FFCCD: Fence-Free Crash-Consistent Concurrent Defragmentation for Persistent Memory

Yuanchao Xu
North Carolina State University
yxu47@ncsu.edu

Chencheng Ye
Huazhong University of Science and Technology
yecc@hust.edu.cn

Yan Solihin
University of Central Florida
Yan.Solihin@ucf.edu

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

ABSTRACT

Persistent Memory (PM) is increasingly supplementing or substituting DRAM as main memory. Prior work have focused on reusability and memory leaks of persistent memory but have not addressed a problem amplified by persistence, *persistent memory fragmentation*, which refers to the continuous worsening of fragmentation of persistent memory throughout its usage. This paper reveals the challenges and proposes the first systematic crash-consistent solution, Fence-Free Crash-consistent Concurrent Defragmentation (FFCCD). FFCCD resues persistent pointer format, root nodes and typed allocation provided by persistent memory programming model to enable concurrent defragmentation on PM. FFCCD introduces architecture support for concurrent defragmentation that enables a fence-free design and fast read barrier, reducing two major overheads of defragmenting persistent memory. The techniques is effective (28–73% fragmentation reduction) and fast (4.1% execution time overhead).

CCS CONCEPTS

• **Hardware → Non-volatile memory**; • **Software and its engineering → Garbage collection**.

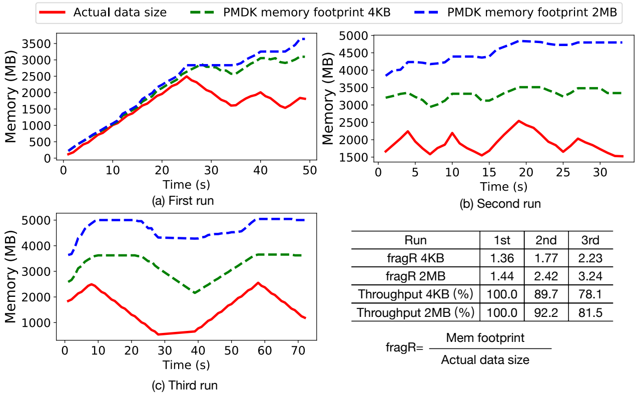
KEYWORDS

Non-volatile memory, Persistent memory, Memory management, Garbage collection, Defragmentation

ACM Reference Format:
Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. 2022. FFCCD: Fence-Free Crash-Consistent Concurrent Defragmentation for Persistent Memory. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527406

1 INTRODUCTION

Persistent memory (PM) is emerging as a promising supplement or substitute of DRAM as the main memory, offering higher density, better scaling potential, lower idle power, and non-volatility, while retaining byte addressability [1, 45, 49, 51]. Prior studies [12, 15, 18, 33, 79] have focused on reusability and memory leaks of PM but have not addressed a problem amplified by persistence, *persistent fragmentation*, which is a situation when much of the memory is allocated in a large number of non-contiguous blocks, leaving a good portion of the memory unallocated, but unusable for most typical scenarios. Persistent fragmentation is distinct from memory leak, which arises due to unclaimed dead objects. Memory leaks can worsen fragmentation, but solving memory leaks does not necessarily remove fragmentation. PM fragmentation leads to unsustainable PM uses (i.e., deteriorating performance throughout the PM lifetime). The main cause of fragmentation is the memory allocator's assumption that objects have a short lifetime, and that fragmentation is a temporary problem that disappears when the program terminates, which is the case in DRAM, but not in PM. Unlike in DRAM, temporal memory fragmentation becomes permanent in PM as the PMOP heap persists across runs and may even live for years with different versions of applications. Without proper treatment, fragmentation in PM keeps deteriorating throughout its lifetime. The issue worsens as the large capacity provided by PM necessitates the use of huge pages [29, 50, 66].



(a) First run

(b) Second run

(c) Third run

Run	1st	2nd	3rd
fragR 4KB	1.36	1.77	2.23
fragR 2MB	1.44	2.42	3.24
Throughput 4KB (%)	100.0	89.7	78.1
Throughput 2MB (%)	100.0	92.2	81.5

$$\text{fragR}=\frac{\text{Mem footprint}}{\text{Actual data size}}$$

Figure 1: PM fragmentation worsens across runs of Echo, a key-value store.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527406

Figure 1 shows an example with a growing memory footprint on subsequent uses of a PMOP. The figure plots the actual data size (i.e., total allocated memory from the application) versus PMDK memory footprint (i.e., total memory pages allocated to application data by the OS) of Echo, a key-value store application from WHISPER benchmark suite [59]. Each subsequent run inherits the persistent fragmentation from the previous run and increases it. By the third run, the footprint doubles or triples the actual data size (a fragmentation ratio of 2.27 and 3.24 on 4KB and 2MB settings), and the throughput also declines by nearly 20% compared to the first run.

Idle-time defragmentation in disks and SSDs is less attractive for PM for several reasons. First, the higher PM throughput means that fragmentation will grow much more quickly in PM. There are also smaller idle windows to perform defragmentation in PM; scheduling defragmentation at a wrong time may result in an unexpected long latency of PMOP open(), the waiting for ongoing defragmentation to finish [43, 44, 75].

Programming language defragmentation solutions fall into two classes. The first is through ad-hoc treatments [55, 71], which use memory management customized to a particular application to perform stop-the-word defragmentation. They do not offer a general solution to PM fragmentation. The second is through Garbage Collection (GC). The primary task of GC is to reclaim reusable memory space automatically. Most GCs would compact useful data objects together into a consecutive space, naturally reducing fragmentation. So memory leak prevention and defragmentation in modern GCs for managed languages prompt a question of how they should be used for memory defragmentation in PM.

There have been efforts to adopt GC for PM, C/C++, and managed languages (e.g., Java). They, however, did not offer efficient solutions to PM defragmentation and did not address crash consistency. The work on C/C++ [12, 15, 18, 33] uses stripped GC (called conservative GC), which reclaims data objects but without compacting objects, and hence does not defragment PM. The work on managed languages [2, 30, 47, 74] simply assumes the use of standard GCs with defragmentation. Espresso [84] is so far the first and only work proposing GC on JAVA for PM by addressing the need for PM crash consistency. Espresso's approach of directly adding several cache writebacks (*clwb*) and a store fence (*sfence*) instruction after each GC-caused PM write leads to high overheads as it makes every GC-caused PM write subject to a full PM access latency. So no prior work achieves the two important PM-specific goals at the same time, crash consistency and efficiency:

- *Crash consistency.* Because PM keeps persistent data for reuse across runs, crash consistency requires that PM data remain consistent across system crashes or program failures. Since GC with defragmentation moves data objects in memory, they present special hazards for crash consistency. Without proper treatments, partially moved objects and partially updated references may cause inconsistency and permanent corruption of PM data.
- *Efficiency.* GC with defragmentation adds substantial runtime overheads. While it may be tolerable for managed languages on a volatile memory, it is not tolerable for PM due to three reasons: (i) Demands: PM has been used so far primarily for system-level tasks (e.g., database, key-value store) with stringent latency and throughput requirements, and C and C++ rather than management languages are popular choices for them. (ii) Hardware: PM write latency is several times larger than that of DRAM, worsening the GC overheads. (iii) Operations: Many operations must happen at each write on PM to maintain crash consistency. By default, each object movement, each reference update, and metadata update need to be followed by several *clwb* instructions and an *sfence* instruction to guarantee that the writes reach PM and that the PM is consistent after these writes. As a result, direct integration of the standard GC on PM while keeping crash consistency adds significant overheads to typical programs (Section 3.2).

