well known constraint for programs using locks is that checkpoints must not occur when a thread is in a critical section [6, 47]. With this constraint, a system can recover a consistent state after a crash without having to include the state of locks in the persistent state. This avoids having to deal with problems such as lock ownership recovery [30].

Existing approaches targeting lock-based applications assume that the persistent state only includes the shared data structures protected by locks [6, 23, 30, 47, 48]. If this is not the case, the programmer must explicitly manage the additional variables to be included in the persistent state during normal execution and at restart. This approach is good enough for programs that manipulate mostly shared data structures (*e.g.*, a key-value store). However it is not practical for more compute-intensive programs (*e.g.*, data processing and machine learning programs). In this case, defining the persistent state is mostly the responsibility of the programmer since most persistent variables are not shared variables.

ResPCT provides an API to position Restart Points in the source code of a program, as detailed in the following. The programmer positions `RP()` calls in her program, and then identifies the variables that belong to the persistent state.

3.3.1 Restart Points. The semantics of `RP()` enforces two properties. First, it ensures that a checkpoint occurs only when all threads have reached a restart point (Line 43 and Lines 49-54). Second, it blocks threads in `RP()` as long as the checkpoint is running (Line 44). The latter guarantees a quiescent state when the checkpoint is taken [43]. Note that these properties do not restrict parallelism between the program threads beyond synchronizing at checkpoints.

The `RP()` function takes as parameter an identifier that must be unique for each `RP()` call and the same in every run of the program. At checkpoint time, a thread stores this identifier in the thread-local variable `RP_id` located in NVMM (Figure 4, Line 41). This information is used during recovery to resume execution from the same point. Note that the variable `RP_id` itself requires InCLL to be able to roll back it if a crash occurs while checkpointing.

3.3.2 Identifying persistent state. It is programmer's responsibility to identify the persistent state of the program. After the programmer has positioned RPs, she deduces which variables must persist as we explain below. Our experiments with modifying existing programs (see Section 5) show that this task is not complex.

There are two constraints to the positioning of RPs. First, an RP may not be inside a critical section. Second, all program threads must eventually encounter an RP. This is essential to ensure that a checkpoint is eventually taken.

Once the RPs have been positioned, the following rules identify the variables that belong to the persistent state and/or require InCLL. The rules are: (i) If the scope of a variable *includes* an RP, then it belongs to the persistent state; (ii) If, after an RP, the first operation on the persistent variable is a read and the variable gets modified at any later point in the program, then the variable requires logging.

More technically, the second rule states that logging should be applied to any persistent variable that makes a sub-part of the program starting from an RP not idempotent.

A sub-part of a program is not idempotent if the sequence of operations on a variable starts with a write-after-read (WAR) dependency. Table 2, inspired from the explanations provided by Kruijf *et al.* [15], illustrates the notion of read-after-write (RAW) and WAR dependencies, and their relation to idempotence. The RAW sequence of instructions is idempotent since executing it multiple times will always lead to the same result. On the other hand, assuming that the initial value of x is 0, executing the WAR sequence a second time, would lead to $y = 8$ instead of $y = 0$. Hence, re-executing the WAR sequence after a crash requires logging to be able to roll back x to its initial value.

	RAW	WAR
Sequence	**x** = 5 y = **x**	y = **x** **x** = 8
Idempotent?	Yes	No

Table 2. RAW and WAR dependencies

We use the code snippet presented in Figure 6, that computes x^p, to illustrate these rules. Figure 6a is a version of the code where multiple RPs have been inserted. Figure 6b shows the ResPCT version of the code. It assumes that an `alloc_in_nvmm()` function is available to allocate variables in NVMM. Note that this example is not very realistic, as several RPs have been inserted in a code snippet that includes few instructions. Positioning RPs in this way is done for illustrative purposes.

In this example, all variables except i must be stored in NVMM. If a thread restarts from the RP at Line 4, variable i will be reinitialized, and if it restarts from the RP at Line 7, it is not needed anymore.

Variable x requires logging, because the sequence of instructions between lines 4 and 7 is not idempotent. Without InCLL, if a thread restarts from the RP at Line 4, there is no guarantee on the value that will be returned by the read operation at Line 6. It could be the initial value of x or some x^i. In the latter case, the code snippet will incorrectly compute x^{i+p} instead of x^p after the crash. On the other hand, variable p does not require logging, as it does not introduce any WAR dependency: p is written only once and never modified. As illustrated in Figure 6b, Line 6, the programmer needs to call `add_modified()` explicitly when such variables is modified.

Figure 6 can be used to illustrate that carefully positioning RPs can reduce the size of the consistent state and limit the need for logging. Removing the RP at Line 4 would remove variable p from the persistent state. If the whole snippet was only surrounded by two RPs, no variable would need logging. It is tempting to conclude from this example that inserting as few RPs as possible is the best solution to improve performance. However, the programmer should still insert enough RPs to ensure that checkpoints are not delayed significantly after the end of the period.