In this work, we propose a Fence-Free Crash-consistent Concurrent Defragmentation (FFCCD), the first concurrent defragmentation that reduces persistent fragmentation for PM and removes persistent memory leaks while achieving both crash consistency and high efficiency. FFCCD comes in two flavors; software-only and hardware-supported. FFCCD leverages careful analysis of idempotent operations and post-crash states, allowing us to eliminate all GC-caused `clwb` and `sfence` instructions, keeping normal operations and data compaction fast, at the expense of more post-crash recovery work. This design is applicable to applications in both managed and unmanaged languages without any architecture support. In addition, we also propose architecture support that further accelerates FFCCD. With FFCCD, memory fragmentation is reduced by 39.3% on concurrent PM data structures and two key-value store applications and 42.7% on some benchmarks. As memory fragmentation decreases, application execution time decreases as well. Consequently, with FFCCD, the improvement in application performance offsets defragmentation overheads substantially, resulting in only a small increase in the total execution time (4.1%).

Overall, in this work, we make the following contributions:

- We analyze the challenges and present the first systematic effort to address PM fragmentation.
- We propose FFCCD and its architecture support to serve for fence-free crash-consistent concurrent fragmentation on PM to achieve both crash consistency and efficiency.
- We evaluate the efficacy of FFCCD, showing 28–73% reduction in memory fragmentation with a 4.1% execution time overhead.

2 BACKGROUND

2.1 Persistent Fragmentation

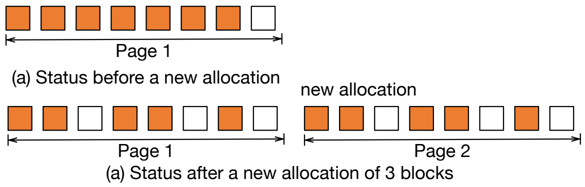

Page 1
(a) Status before a new allocation
new allocation
Page 1
Page 2
(a) Status after a new allocation of 3 blocks

Figure 2: Example of memory fragmentation

Fragmentation is an inherent limitation of memory allocators without compaction[40, 41]. Figure 2 shows an example of memory fragmentation. Each solid box represents an allocated memory block (16 bytes), and each empty box is a free block. At the moment, shown in Figure 2 (a), even though there are enough free blocks on page 1 to meet the new request, the consecutive free space is not sufficient. Consequently, the memory allocator has to allocate a new page for the new request. Fragmentation causes inefficient memory usage. It also degrades program performance as it unnecessarily increases the memory footprint and TLB entries. Unlike in DRAM, fragmentation in PM persists after a run.

2.2 Persistent Memory Programming Support

2.2.1 PMOP. There are at least two paradigms for using PM. One may use it as storage to host a file system; the other uses it via a new abstraction where a data structure is wrapped into a *persistent memory object pool* (PMOP), allowing data structures to be hosted persistently in physical memory without file backing [88]. PMOPs may combine some features of a file system (naming, permission, durability, and sharing) and some features of data structures (pointer-rich, address space mapping, purely load/store access. In this paper, we assume the latter.

A PMOP may be a container for a data structure that lives beyond process termination and system reboots. A PMOP requires several properties to be supported: *crash consistency* allows a PMOP to remain in a consistent state across software crashes or power failures, and *relocatability* where a PMOP can be relocated to different VA ranges in different runs [18, 33, 81, 92] of an application.

To support relocatability, each pointer (64-bit) used in a data structure consists of a PMOP ID and an offset within this PMOP. PMDK `libpmemobj` [33] have described interfaces for manipulating pools and objects, which we adopt. Every PMOP has at least one entry point called a *root*, which stores the start address of a set of connected persistent objects [15, 18].

```
1  struct node{int data; POBJ_ENTRY(node)* next;};
2  void insert_head(int value) {
3  PMEMobjpool * pop = pmemobj_create (path, layout(node),size,0666);
4    TOID(struct node) root = POBJ_ROOT(pop, struct node);
5    TX_BEGIN(pop) {
6      TOID(struct node) newnode = TX_NEW(struct node);
7      D_RW(newnode)->data = value;
8      if (TOID_IS_NULL(root)) {…}
9      else {TX_ADD_DIRECT (newnode); D_RW(newnode)->next=&D_RW(root);
10       TX_ADD_DIRECT (node); D_RW(root) = &D_RW(node);
11    }TX_END
12 }
```

Figure 3: List Insertion Example of PMDK libpmemobj.

2.2.2 Example. To illustrate PM programming model, Figure 3 shows a node insertion for a singly linked list (omitting some checks) with PMDK `libpemeobj`. In the structure definition, a persistent pointer is used by invoking `POBJ_ENTRY` (Line 1). In the insertion function, the code creates a PMOP by invoking `pmemobj_create` if it is the first use (Line 3). This function specifies a piece of space for allocating node objects. A root node is created through `POBJ_ROOT()` and the return value is assigned to *typed node object* `root` that is created in `TOID()` (Line 4).

Then an insertion transaction is defined for this PMOP. It uses `TX_NEW` to allocate a node object newnode(Line 6). After that, the input `value` is written to this object through `D_RW()`. `D_RW()` converts a persistent pointer into a readable and writable normal pointer by adding the base address of this PMOP and the offset (Line 7). If the root is not NULL (Line 8), the program inserts this newnode to the head. This branch first annotates the logging of the entire newnode through `TX_ADD_DIRECT()`, then it assigns the persistent pointer of `root` to the `next` field in newnode.

2.3 Garbage Collection

2.3.1 Mark-and-Compact Garbage Collection (MCGC). MCGC [41] can fix both memory leaks and memory fragmentation. An MCGC involves three steps, *marking*, *summary*, and *compacting*, as illustrated in Figure 4. Initially, the program has some roots, and suppose that allocated objects occupy two memory pages. In the *marking* phase, GC traverses all objects from roots to identify all reachable objects. In the summary phase, GC summarizes the fragmentation state and identifies *relocation pages* (i.e., pages that need to be compacted) and *destination pages* where objects in relocation pages will move to. The data structure storing destination information is called the *forwarding table*. Unreachable objects on other pages are returned to free lists. After that, the *compacting* phase moves all reachable objects in relocation to destination pages and updates references to them. Then the second page is released.

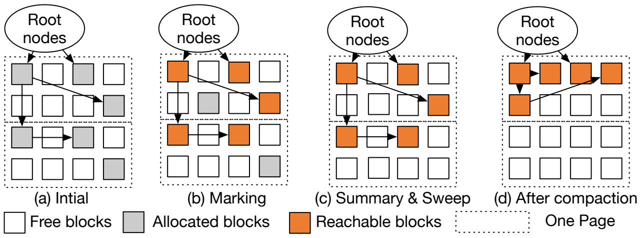


Figure 4: Illustration of mark-and-compact garbage collection.

2.3.2 Concurrent Garbage Collection. Based on whether the application pauses from GCs, GCs can either be *stop-the-world* (stalling all application threads while the collection is performed) or concurrent (without stalling the application).

Stop-the-world (STW) GCs are popular as they are easy to implement. They, however, cause high tail latency for real-world applications. For example, for a 16GB heap, the concurrent ZGC only incurs at most 1ms application pause while stop-the-world G1 GC incurs an average 250ms pause [64]. Growing heap size can even lead to pauses of over a minute [42]. In comparison, data centers often have deadlines of around one ms [11, 26, 57].

Concurrent GCs hence receive more interest [22, 63]. However, to ensure a consistent view of memory, concurrent GCs rely on *barriers*, which are small pieces of code that are added to every reference/pointer operation by the compiler in managed languages.

GC barriers are different from synchronization barriers in parallel programs, or memory barriers in memory consistency models.