3.3.3 Handling condition variables .

ResPCT also supports condition variables for thread synchronization. However it should be handled with care to avoid deadlocks, since threads can be blocked waiting on a condition variable. All threads must reach an RP for a checkpoint to complete. The threads that reached an RP are blocked until the checkpoint completes. If the only way for a thread waiting on a condition variable to be woken up, is to be signaled by one of the thread that is blocked on an RP, a deadlock occurs.

ResPCT provides an API to deal with this problem. The programmer must follow two rules illustrated in Figure 7: (i) surrounding wait calls on condition variables with a `checkpoint_allow()` and a `checkpoint_prevent()` call, and (ii) positioning an RP immediately before entering the critical section.

The `checkpoint_allow()` call ensures that a thread does not prevent a checkpoint from completing, while the thread is waiting at `cond_wait()`. After `cond_wait()` returns, before resuming the program execution, `checkpoint_prevent()` disables the occurrence of checkpoints (Line 33 of Figure 4). However, it checks `timer` to detect whether there is an ongoing checkpoint (Line 34). ResPCT may have initiated a checkpoint while the thread was blocked in `cond_wait()`. If so, the thread must wait for this checkpoint to end. Hence, `checkpoint_prevent()` allows the checkpoint occurrence (Line 35), waits until it is finished, and revokes the permission afterwards (Line 39).

```
1 RP(id1);
2 int x=2
3 int p = 10
4 RP(id2)
5 for(int i=0; i<p; i++):
6     x = x*x
7 RP(id3)
8 print(x)
```

(a) Code with RPs

```
1  RP(id1)
2  InCLL_data<int> x= ↩
       alloc_in_nvmm()
3  init_InCLL(&x, 2)
4  int p = alloc_in_nvmm()
5  p=10
6  add_modified(&p)
7  RP(id2)
8  for(int i=0; i<p; i++):
9      update_InCLL(&x, x*x)
10 RP(id3)
11 print(x)
```

(b) Final code with logging

Figure 6. Example of code with ResPCT

```
RP();
lock(mutex);
while(...):
    checkpoint_allow();
    cond_wait(..., mutex);
    checkpoint_prevent(mutex);
...
unlock(mutex);
```

Figure 7. Code with a condition variable

Additionally, note that `checkpoint_prevent()` takes a lock as input parameter. Indeed, while it is waiting for the ongoing checkpoint to terminate, `checkpoint_prevent()` must temporarily release the lock grabbed at `cond_wait()` to avoid a similar deadlock scenario as described earlier. The lock is re-acquired after the checkpoint ends (Lines 36-38).

Calling `RP()` immediately before the critical section, ensures that if a checkpoint completes while a thread waits on the condition variable and a crash occurs, the thread restarts from the entrance of the critical section. Otherwise, it would restart from the preceding RP and re-execute the instructions between the RP and the beginning of the critical section. However, the recovery would not roll back the corresponding updates since they occurred before the checkpoint.

This way of handling condition variables is correct only if the program does not execute any store instruction between the entrance of the critical section and the call to `cond_wait()`. Otherwise, these updates would be re-executed, although other threads might have already seen them. However, we posit that most algorithms based on condition variables start a critical section by checking if it should wait on the condition variable. As such, our design covers most of the practical use cases.

Note that this approach can be generalized to any blocking function call. To deal with a blocking function call outside a critical section, the programmer can use a simplified version of `checkpoint_prevent()` that does not manipulate locks.

4 Proof

Due to space constraints, we only provide here a sketch of the proof. The proof is in two parts. First, it shows that a deadlock-free program modified with ResPCT remains deadlock-free. Then, it demonstrates that ResPCT provides buffered durable linearizability.

The proof assumes that ResPCT is applied to a linearizable algorithm A as defined in [22]. In an execution of A, synchronizations between threads create happens-before relation, noted $\prec$, between memory accesses, as defined in the C++ memory model [5]. Given a memory location l stored in NVMM, we note $NV(l)$ its value in NVMM and $V(l)$ its value taking into account the most recent update that might not have reached NVMM yet. With ResPCT, the execution of the algorithm is divided into epochs. Epochs are identified by a monotonically increasing sequence number. The value of l in NVMM during epoch i is noted $NV_i(l)$. Note that $NV_i(l)$ might change several times during an epoch. We note $NV_i^{end}(l)$, the last value assigned to l in NVMM during epoch i. The epoch changes when the global variable *epoch* is incremented (Line 56 of Figure 4).

4.1 Liveness

We assume that the original algorithm A is deadlock-free and that the only synchronization mechanisms used in algorithm A are mutex locks.[4] We also assume that the algorithm is correctly modified to apply ResPCT. RPs are positioned so that all threads eventually call `RP()` and RPs are not inserted inside critical sections.

To prove that the algorithm is deadlock-free, we have to show that no thread will remain blocked forever when the checkpoint procedure starts. To this end, we introduce two lemmas related to the behavior of `RP()` when the `checkpoint()` procedure executes.

Lemma 4.1. *After the* `checkpoint()` *procedure starts executing, every thread eventually calls* `RP()`*.*

Proof. The call to `checkpoint()` only blocks threads in calls to `RP()`. Since RPs are not inserted inside critical sections, no thread is prevented from releasing a lock. As algorithm A is deadlock-free, threads cannot be blocked infinitely at the entrance of a critical section. Therefore, they will eventually reach the `RP()` calls. □

Lemma 4.2. *After the* `checkpoint()` *procedure starts executing, any thread that calls* `RP()` *is blocked.*

Proof. It follows directly from the construction of the algorithm (see Lines 44 and 47 of Figure 4). □

These two lemmas imply that when the checkpoint procedure is called, it will eventually terminate. It is the necessary condition to conclude that:

Proposition 4.3. *An algorithm modified with ResPCT is deadlock-free.*

4.2 Buffered durable linearizability

The proof focuses on memory locations that correspond to data stored in NVMM. It assumes that algorithm A was modified according to the rules defined in Section 3. All data required to restart A are stored in NVMM, and InCLL is applied to all locations that require logging.

The proof is in two steps. First, we present lemmas that allow concluding about the consistency of the state flushed to NVMM at the end of each epoch. Then, we focus on the rollback of partial updates that might have reached NVMM.

This invariant on epoch changes follows directly from Lemmas 4.1 and 4.2:

Invariant 4.4. *When the global variable epoch is updated (line 56), all threads are blocked in function* `RP()`*.*

We note $RP_i(t)$, the call to `RP()` by thread t that allows *epoch* to be updated to value $i+1$, and $Incr_i(epoch)$ the memory operation that updates *epoch* to $i+1$. Any load or store operation issued by a thread belongs to a single epoch. We note $E(op)$, the epoch of operation op.

Invariant 4.4 implies that $RP_i^{call}(t) \prec Incr_i(epoch) \prec RP_i^{exit}(t)$, where RP^{call} and RP^{exit} are the entrance (Line 40) and the exit (Line 45) of function `RP()`. Using these happens-before relations, we deduce the following lemma:

Lemma 4.5. *$\forall op_1, op_2$, two memory operations such that $op_1 \prec op_2$, if $E(op_2) = i$ then $E(op_1) \leq i$.*

Proof. We prove this by contradiction. We assume that thread t_1 executes op_1, thread t_2 executes op_2, and $E(op_1) > i$. Invariant 4.4 implies that $Incr_i(epoch) \prec RP_i^{exit}(t_1) \prec E(op_1)$. Since $E(op_2) = i$, $E(op_2) \prec RP_i^{call}(t_2) \prec Incr_i(epoch)$. This implies that $op_2 \prec op_1$, which is a contradiction. □

Lemma 4.5 states that the volatile memory state at the end of epoch i is a consistent cut of the history of the memory operations during the execution of algorithm A.

Then, we introduce Lemma 4.6. It states that all updates executed during an epoch reach NVMM at the end of the epoch.

Lemma 4.6. *$\forall l$ such that $V_i(l) \neq NV_{i-1}^{end}(l)$, then $NV_i^{end}(l) = V_i^{end}(l)$.*

Proof. Invariant 4.4 implies that no store operation executes concurrently with the increment of the global variable *epoch*. By construction, calling `update_InCLL()` is the only way

[4] The proof can be trivially extended to the case of condition variables, as long as the modifications specified in Section 3.3.3 are applied.

to update memory locations. The first time a location is updated in epoch i by thread t, its address is added to the `to_be_flushed` list (Line 28 of Figure 4) before thread t calls RP(). Since, `flush_modified()` is called immediately before *epoch* is updated to $i + 1$ (Line 55), this concludes the proof. □

Finally, we conclude about the consistency of the state written to NVMM at the end of each epoch.

Lemma 4.7. *The memory state in NVMM at the beginning of epoch i corresponds to a linearizable execution of A up to the end of epoch $i - 1$.*

Proof. It follows directly from the lemmas 4.5 and 4.6. □

To reason about the rollback of partial updates, we start by introducing two lemmas about properties of InCLL fields.

Lemma 4.8. *Given a store operation that assigns value x to l in epoch i, if $NV_i(l) = x$, then $NV_i(l.epoch\_id) = i$.*

Proof. The proof follows directly from the construction of the algorithm (see Lines 24-29 of Figure 4) and the properties of the PCSO model (defined in Section 2.1). □

Lemma 4.9. *Given a location l modified in epoch i, if $NV_i(l.epoch\_id) = i$, then $NV_i(l.backup) = NV_{i-1}^{end}(l)$.*

Proof. The fact that $NV_i(l.backup)$ is always updated before $NV_i(l.epoch\_id)$ follows from the same arguments as used for Lemma 4.8. The fact that $NV_i(l.backup)$ is updated to $NV_{i-1}^{end}(l)$ is a direct consequence of Lemma 4.6. □

About the recovery procedure, we state that:

Lemma 4.10. *After a crash in epoch i, at the end of the recovery procedure, $\forall l : V(l) = NV_{i-1}^{end}(l)$.*

Proof. We distinguish between two cases. For a memory location l, either (a) $NV(l.epoch\_id) < i$, or (b) $NV(l.epoch\_id) = i$.