In concurrent GCs, barriers address two inconsistent conditions. First, the application may modify references after the GC marking step has already visited them, thereby hiding references and freeing reachable objects. The solution is to add a *write barrier* on all reference write that will add the new value into the marking queue. Second, the application may access a moved object using an old reference. This problem is avoided by using *read barriers*: When reading a reference, the barrier code checks whether the object pointed by this reference has moved or not, and if it has, the barrier code looks up the destination address and updates this reference. If it has not, this object will then be moved (i.e., copied to the destination), and this reference is updated.

The marking and summary steps are *idempotent*, i.e., repeated invocations yield the same output and do not cause high overheads. However, the compacting step is not idempotent and incurs overhead on PM (See Section 3.2).

2.4 Prior PM libraries with GC

Prior work in PM for C/C++ explored potential methods to solve the persistent leak problem, but none of them addressed the persistent fragmentation problem. PMDK [33] requires programmers to manually implement memory leak detection and repair programs for each type to fix persistent memory leaks. NV-heap [18] and Corundum [30]a proposed reference counting to fix memory leaks, while naive reference counting cannot handle leaks with circular references and incurs large overhead to maintain all reference counters. Atlas [15] and Makalu [12] propose a lock-based memory allocator with post-crash garbage collection to reclaim memory leaks. The GCs in all of these works are conservative GC, fixing some leaks but not defragmenting.

There is some work in PM on managed languages that is relevant to GC, but none of them provide crash-consistent GC. Write-Rational GC [2] and Panthera [80] propose to use GC to move frequently written objects to DRAM and infrequently written objects to PM to save PM lifetime and DRAM energy. Auto-persist [74] and P-inspect [47] move data from DRAM to PM to automatically persistify objects in JAVA. They do not focus on the special challenges of crash-consistent GC. Yang *et. al.* propose to improve PM bandwidth utilization to improve GC efficiency [91].

3 APPROACH AND SOLUTION

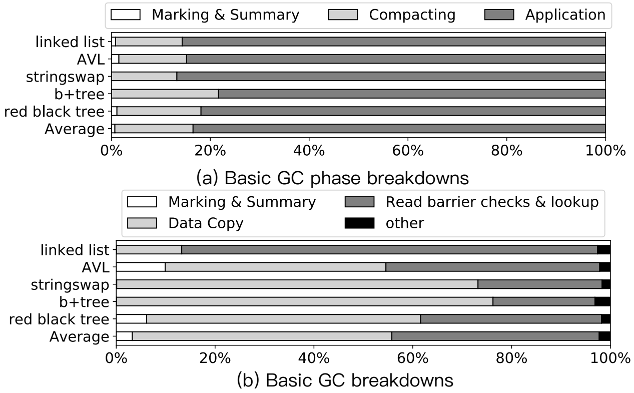
In discussing our approach, we start with discussing the rationale for building our fragmentation management over a GC (Section 3.1) and the efficiency in crash consistency GC on PM (Section 3.2). Then we discuss our FFCCD design, highlighting its key features (Section 3.3). We provide the details to achieve the design: architecture support for acceleration (Section 4) and software implementation (Section 5).

3.1 Basis for Building Fragmentation Management over GC

Building our fragmentation reduction over a compacting GC in unmanaged languages seems contrary to the conventional wisdom that GC is possible only for managed languages. The fundamental problem of using a GC in C language is that C programs may expose raw memory addresses to programmers, who may access objects in ways that prevent the runtime from distinguishing integers and pointers and hence updating all references correctly. We made several observations that the challenge is circumventable for PM. First, thanks to the need for PM to support future data reuse, object creators in PM need to specify the roots of main data structures (DS) for reusing, as required by existing PM programming models [15, 18]; the roots provide the starting point for the GC marking step. Second, for the same reason, in PM programming models [15, 18], the object creators record type information of all objects for future references, allowing us to distinguish data and references [33]. Third, the use of offset-based persistent pointers in PMOP [18, 33, 79, 81] requires the program to use special APIs to convert persistent pointers into virtual addresses on every read/write. This creates an opportunity to repurpose the APIs as read barriers to support concurrent GC in C/C++ programs. These observations and properties lay the basis for GC to work in PM with C/C++: it can detect all reachable objects, distinguish pointers from integers, reclaim leaks, and update all references if objects are moved. This work is based on the programming models like `libpmemobj` and `libpmemobj++`, which employ typed allocation, typed persistent pointers, and root nodes. Some libraries in PMDK [33] do not follow the above programming model, like `libpmem`. They do not even support the manual fix of persistent memory leaks.

3.2 Crash Consistency and Performance Bottlenecks

Recall that our fragmentation reduction approach requirements include crash consistency and low-performance overheads. Figure 5(a) shows the overheads of the basic GC from Espresso [84] design, originally proposed for JAVA, when we adapted it for C/C++, with crash consistency and concurrent GC. To adapt Espresso design for C/C++, the PMDK `D_RW` and `D_RO` functions (only `D_RW` is shown in the figure), required for dereferencing a persistent pointer, have read barrier code inserted to support the compacting phase of GC.



(a) Basic GC phase breakdowns

(b) Basic GC breakdowns

Figure 5: Baseline GC overhead breakdown.

The figure shows that Espresso slows down PM programs substantially, by 16.5% on average, representing 22.1% overheads over the application alone. Furthermore, the compacting phase occupies most of the GC time (See Section 6 for more details). Figure 5(b) shows the GC time breakdowns. The source of the overheads is primarily the crash-consistent memcpy() of data from the relocation page to the destination page, followed by the read barrier check and lookup. The former includes additional persist barrier instructions, stalls due to persist barriers on each GC-caused PM write, high PM write latency, and write amplification that pressures the PM performance given that PM has a significantly lower write bandwidth than DRAM (20% of DRAM write).

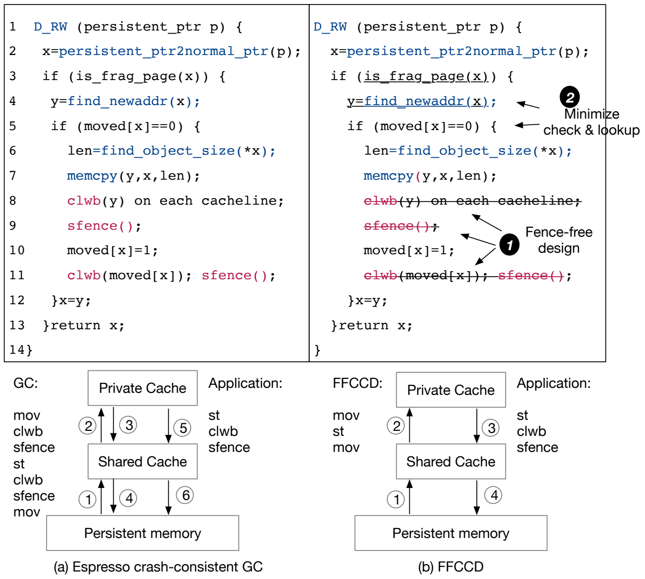
3.3 FFCCD Overview

We propose a fence-free crash-consistent concurrent defragmentation (FFCCD) to address the challenges of introducing crash-consistent defragmentation on PM with low overhead. FFCCD is built on a standard Mark-and-Compact Garbage Collection (MCGC). We review MCGC steps to enhance it with crash consistency. By analyzing Espresso [84], a crash-consistent GC on JAVA, we propose a fence-free design.

3.3.1 Review MCGC. MCGC has three phases: *marking*, *summary*, and *compacting*. In the marking phase, MCGC maintains a bitmap to mark all objects reachable from roots. The summary phase summarizes the fragmentation state and selects relocation pages and destination pages for all reachable objects in the relocation pages. The destination information is stored in the forwarding table. Analyzing them, we note that both marking and summary phases are *idempotent*, as these two phases only collect information from application memory and do not modify it. The final phase, compacting, copies objects from the relocation pages to destinations looked up from the forwarding table. Afterward, as a reference to a moved object is read, a read barrier will be triggered, updating the reference with the new destination address.

The default compacting phase may lead to several inconsistent application states for PM: (1) Inconsistent forwarding table. The forwarding table stores object virtual destination addresses. However, cross-run a PMOP may be mapped to a different virtual address range; hence the destination addresses are invalid, and compacting cannot safely resume. (2) A partially moved object. At a read barrier, PM copies an object, updates the object movement state, and updates a reference to the object. These stores may reach PM in an order different from the program order. Partially or out-of-order updates may lead to application data inconsistency or data loss.

3.3.2 State-of-the-art Crash-consistent GC Design. Espresso [84] proposes a crash-consistent CG design in JAVA. Figure 6 (a) shows an adaptation of their design if they had been implemented in C/C++. The PMDK D_RW and D_RO functions (only D_RW is shown in the figure), required for dereferencing a persistent pointer, has read barrier code inserted to support the compacting phase of GC. It includes distinguishing normal references and references to relocation pages (line 3), finding the destination address from the forwarding table (line 4), checking the movement state of the object pointed by this reference (line 5), memcpy() this object to the destination address (lines 7-9), updating movement state (lines 10-11), and updating the reference (line 13). Even though Espresso uses a per-object timestamp to indicate whether an object is relocated or not, this design is conceptionally the same as that in Figure 6 (a).



```
1  D_RW (persistent_ptr p) {
2   x=persistent_ptr2normal_ptr(p);
3   if (is_frag_page(x)) {
4    y=find_newaddr(x);
5    if (moved[x]==0) {
6      len=find_object_size(*x);
7      memcpy(y,x,len);
8      clwb(y) on each cacheline;
9      sfence();
10     moved[x]=1;
11     clwb(moved[x]); sfence();
12   }x=y;
13  }return x;
14}
```

```
D_RW (persistent_ptr p) {
  x=persistent_ptr2normal_ptr(p);
  if (is_frag_page(x)) {
    y=find_newaddr(x);
    if (moved[x]==0) {
      len=find_object_size(*x);
      memcpy(y,x,len);
      ~~clwb(y) on each cacheline;~~
      ~~sfence()~~;
      moved[x]=1;
      ~~clwb(moved[x]); sfence()~~;
    }x=y;
  }return x;
}
```

(a) Espresso crash-consistent GC (b) FFCCD

Figure 6: Espresso [84] GC Design (a) and FFCCD Design (b).

To provide crash-consistent GC, Espresso proposes adding several *clwb*s and two *sfence*s into the read barrier. The inserted *clwb*s and a *sfence* are necessary between memcpy() and the movement state update (lines 7-10) to ensure the object move has persisted prior to updating its movement state. Suppose the movement state update is persisted before the entire object was copied, and a crash happens. During recovery, GC will find that the movement state has been updated, incorrectly assume that the object has finished moving, and terminate recovery of this object, leading to application data inconsistency. The *clwb* and *sfence* are also necessary between the movement state update and reference update (lines 10-13). Suppose the reference is persisted before the movement state update is persisted. In that case, the application may already use the new reference to write new content in this object at the destination address when a crash occurs. During recovery, GC may find a stale movement state and incorrectly assume that the object was not copied. Hence, it may recover by copying the old value at the old address to the destination address, overwriting the new data, leading to inconsistent application data. The reference update is idempotent as long as this object's memcpy() and the state update are persistent. The reference will always be redirected to the latest value address with the read barrier and crash-consistent forwarding table.

3.3.3 Single-fence and Fence-free Crash-consistent Concurrent Defragmentation. In the Espresso design, there are several performance problems in its read barrier. (i) There is an explicit check on whether a pointer is to an object on a relocation page. (ii) If so, its new address needs to be attained by checking a large table in memory, with poor locality. (iii) Two writes that are critical to the crash consistency requirement (the memcpy() and the movement state update) both incur a round trip to load data from PM and store data to PM. (iv) The two writes require two persist barriers (i.e., two pairs of *clwb* and *sfence* instructions) and their latency.

To reduce the overheads stated above, we propose two solutions. In the first solution, we propose to reduce the fourth overhead source by removing one (i.e., 50%) of the persist barriers. This can be achieved at the cost of slight complexity increase in the crash recovery protocol. We refer to this as single fence crash-consistent concurrent defragmentation (SFCCD). SFCCD does not require hardware support. However, with hardware support, we can remove or reduce all four sources of overheads. We refer to the latter as fence-free crash-consistent (FFCCD). FFCCD removes all persist barriers (Figure 6(b)), offloads the object movement to a copying functional unit, and provides hardware caching for looking up the forwarding table. We create two designs for FFCCD. In both, we make the forwarding table produced by the summary phase persistent before the compaction phase continues its execution. During compaction, we insert a read barrier into PMDK `D_RW` and `D_RO`. We have the following observations that lead to our two designs.

Observation 1: Memcpy() for an object is idempotent as long as the object is clean. For an object that has not been modified, its value in the original location is clean and consistent with that in PM. Hence, upon a crash, the data in the original location enables the GC to redo memcpy() for the object after a crash.

Removing the *sfence* from line 9 of Figure 6(a) means that it is possible for the movement state update to persist prior to the copied object being persisted. Espresso's approach uses a *sfence* to avoid that. However, we make the following observation that inconsistency that arises from the lack of *sfence* is something that crash recovery can handle.

Observation 2: During recovery after a crash, inconsistency in any copied objects due to the unflushed copies can be fixed by inspecting the destination value. During recovery, we can inspect the movement state and object values in the destination address. (1) For objects whose movement state is "moved", we inspect this object's content in the destination address. If the content is different from the relocation address value, it means this object's memcpy() did not persist (fully). Then, SFCCD will repeat memcpy() for cachelines belonging to this object and persist them. For objects whose movement state is "not moved", the memcpy() can be redone after SFCCD resumes execution.

```
D_RW (persistent_ptr p) {
 x=persistent_ptr2normal_ptr(p);
 if (is_frag_page(x)) {
  y=find_newaddr(x);
  if (moved[x]==0) {
    len=find_object_size(*x);
    memcpy(y,x,len);
    clwb(y) on each cacheline;
    moved[x]=1;
    clwb(moved[x]); sfence();
  } x=y;
 }return x;
}
```

(a) Single-fence CCD read barrier

```
Recovery () {
  for each object A in relocation page
    if(moved[&A]==1) {
      y=find_newaddr(&A);
        if (A!=*y) {
          len=find_object_size(*x);
          memcpy(y,x,len);
        }
    }
}
```

(b) Single-fence CCD recovery

Figure 7: SFCCD read barrier (a) and recovery code (b).

Based on the above observations, Figure 7 shows the SFCCD read barrier and recovery pseudo code. This design only needs one *sfence* (a 50% reduction) to improve the GC read barrier at the expense of a little more work at the recovery. The remaining *sfence* ensures that when execution exits from the read barrier, an object that has moved have also updated its movement state. This is a justified trade-off because crashes are rare in many systems.

Now we analyze the possibility that we go further by removing all *sfence*s and *clwb*s from the read barrier.