In case (a), $V(l)$ will be equal to $NV(l)$ (Figure 5 Line 63). Since $NV(l.epoch\_id) < i$, $NV(l)$ was not modified in epoch i (Lemma 4.8). Lemma 4.6 implies that $NV(l) = NV_{i-1}^{end}(l)$, which concludes the proof for this case.

In case (b), the recovery procedure sets $V(l)$ to $NV_i(l.backup)$ (Line 64). It follows directly from Lemma 4.9 that $V(l) = NV_{i-1}^{end}(l)$. □

Lemma 4.10 states that ResPCT will roll back all updates issued during a failed epoch that have reached NVMM. Together with Lemma 4.7, it leads to the following proposition:

Proposition 4.11. *A linearizable algorithm A running with ResPCT is buffered durably linearizable.*

5 Evaluation

In this section we evaluate ResPCT using micro benchmarks and compare it with other techniques. We analyze its failure-free performance, study the impact of the checkpoint frequency, and evaluate its recovery time. We also present the results of applying ResPCT to real-world workloads.

Implementation. We implemented a prototype of ResPCT as a shared library in C. To achieve high performance and ensure correctness, we paid special attention to cache line alignment[5] and false sharing. We also used compiler fences to avoid the reordering of instructions in `update_InCLL()`. A per-thread list implements the `to_be_flushed` list. As suggested in [46], a pool of flusher threads flushes data to NVMM in parallel during checkpoints.

Experimental setup. We use a dual socket Linux machine, equipped with 384 GiB of DRAM, 1.5 TiB (12×128 GB) of Intel's Optane DC Persistent Memory [25] and two 16-core (32 hardware threads) Intel Xeon Gold 5218 processors with a 2.30GHz frequency. We disabled Turbo Boost Frequency. The Linux kernel version is 5.4. We compiled the programs using gcc 9.2.1 with the highest level of optimization.

We use NVMM DIMMs in direct-access (DAX) mode and configure them in system-ram mode [38] to access NVMM as a separate NUMA node. To avoid performance variations due to NUMA effects, we evenly distribute threads between the two processors. Furthermore, to improve locality, we use a one-to-one mapping between the program threads and the flusher threads, and pin them to the same cores.

All presented results are averaged over 10 runs. Since the standard deviation is below 5% for all experiments, we did not include this information in the graphs to avoid clutter. If not mentioned otherwise, experiments are run with 64ms checkpoint intervals.

5.1 Micro Benchmarks

We select two concurrent data structures to extensively compare ResPCT with other systems: a Queue and a HashMap. We implement a queue protected by one lock. The size of the elements is 8 bytes. For the HashMap, we use the code from the Synch framework [28]. It uses one lock per bucket. The size of the keys and the values is again 8 bytes. These two data structures enable us to test different patterns. The HashMap provides a lot of parallelism while the Queue induces a lot of contention on the lock. The algorithms are implemented using `pthread` mutex locks.

These experiments compare ResPCT to other systems. Our evaluation includes two state-of-the-art solutions for buffered durable linearizability: `Montage` and `PMThreads`. It also includes two solutions that implement durable linearizability: `Clobber-NVM` and `Quadra/Trinity`. Finally, for each data structure, we consider specific algorithms: a

[5] Leveraging `posix_memalign()` function.

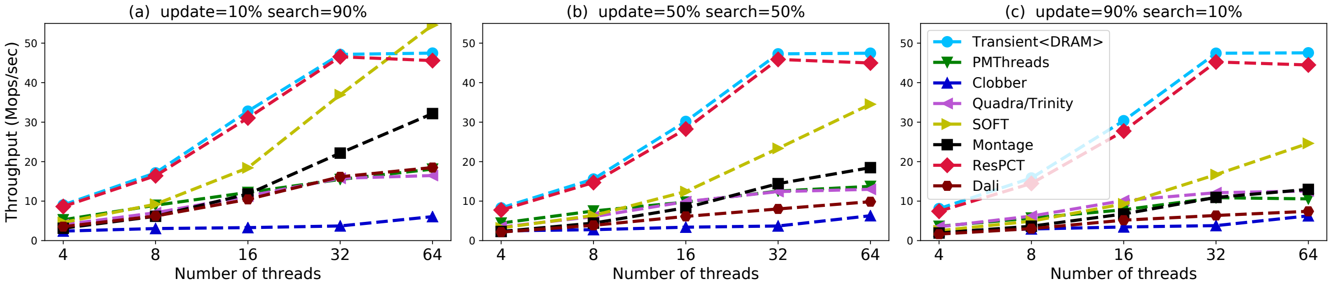

Figure 8. Performance with the HashMap

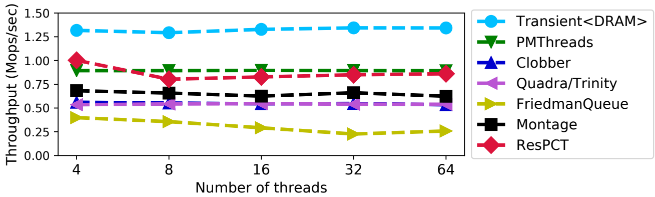

Figure 9. Performance with the Queue

persistent lock-free queue [17] (FriedmanQueue), a persistent lock-free hash table [51] (SOFT), and a persistent hash table based on checkpointing [36] (Dali). To have a basis for comparison, we also evaluate the unmodified transient code on DRAM (labeled Transient<DRAM>).