Observation 3: The reference update is idempotent, and it can be repeated if memcpy() data persisted or be undone if memcpy() data did not persist. The key insight here is that post-crash recovery can establish consistency between the reference update and object movement via alternative mechanisms. The reference update can be undone depending on whether memcpy() data persisted at the destination address or not. (1) If the memcpy data persisted, it means the reference update was correct, and the FFCCD recovery function does not need to take special action. If, for some reason, the reference update did not persist, then the reference update is lost. However, this is not a problem because the next time the reference is accessed again, the reference will then be updated when the read barrier is invoked, and the crash-consistent forwarding table will provide the destination address for the update. (2) If the memcpy data is not persisted at all, FFCCD recovery can undo the reference update by looking up the forwarding table. This ensures consistency between the object at the new location and the reference will be eventually consistent (in the former case) or undone to a consistent state (in the latter case).

Our last observation concerns a larger object spanning more than one cacheline, for which memcpy() partially persisted, i.e., some cachelines persisted, but others did not.

Observation 4: If at least one cacheline of a memcpy() object persisted, we need to create a consistent state by finishing this object movement and movement state update. When we remove all *sfence*s and *clwb*s, if one cacheline of a memcpy() object persisted, it could contain the old value in relocation pages or the new value overwritten by the application. During recovery, FFCCD cannot distinguish these cases. Therefore, we need to resume finishing the object's memcpy() for other cachelines that did not persist, and update its movement state. Otherwise, FFCCD may use the old value to overwrite the application's new value, leading to an inconsistent state. Meanwhile, if the application writes to an object that is all lost, the application has the mechanism to redo it, and GC does not need to handle it.

Based on these observations, to provide FFCCD, we need a mechanism to know whether a cacheline has reached the persistence domain by introducing architecture support (Section 4). With such support, FFCCD can remove all *clwb*s and *sfence*s in the read barrier and provide crash consistency at recovery. We refer to a cacheline of a moved object that has reached the persistent domain as *reached.* During recovery, by inspecting reached cacheline of memcpy(), FFCCD can discern two object states (not reached and partially reached). For an object that is partially moved, FFCCD repeats memcpy() for it and updates its movement state as "moved". For a not yet moved object, FFCCD undoes reference updates for it. Then, FFCCD resumes application recovery. Finally, FFCCD resumes application execution and FFCCD execution.

Our fence-free design has the following benefits: (1) The read barrier does not need to execute any *clwb* or *sfence* instructions to support crash consistency. This completely avoids full PM latency exposure that would have been there with sfence. (2) The memcpy() is not followed by cacheline flush/writeback; hence we incur fewer PM writes (good for performance and write endurance) while the cacheline remains available in the cache for future reuse (good for cache locality). Objects at the new location and their movement state reach the PM lazily, either from natural cache eviction, or if the application itself (not GC) flushes them to PM.

4 ARCHITECTURE DESIGN

4.1 Overview

This section gives an overview of the architecture supports. Fig. 8 summarizes the varying architecture supports needed by three different designs: SFCCD (no architecture support), FFCCD (lightweight architecture support for relocate), and FFCCD with checklookup architecture support.

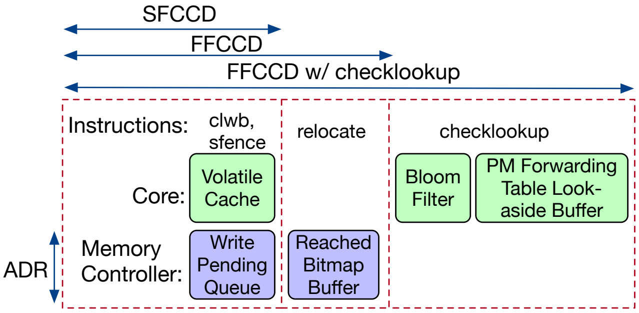
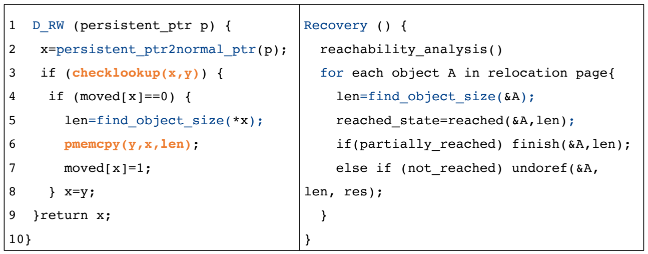


Figure 8: SFCCD needs no architecture support other than the currently existing one. FFCCD can be supported with only lightweight architecture support for relocate (using RBB) or full architecture support for checklookup (using bloom filter and lookaside buffer).

The relocate instruction performs copying data from a source address Rs with length specified in Rl to a destination address Rd. A similar instruction has been proposed in the literature for bulk copying [38]. However, our instruction has a major difference in that it not only performs copying, but it also tracks the persistence status of each destination cachelines, which is important for crash recovery. The instruction may be implemented by the addition of a special functional unit that performs the actual copying with finite state machine tracking or is expanded into a loop of loads and stores when it is decoded. Furthermore, for practical consideration, the instruction may require at most one page involved in the source and another in the destination. If an object spans multiple pages, the copying may be broken into several relocate instructions. This implementation is wrapped in a *pmemcpy()* API function.

The checklookup instruction checks whether a reference in register Rs points to relocation object, and if so, looks up the forwarding table to retrieve the new address it should move to and place this address in register Rd.

With the architecture support, the read barrier and recovery of FFCCD are shown in Figure 9. We replace memcpy() with pmemcpy(), where the *relocate (y, x)* instruction(s) replace the *mov* instruction(s) issued by memcpy() in the read barrier. The *checklookup* instruction replaces the check and forwarding table lookup. During recovery, the recovery function runs the reachability analysis to support undoing reference updates. Then, the function checks whether each object is partially reached or not reached. It finishes memcpy() and updates movement states of reached objects, and undoes reference updates of not reached objects. The recovery function itself uses a more conservative approach, with persist barriers and logging to ensure the recovery function itself is easy to recover.



```
D_RW (persistent_ptr p) {
 x=persistent_ptr2normal_ptr(p);
 if (checklookup(x,y)) {
  if (moved[x]==0) {
    len=find_object_size(*x);
    pmemcpy(y,x,len);
    moved[x]=1;
  } x=y;
}return x;
}
```

(a) FFCCD read barrier

```
Recovery () {
  reachability_analysis()
  for each object A in relocation page{
    len=find_object_size(&A);
    reached_state=reached(&A,len);
    if(partially_reached) finish(&A,len);
    else if (not_reached) undoref(&A,
len, res);
  }
}
```

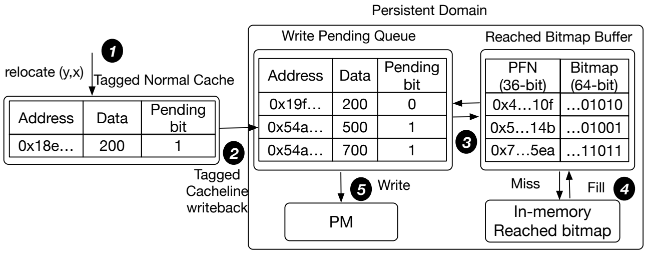
(b) FFCCD recovery

Figure 9: FFCCD read barrier (a) and recovery code (b).

4.2 Fence-Free Design

Figure 10 illustrates the design and mechanism of FFCCD. The new instruction *relocate (y, x)* performs the same function as *mov (y,x)* instruction except for setting a single-bit corresponding to a destination cacheline. This bit is a part of the data structure called *reached bitmap* kept in the main memory. The structure is created at the beginning of the compacting phase of the GC cycle and is deallocated at the end of the compacting phase. The reached bitmap is used to aid crash recovery. In order to avoid reading and updating the structure in memory too frequently, we add a hardware cache called Reached Bitmap Buffer (RBB) to the memory controller. Each RBB entry corresponds to a single destination page (represented by a 36-bit physical frame number (PFN)). Each entry contains a 64-bit bitmap representing 64 cachelines in the destination page; a value of 1 indicates the cacheline has reached persistence.