For PMThreads, we modified the source code provided by the authors[6] to parallelize the checkpointing procedure. We identified that the single flusher thread was the bottleneck of the system. For Quadra/Trinity[7] and Clobber-NVM[8], we use the source code provided by the authors. All other algorithms are the implementations provided by the authors of Montage[9]. For Quadra/Trinity, we use the most efficient algorithm for each data structure: Quadra for the Queue and Trinity for the HashMap. In their paper, the authors propose a solution based on flat combining [21] to obtain a very efficient implementation of critical sections. To have a fair comparison, we evaluate Quadra with a pthread lock for the Queue instead.

Figure 8 presents the results for the HashMap with 10^6 buckets and 2×10^6 keys. We evaluate 3 workloads with the following update/search ratios: 1:9, 1:1 and 9:1. Half of the updates are inserts and half are deletes. Figure 9 presents the performance of the Queue with an enqueue/dequeue ratio of 1:1. Both data structures are pre-filled before starting the evaluation, with 1k element for the Queue and 1M key-value pairs for the HashMap. Both figures show the performance (in Mops per second) as a function of the number of threads.

When comparing the performance of ResPCT with the transient algorithms executed on DRAM, we observe at most 9% overhead with the HashMap. With 64 threads, it is 4% with 90% of search operations. For the Queue, the highest overhead with ResPCT reaches 37%. However, as we show in Section 5.2, half of this overhead is also observed when executing the transient program on NVMM.

The results presented in Figure 8 show that, at large core count, ResPCT significantly outperforms all existing buffered durably linearizable solutions (with at least a 3.1× speedup for the write-intensive workload) and all durably linearizable solutions (with at least a 2.7× speedup for the write-intensive workload). Only the dedicated lock-free SOFT algorithm manages to outperform our solution for the read-intensive workload. But in this case, this algorithm is even faster than the transient lock-based algorithm.

The tests with the Queue (Figure 9) show that ResPCT outperforms all systems except PMThreads. PMThreads is very efficient in this case because it is based on a shadowing approach that creates a copy of the data in DRAM. However, the main performance overhead of PMThreads comes from tracking modifications when the persistent state is large, as its performance with the HashMap illustrates.

The performance of Montage, the other system based on checkpoints, is rather good. However, it is significantly lower than ResPCT for two main reasons. First, Montage puts more stress on the memory allocator as each update requires allocating a new element. This is illustrated by its limited performance for the write-intensive workload with the HashMap. Second, Montage requires managing additional metadata for some algorithms. For the Queue, the management of a global sequence number updated inside the critical section affects its performance.

Finally, Quadra/Trinity, which guarantees durable linearizability, has performance that is close to PMThreads and Montage. This illustrates the advantage of InCLL over other logging techniques. ResPCT, achieves even higher performance by targeting buffered durable linearizability.

[6] https://doi.org/10.5281/zenodo.3756416
[7] https://doi.org/10.5281/zenodo.4362578
[8] https://doi.org/10.5281/zenodo.4322233
[9] https://github.com/urcs-sync/Montage

5.2 Detailed analysis of ResPCT

Overhead analysis. To analyze the overhead of ResPCT in details, we evaluated three additional configurations: (i) The transient program on NVMM, (ii) ResPCT with InCLL, modification tracking, but no checkpoint (`ResPCT-InCLL`), and (iii) ResPCT with the complete algorithm except flushing the modified data on NVMM (`ResPCT-noFlush`). Figure 10 presents the results with the Queue and two workloads for the HashMap: read intensive (10% update, 90% search) and write intensive (90% update, 10% search). We use 64 threads for this evaluation. The figure shows the throughput normalized to Transient<DRAM>.

We can observe that the higher overhead with the Queue mostly comes from running on NVMM. All other costs are small. Comparing `Transient<NVMM>` to `ResPCT-InCLL` shows that InCLL enables ResPCT to implement a very efficient undo log and modification tracking solution.

The comparison between `Transient<NVMM>` and `ResPCT-InCLL` performance shows that InCLL has a negligible impact on performance. Both the original Queue and Hashmap programs use up the entire space in cache lines. Thus implementing InCLL increases the memory footprint. However, it has no significant impact on performance.

The comparison between ResPCT and `ResPCT-noFlush` with the Hashmap illustrates that, a checkpoint takes little time compared to the epoch duration even when the amount of data to flush is large. For the write-intensive workload, 700k addresses are flushed on average during each checkpoint, which is around 6× more than with the read intensive workload. This is because cache lines are flushed in parallel during checkpoints.

This detailed analysis further explains the results of Figure 8, where the performance of ResPCT for the read and write-intensive workloads do not differ much, contrary to what is observed with other systems. The overhead of ResPCT is 4% for the read-intensive workload and 8% for the write-intensive workload. In comparison, other techniques suffer more with the write-intensive workload due to increased number of flushes (durable linearizability), costly modification tracking (PMThreads), or costly updates (requiring memory allocation in Montage).

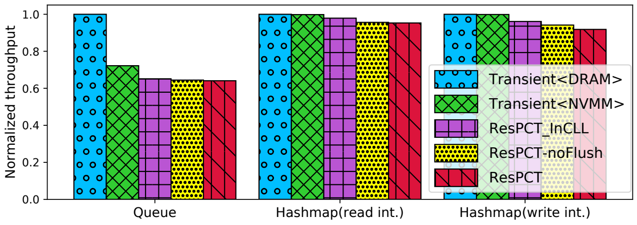

Figure 10. Detailed analysis of ResPCT overhead

Impact of the checkpoint period. Figure 11 illustrates the impact of the checkpoint period duration on performance, for the HashMap with the write-intensive workload and 64 threads. The checkpoint period varies between 1ms and 64ms. Since the results are qualitatively the same with the read-intensive workload and with the Queue, we chose not to include them to avoid clutter.

The results show that we can execute checkpoints every 16 ms, with almost no performance overhead. With 4-ms epochs, ResPCT is still better than any other systems we have evaluated (considering 64-ms epochs for periodic approaches), except SOFT. Since ResPCT delays checkpoints until all threads reach an RP, one could wonder if the effective period duration is longer than the expected one. However, with 4-ms epochs, we measured an effective period duration of 5 ms, which we think is an acceptable delay.

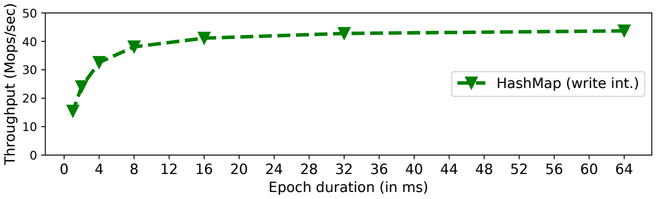

Figure 11. ResPCT with different checkpoint periods

Recovery time. Figure 12 shows the performance of the recovery procedure for the HashMap with the write-intensive workload. We measured the time needed to reconstruct a consistent global state after a crash. Results are presented as a function of the number of buckets in the HashMap. There are at most 2 elements per bucket. For this test, we execute the workload for a few seconds before simulating a crash.

Since the HashMap is a highly concurrent data structure, we are able to parallelize its recovery procedure. 32 threads are used during the recovery procedure. With a 4M-buckets HashMap, recovery takes less than 240 ms.

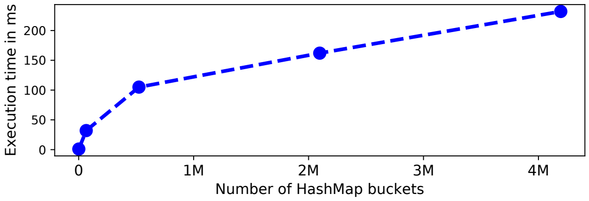

Figure 12. Recovery performance for the HashMap

5.3 Real-world workloads

This section presents the evaluation of ResPCT using real-world workloads. We consider compute-intensive programs coming from two benchmark suites, as well as the popular in-memory key-value store Memcached [34].

Compute-intensive workloads. To illustrate that ResPCT can also efficiently provide fault-tolerance for compute-oriented workloads, we evaluate it with four applications coming from the Parsec [4] and the Phoenix [41] benchmark suites. From Parsec, we select Dedup which implements a data processing pipeline that relies on condition variables to synchronize the stages of the pipeline. We also select Swaptions, a lockless application that uses the data-parallel parallelization model. Phoenix provides a set of benchmarks implemented using the MapReduce programming model. We test two of the benchmarks: Matrix Multiplication (MatMul) and Linear Regression (LR). For each benchmark, we chose a problem size that is big enough so that they run at least several seconds. We selected these benchmarks to be representative of different workloads: heavily lock-based (Dedup), lock-less (Swaptions), compute intensive (MatMul) and machine learning oriented (LR).

Figure 13 illustrates the performance of these applications, with the execution time normalized to Transient<DRAM>. All experiments were run using 64 program threads since this configuration offers the best performance for the transient versions. These results show that the overhead of ResPCT is between 17% and 21% for these real-world applications.

Positioning RPs. These experiments allow us to discuss about the positioning of RPs and its impact on performance.

For correctness, RPs can be placed anywhere in the code, except inside critical sections. In our experience, positioning RPs was straightforward as we chose to put an RP call after each logical block of code, for instance, after computing the value of each cell in the matrix multiplication, or after inserting an element in the hash table. However, positioning RPs naively might affect performance by: (i) uselessly increasing the persistent state size, or (ii) making some threads waiting for others at the checkpoint.