Each cacheline and the Write Pending Queue (WPQ) are augmented with one pending bit. This pending bit temporarily tracks a cacheline involved as the destination of a *relocate* instruction while the cacheline is in the cache hierarchy. Whenever the cacheline has reached the persistence domain, which is the memory controller, the pending bit is used to update the RBB.



Address	Data	Pending bit
0x18e…	200	1

Address	Data	Pending bit
0x19f…	200	0
0x54a…	500	1
0x54a…	700	1

PFN (36-bit)	Bitmap (64-bit)
0x4…10f	…01010
0x5…14b	…01001
0x7…5ea	…11011

Figure 10: FFCCD design.

To illustrate the full flow, suppose that the *relocate (y, x)* instruction moves an object that fits a cacheline from address *x* to address y. The instruction execution sets the pending bit of address y 1 ❶. The pending bit value is carried into WPQ when a cacheline is written back ❷. For the cacheline whose pending bit is 1, this cacheline uses its address to compare against PFN in RBB and set the corresponding cacheline bit in the matched RBB entry ❸ as 1, indicating this cacheline has reached the persistence domain. A miss in RBB will fetch the corresponding entry from memory ❹. After setting the bit in the bitmap, this cacheline clears its pending bit to 0. All cache lines whose pending bits are 0 perform normal writes to PM ❺.

After power off, the content in RBB will be flushed into PM, and the logic between WPQ and RBB will also be performed during flushing. At the recovery, FFCCD uses the reached bitmap to determine whether an object is not moved or partially moved and then performs recovery as described in Section 3.3.3. The detailed algorithm is shown in Figure 9.

4.3 Hardware Checks and Lookup

This subsection introduces our PM-aware forwarding table design and architecture support to minimize check and lookup overheads.

4.3.1 PM-aware Forwarding Table. We design the PM-aware forwarding table (PMFT) to provide crash consistency and lookup efficiency. The forwarding table is used to record the destination address of all reachable objects in relocation pages. Directly using virtual addresses (VA) in the forwarding table fails to meet crash consistency for PM because PMOP will be mapped to different VA ranges before and after a crash. In addition, if the forwarding table includes object size and type to construct a more compact one (hashed forwarding table), it saves some space, but it is not suitable for hardware acceleration due to irregular access.

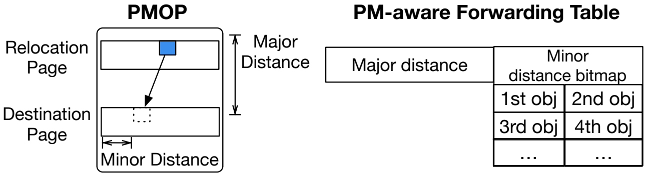


Figure 11: PM-aware forwarding table.

We design the PMFT by leveraging the fact that PM pointers introduce another layer of redirection. PMFT is a software data structure that incorporates this redirection to provide crash consistency and space efficiency. Figure 11 shows PMFT design. Each PMFT entry contains a tag that indicates the relocation page, and a part of the bits in the relocation page is used to index the PMFT. For an object that needs to be relocated from a relocation page to a destination page, each page start address has an offset within this PMOP. We regard the offset of the target page as the major distance. On the destination page, the start address that this cacheline should be placed is an offset between the start address of this page and the start address of this object. We regard this as a minor distance. The minor distance uses the alignment size (16-byte in the current glibc [25]) as the minimal granularity. Each cacheline can obtain its new offset within this PMOP by adding the major distance of its page and the minor distance of this cacheline. To locate the minor distance in the target page of each cacheline, we use the relocation page minor distance as the index and the minor distance of the target page as the value to construct a Minor Distant Map. Each minor distant entry is 1 byte for 256 16-byte possible addresses, and there are at most 256 entries for one page. We assume the PMOP size is up to 1 GB, then the size of the first two fields in PMFT is 18-bit. Each 4-KB fragmented page needs 259 bytes to represent the entire forwarding information.

To enable the GC to continue execution after crashes, PMFT needs to calculate all object destinations before the compacting phase to provide deterministic relocation. It means that whatever an object relocation is performed by any component (the read barrier or the recovery function), relocating an object will always have the same outcome (destination) regardless of whether it is performed before or after a crash. Compared to ZGC forwarding table [63], which calculates all object destinations on the fly, its forwarding table memory overhead percentage over the relocation page size is 3.2%. However, ZGC cannot resume execution due to non-deterministic relocation. Our PMFT provides deterministic relocation with an acceptable 6.3% of the relocation page size.

We will still use 4KB as the granularity to store forwarding information for space efficiency for huge pages. For example, if we use 2MB as the granularity, the major distance only needs 9 bits. However, the minor distance needs 17 bits to represent 131, 072 possible destination addresses, incurring 13.3% of the relocation page size.

4.3.2 Architecture Support. The check and lookup is the second-largest bottleneck in the MCGC. In particular, indexing the PMFT, and reading it from memory, incur significant overheads. To minimize check and lookup overhead, we propose to use a bloom filter and PM-aware forwarding buffer to accelerate these two steps. A new instruction, *checklookup*, is introduced to access these two hardware components and replace two functions in lines 3-4 in Figure 6.

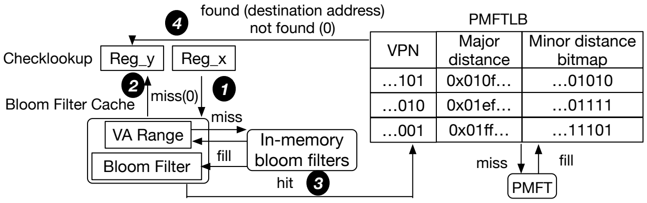


Figure 12: Checklookup design and logic.

Figure 10 illustrates the design and logic of *checklookup* instruction. A Bloom Filter Cache (BFC) is introduced to store relocation pages' Virtual Page Number and the virtual address (VA) range in this bloom filter. Several in-memory bloom filters are constructed to record all relocation pages during the summary phase. Furthermore, a PMFT look-aside buffer (PMFTLB) is added to cache PMFT information. Each entry has a 36-bit VPN, 18-bit major distance, and a 256-byte minor distance bitmap.

The *checklookup* instruction takes the relocation object's VA (stored in *Reg_x*), performs a check, and conditionally looks up the destination address of this object and stores it in *Reg_y*. The *checklookup* instruction first uses the object's VA to check against the VA range in the BFC. If an object's VA does not belong to this VA range, BFC will fetch the corresponding bloom filter into memory ❶. Then, the object's VA is checked against the bloom filter. If BFC returns a miss, the *Reg_y* becomes 0 and the *checklookup* instruction finishes ❷. If BFC returns a hit, the object's VA is further be used to lookup PMFTLB ❸. The destination address obtained from the matched PMFTLB entry is stored in the *Reg_y*. If it is not found in PMFTLB and PMFT, 0 is returned to *Reg_y*.

False positives of the bloom filter will not influence the correctness. Because if a non-compacting page goes to our designed scheme, it will encounter a not found through *checklookup* PMFT entries. Then, no further action is needed. This store will proceed as a normal PM store. The load request has a similar logic to handle false positives.

4.4 Hardware cost

The hardware cost of our design is shown in Table 1. Our FFCCD design adds cache and the write pending queue with a 1-bit tag. We introduce a reached bitmap buffer in the memory controller, a bloom filter, and a PMFTLB in the processor. PMFT and reached bitmap and in-memory bloom filter are in-memory software data structures. We use Cacti [78] to evaluate the die area needs with the 45nm process Nehalem processor [82]. The total on-chip storage introduced is 2256 bytes and consumes only 0.1% die area. The in-memory space linearly grows with relocation page size. The memory space overhead percentage over relocation page size is only 6.52%.

Table 1: Hardware cost

New on-chip Components	Entry size (bytes)	# of entries	Size (bytes)	Area (mm^2)
Reached bitmap buffer	12.5	8	100	0.004
PMFTLB	70.75	16	1132	0.045
Bloom Filter Cache	N/A	N/A	1024	0.041
In-memory persistent space (bytes)	Entry size per 4 KB page (bytes)	Memory overhead percentage over the relocation page size (%)		
PMFT	259	6.32		
Reached bitmap	8	0.2		

ADR requires WPQ logic, battery, failure mode, and the capability to detect power loss signal [70]. WPQ logic has the Detecting Platform Capabilities to detect power failure and trigger flushing WPQ content to memory. ADR provides sufficient power to the WPQ, persistent memory, and the memory controller to flush the WPQ content. Adding RBB requires some increase to the battery capacity or capacitor to support additional reads and writes, but the WPQ logic, controller, and platform capabilities can be shared and reused. Based on PM performance [35], eight 64-byte reads and eight 64-byte writes conservatively need 0.3us. Assuming a Lithin technique [67], with energy density of 10^{-2} Wh per mm^3 [85], RBB needs an additional 0.017mm^3 volume. In contrast, eADR needs 300mm^3 to flush all level caches [3].

4.5 Soundness

Our designs only modify crash-consistency-related operations in concurrent GC. Regardless of crash consistency, concurrent GCs work correctly with read barriers in data-race-free programs. If two threads are accessing the same object and at least one is write, the programmer should put this access into a critical section. The GC-caused relocation from D_RW() is within this critical section, and hence only one thread can access this object and perform relocation. Data relocation, new writes to relocated objects, and GC metadata are all visible to other threads after the critical section.

We have demonstrated that our designs provide GC crash consistency and recoverability for each thread. If the multi-threaded program is data-race-free, crash consistency and recoverability are guaranteed in multi-threaded programs. We evaluate the crash-consistency correctness of GC and program data in single and multi-threaded programs in Section 7.1.

5 SOFTWARE IMPLEMENTATION

We add three new interfaces and modify four existing interfaces for applications to use Espresso, SFCCD, or FFCCD. We introduce (1) `init()` to initialize fragmentation metadata and start monitoring fragmentation state to trigger defragmentation. Users deliver the defragmentation setting, including the fragmentation ratio to trigger defragmentation and the fragmentation target ratio in the summary phase. (2) `exit()` notify the defragmentation component to finish on-going defragmentation, release all related metadata and terminate defragmentation; (3) `recovery()` to recover inconsistent state of defragmentation, as described in Section 4.1 and Section 3.3.

We modify four existing interfaces: `D_RW()` and `D_RO()` modifications are shown in Section 4.1 and Section 3.3. We further modify `pmalloc()` and `pfree()` to record fragmentation state (e.g., fragmentation ratio) and trigger defragmentation when the PMOP fragmentation ratio is high than the threshold. Meanwhile, `pmalloc()` and `pfree()` periodically check reached bitmap in FFCCD and moved bitmap in Espresso and SFCCD to release the that have finished relocation of all objects in this page. If there is no change between two checks, defragmentation runs reachability again to finish all pending relocation and reference updates, and release relocation pages.

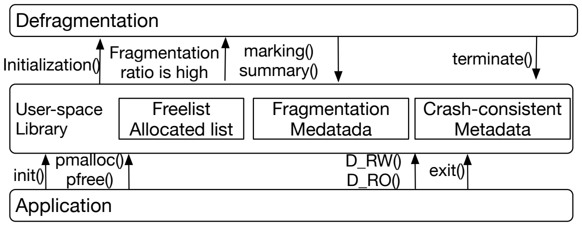


Figure 13: The workflow of FFCCD.

The defragmentation implementation mainly includes four functions, `marking()`, `summary()`, `terminate()` and `recovery()`. (1) The `marking()` runs the stop-the-world reachability analysis from roots to mark reachable objects. (2) The `summary()` calculate fragmentation ratio of each page and sort them according to the fragmentation ratio. Then, according to the defragmentation target ratio, this function selects *k* pages whose fragmentation is top *k*, such that compacting these pages into new pages will reduce the overall fragmentation ratio as the target ratio. After, this function calculates the forwarding table and persists it. The unreachable objects are returned to the freelist. (3) The `terminate()` is issued by `exit()`. It finishes all pending relocation and reference updates, and releases relocation pages. After that, it releases defragmentation-related metadata and terminates defragmentation. (4) The `recovery()` implementation is introduced in `pmalloc()` and `pfree()`. The workflow of using defragmentation in the application and their interactions are shown in Figure 13.

6 EVALUATION METHODOLOGY

SFCCD and FFCCD evaluation: We use a dynamic trace-driven simulation to evaluate Espresso on C/C++, SFCCD, and FFCCD. We first implement Espresso, SFCCD, and FFCCD with special instructions on C/C++. We execute programs with Espresso, SFCCD or FFCCD with 2MB huge pages on Sniper simulator [14], a cycle-level x86 simulator. Programs with Espresso and SFCCD are executed on top of the simulator without hardware changes. Programs with FFCCD are executed on top of the simulator with our FFCCD hardware design breakdowns. The extra memory accesses from the proposed architecture are included in the FFCCD evaluation by counting TLBs and memory/cache latencies. Table 2 shows our simulation parameters.

Table 2: Simulation parameters.

Processor	2.6 GHz, 5-way issue Out-of-order pipeline, 352-entry ROB, Intel x86-64 architecture
Cache	L1D cache 8-ways 48KB, 4-cycle access time; L2 cache: 16-ways 3MB, 25-cycle access time

Memory	DRAM latency: 120 cycles; PM latency: 360 cycles;
	WPQ latency: 30 cycles; Bandwidth: 24GB/s DRAM;
	12GB/s PM read; 4GB/s PM write

TLB	L1 data TLB: 4KB pages, 4-way, 64 entries;
	L1 data TLB: 2MB pages, 4-way, 32 entries;
	L2 4KB/2MB pages, 6-way, 1536 entries;
	1 cycle L1 TLB access, 4 cycles L2 TLB access;
	60 cycles 2MB TLB miss penalty

FFCCD	PFMTLB: 16 entries; RBB: 8 entries;
	Bloom filter size: 1024 bytes; In-memory bloom filter: 8;
	Bloom filter miss: 120 cycles; Bloom filter check: 2 cycles;
	PFMTLB latency: 4 cycles; RBB latency: 30 cycles;

Workloads: We evaluate two PM key-value store applications, Echo [59] and pmemkv [34]; two state-of-art concurrent data structures BzTree [5], and FPtree [65]; and five microbenchmarks (Linked List (LL), AVL Tree (AVL), String Swap (SS), B+tree (BT), Red-black tree (RBT)). Furthermore, we conduct a case study on Redis [55]. Similar to other fragmentation studies [50, 69], we initialize each benchmark (except Redis) with 5 million insertions with 128-byte values. Each application executes 4 million delete operations and 4 million insert operations separately to represent application memory increasing and decreasing stages. Each program will execute three phases: a deleting phase, an inserting phase, and another deleting phase. After that, each program terminates. We report the results of the above 3 phases.