Swaptions and LR are examples where, with the initial positioning of RPs, we observed a slowdown of 4× and 9× respectively. We take the case of LR to illustrate.[10] We first placed RPs after processing each input data point, which lead to the 9× slowdown. However, identifying the problem was easy. Calling `update_InCLL()` for the modifications related to the processing of each point was too costly.

We modified the positioning of RPs to place them after processing a batch of 1000 points. This was enough to make the overhead drop to around 20%. Such performance improvement was only possible thanks to the flexibility of the manual insertion of RPs. Unlike other systems, where the persistent state is defined by critical sections [47, 48] or transactions [40], ResPCT allows programmers to adjust the persistent state without changing the application logic.

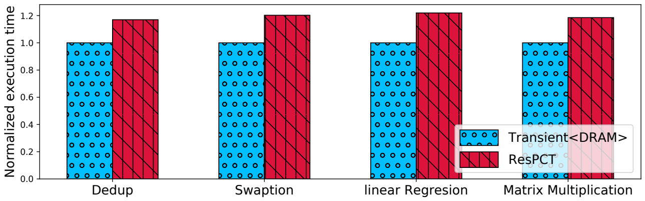

Figure 13. Performance with compute-intensive applications

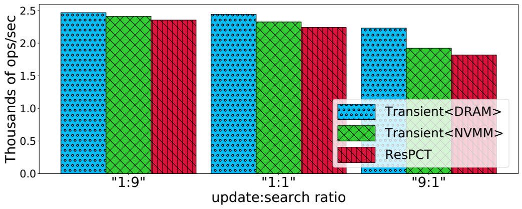

Figure 14. Performance of Memcached with 10^6 keys with value size of 100 bytes

We evaluated the impact of the positioning of RPs on the effective epoch duration. For LR, for a 64-ms and a 4-ms period, the measured duration was 65 ms and 5 ms respectively, indicating that checkpoints are not delayed. We explain this by the fact that each thread still makes RP calls frequently enough to avoid delaying checkpoints. Moreover, positioning RPs at the end of logical blocks makes all threads reach RPs almost at the same time in most cases.

Memcached. Memcached [34] is a popular in-memory key-value store which is used as an object cache by web applications. Similarly to existing evaluations [35], we modified Memcached to store the hash table of key-value objects in NVMM. We evaluate the asynchronous writes version, which immediately returns the response to the client without waiting for the object to become durable. It corresponds to the default consistency of RocksDB [16].

We generate workloads with the YCSB [12] benchmark, using 32 clients and 4 worker threads for the Memcached server. We warm-up the hash table by inserting 1M key-value pairs (i.e., YCSB load phase) and then perform 1 million put/get operations based on the workload characteristics. We consider read-intensive (90% reads and 10% writes), write-intensive (10% reads and 90% writes) and balanced (50% reads and 50% writes) workloads.

Figure 14 shows the throughput in thousands of operations per second. The overhead of ResPCT is 5% for the read-dominant workload which is the most common pattern for this kind of systems. Even for the write-dominant workload, the performance overhead is only 18.5%. We also measured the latency. The overhead was at most 10% with write-intensive workload. Comparing Transient<DRAM> with Transient<NVMM> shows that most of this overhead simply comes from executing on NVMM.

[10] The problem and the solution were very similar with Swaptions.

	Added or modified LoC	Original LoC	Modifications in %
Hashmap	19	282	6.74%
Queue	31	607	5.11%
Dedup	244	3351	7.28%
Swaptions	29	1137	2.55%
MatMul	13	196	6.63%
LR	69	138	50.00%
Memcached	97	20520	0.47%

Table 3. Number of lines modified in the applications

5.4 Modifying the original code to apply ResPCT

In this section, we describe the amount of effort that was required to modify existing applications to integrate ResPCT.

It did not took us more than a few hours to modify each program. Most modifications were to insert `update_InCLL()` and `add_modified()` calls to implement logging. This requires identifying the variables that belong to the persistent state. Based on our experience, it is straightforward when RPs are positioned after logical blocks of code.

Table 3 illustrates the number of lines of code (LoC) that we had to add or modify in the source code of applications. The modifications represent between 2.5 and 7.2% of the LoC. LR and Memcached are not representative for different reasons. A large part of the code of Memcached is dedicated to handle client-server communication. Since we did not modify this part of the code, modifications only represent 0.47% of the LoC. In the case of LR, the percentage of modifications is very high because we had to duplicate part of the code to position RPs as described in Section 5.3.

6 Discussion

The section discusses some possible limitations of ResPCT.

ResPCT impacts the data layout of programs as InCLL implies that the log for a variable is located in the same cache line as the variable itself. For instance, in the case of the Queue program, because of InCLL the elements of the queue are not stored at contiguous addresses as in the original program. Hence, accessing elements using pointer arithmetic requires using the corresponding `InCLL_data<T>` type. This is extra work for the programmer but considering the achieved performance, we think that it is acceptable.

Our experience shows that modifying programs manually to apply ResPCT is rather simple. Still, it would be good to be able to apply those changes automatically. While automating some of them (e.g. handling `cond_wait()`) could be easy, dealing with all the corner cases for detecting persistent variables and required logging can be difficult, especially when variables are accessed through pointers. As far as we know, even the existing *transparent* solutions [47] rely on the programmer to correctly manage persistent variables accessed outside of critical sections or transactions. Addressing this problem is an interesting direction for future work.

One could wonder whether a solutions like ResPCT is still needed when some techniques, such as the Enhanced-Asynchronous DRAM Refresh (eADR) feature from Intel [24], are proposed to include the caches in the persistent domain. However, it will take time before all servers are equipped with such technologies. Furthermore, since such approaches rely on batteries [1, 7], it raises questions regarding their energy efficiency and their reliability at scale, that might limit their acceptance. Hence, proposing efficient fault-tolerant solutions that can handle volatile caches is still valuable.

7 Conclusion

ResPCT is a general solution to provide fault-tolerance for multi-threaded applications running on top of NVMM using periodic checkpoints. ResPCT is based on In-Cache-Line Logging, which is used both to implement an undo log and a modification tracking solution. With this approach, tracking modifications requires executing only a few extra instructions when a store to a persistent variable is issued, and undo logging is implemented without explicitly synchronizing with NVMM. Furthermore, thanks to an API that allows programmers to explicitly position restart points in their program, ResPCT helps reducing the state that should be stored in NVMM, and so, improves performance. Evaluations show that ResPCT can significantly outperform all state-of-the-art solutions for the implementation of ubiquitous concurrent data structures. ResPCT has a low overhead for very diverse workloads, ranging from an in-memory KV store to MapReduce jobs. Because of its low overhead, ResPCT achieves high performance even with a checkpointing period as low as a few milliseconds.

Acknowledgments

We would like to thank our shepherd, Marc Shapiro, and the anonymous reviewers for their valuable feedback. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Seoul, South Korea). 111–124.

[2] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (Los Angeles, USA). 439–451.

[3] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High

performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis (SC)* (Seatle, USA). 1–32.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)* (Toronto, Canada). 72–81.

[5] Hans-J Boehm and Sarita V Adve. 2008. Foundations of the C++ concurrency memory model. *ACM SIGPLAN Notices* 43, 6 (2008), 68–78.

[6] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014), 433–452.

[7] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 16–31.

[8] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[9] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, USA). 441–454.

[10] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–24.

[11] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Vienna, Austria). 259–269.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)* (Indianapolis, USA). 143–154.

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Vienna, Austria). 271–282.

[14] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)* (Heraklion, Crete, Greece). 1–15.

[15] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)* (Beijing, China). 475–486.

[16] Facebook. 2017. RocksDB. http://rocksdb.org/.

[17] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices* 53, 1 (2018), 28–40.

[18] Ellis R Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)* (Santa Clara, USA). 1–14.

[19] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. 2018. Persistency for synchronization-free regions. *ACM SIGPLAN Notices* 53, 4 (2018), 46–61.

[20] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: a scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIXATC 19)* (Renton, USA). 913–928.

[21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)* (Thira, Santorini, Greece). 355–364.

[22] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[23] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia). 468–482.

[24] Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[25] Intel. 2021. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Brief announcement: Preserving happens-before in persistent memory. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Pacific Grove, USA). 157–159.

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

[28] Nikolaos D. Kallimanis. 2021. Synch: A framework for concurrent data-structures and benchmarks. *Journal of Open Source Software* 6, 64 (2021), 3143.

[29] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017), 329–343.

[30] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Fukuoka, Japan). 258–270.

[31] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (Monterey, USA). 295–310.

[32] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia). 499–512.

[33] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). 789–806.

[34] Memcached. 2018. memcached – a distributed memory object caching system. http://memcached.org/.

[35] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. *ACM SIGPLAN Notices* 52, 4 (2017), 135–148.

[36] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed*

Computing (DISC) (Vienna, Austria).

[37] pmem.io. 2022. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[38] The NDCTL project. 2022. NDCTL User Guide. https://docs.pmem.io/ndctl-user-guide/.

[39] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

[40] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1–15.

[41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)* (Phoenix, USA). 13–24.

[42] Andy Rudoff. 2017. Persistent memory programming. *Login: The Usenix Magazine* 42, 2 (2017), 34–40.

[43] Yuval Tamir and Carlo H Sequin. 1984. Error recovery in multicomputers using global checkpoints. In *In International Conference on Parallel Processing (ICPP)* (Lausanne, Switzerland). 32–41.

[44] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[45] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L Scott. 2021. A Fast, General System for Buffered Persistent Data Structures. In *International Conference on Parallel Processing (ICPP)* (Chicago, USA). 1–11.

[46] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. 2020. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Atlanta, USA). 427–439.

[47] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (London, UK). 623–637.

[48] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 346–359.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, USA). 169–182.

[50] Pantea Zardoshti, Michael Spear, Aida Vosoughi, and Garret Swart. 2020. Understanding and Improving Persistent Transactions on Optane™ DC Memory. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (New Orleans, USA). 348–357.

[51] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.