Schemes: We execute the following four schemes. **Baseline**: Programs use the original PMDK 1.11.1 `libpmemobj` library to execute the workloads without defragmentation. **Espresso**, **SFCCD** and **FFCCD**: Programs use the modified PMDK `libpmemobj` library to execute the workloads.

Metrics: We report fragmentation reduction, which is calculated through equation 1.

$$Fragmentation\_reduction = \frac{Memory\_footprint\_reduction}{Memory\_footprint - Size\_of\_living\_data} \quad (1)$$

Parameters of defragmentation:

We evaluate defragmentation with two sets of parameters: (1) Normal defragmentation parameters in Redis [55], which trigger defragmentation when the fragmentation ratio is above 1.5, and the target one is 1.25. (2) Relaxed defragmentation parameters, which trigger defragmentation when the fragmentation ratio is higher than 1.7 and the target fragmentation ratio is 1.5.

7 EVALUATION

7.1 Correctness and Soundness

We use a fault-injection test used in previous PM crash-consistency validation studies [13, 31, 96] to validate the correctness of crash-consistency. We design the fault-injection test as follows. (1) We added a region in the program to simulate RBB. Each *relocate* instruction adds an entry into this region, and each *clwb* and *sfence* removes the corresponding entry if this region has. (2) We added a fault handler to the program. When a fault signal is received, the fault handler suspends program execution, flushes remaining entries in the region to the program PMOP, and terminates the program. We execute the defragmentation recovery on top of the intermediate program PMOP and the flushed content from WPQ and RBB.

We validate the consistency in two steps: program data and GC consistency validations. (1) We wrote a checker program to inspect whether the program data is consistent, including readability of all objects, absence of dangling pointers, and data structure topology. (2) The checker inspects the current relocation state of each object in the program and validates whether the state is the same as indicated in GC metadata. If an object was not moved, no reference should point to the new location, and the content in the new location must be the default value. The first consistency validation verifies that the program data is not corrupted by GC if there is a crash. The second consistency validation verifies that GC metadata matches the memory state and GC does not lead to inconsistency; hence it is safe to resume GC after restarting the program.

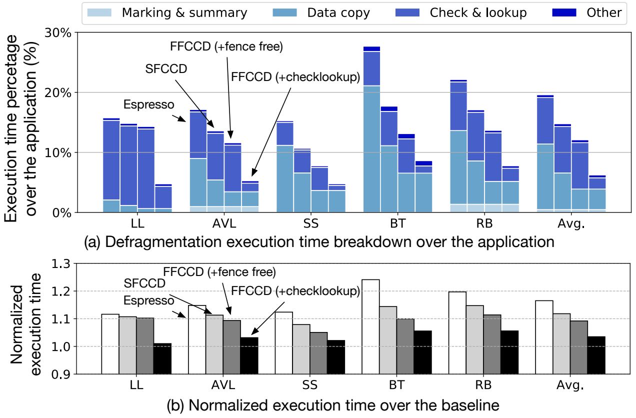
We execute the above fault-injection and post-crash validation for SFCCD and FFCCD on all microbenchmarks with one thread and concurrent PM data structure with 1, 2, 4, and 8 threads. We execute one thousand fault-injection tests for various settings (26 settings). We found that both GC schemes passed all the tests.

7.2 Results on Benchmarks

Table 3: Fragmentation effectiveness for various benchmarks

Prog.	Avg. Memory Usage on 2MB pages (MB)				Fragmentation Reduction (%)	
	PMDK	Actual	Ours (N)	Ours (R)	Ours (N)	Ours (R)
LL	245.6	145.3	194.6	225.5	50.8	20.0
AVL	369.3	230.1	309.5	334.6	43.0	24.9
SS	493.5	311.9	435.7	469.2	31.8	13.4
BT	845.7	536.1	735.3	817.6	35.7	9.1
RB	488.2	302.1	391.1	443.1	52.2	24.3
Avg.	488.5	305.1	413.2	458.0	42.7	18.3

The results of the benchmarks are shown in Table 3, where columns 2 to 5 are the average memory footprint of PMDK, actual live data, FFCCD with normal (N) defragmentation parameters, and FFCCD with relaxed (R) defragmentation parameters. Columns 6 and 7 are the average fragmentation reduction of FFCCD with two different parameter sets. The table shows that the memory footprint with PMDK is 60% larger than the live data size. FFCCD reduces the fragmentation by 42.7% and 18.3% during the execution. The fragmentation reduction of *B+tree* is small due to internal fragmentation, where one node can store 4 values.



(a) Defragmentation execution time breakdown over the application

(b) Normalized execution time over the baseline

Figure 14: (a) Defragmentation execution time breakdown (b) Normalized execution time of microbenchmarks.

Figure 14 (a) shows the execution time breakdown of the defragmentation only over the baseline where the application runs without GC, comparing the Espresso, our SFCCD, FFCCD with fence-free architecture, and FFCCD with all architecture support. Espresso incurs large performance overheads due to its slow read barriers and inefficient crash consistency management involving two pairs of *clwb*s and *sfence*s per barrier. SFCCD removes half of the persist barriers, reducing the overheads of data copy by 40%. FFCCD removes all barriers using the fence-free architecture support, reducing the overhead of data copy by 66%. Our check and lookup architecture support with bloom filter and PMFTLB further optimizes the read barrier, reducing the execution time of check and lookup nearly 80%. Overall, defragmentation time in FFCCD is reduced by 68% compared to that of Espresso.

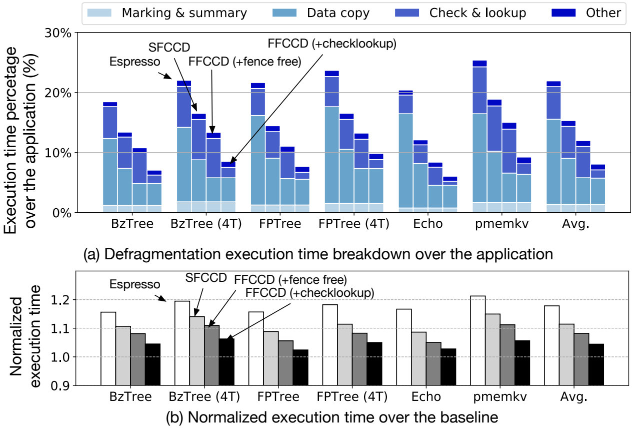
Note that the above figure does not account for the application running faster due to less fragmentation. The fragmentation causes more TLB entries and reduces cache locality, hence larger TLB misses and cache misses. Defragmentation can improve throughput by making data more compact. When the total execution time (application plus defragmentation) is compared to the same baseline, Figure 14 (b) shows that our best scheme, FFCCD, incurs a 3.5% execution time overhead over the application without defragmentation. The normalized execution time of these schemes over baseline (no defragmentation). This result indicates that with efficient defragmentation and FFCCD architecture support, defragmentation comes at low-performance overheads.

7.3 Concurrent PM Data Structure and Application Results

Table 4: Fragmentation effectiveness for various applications

DS & App.	Avg. Memory Usage on 2MB pages (MB)			Fragmentation Reduction (%)
	PMDK	Actual	Ours	Ours
BzTree	763.1	482.5	662.1	36.0
BzTree (4T)	785.6	495.2	679.5	36.5
FPTree	661.5	413.6	551.0	44.6
FPTree (4T)	675.4	422.7	564.3	44.0
Echo	711.7	450.5	638.1	28.2
pmemkv	653.2	399.1	535.3	46.4
Avg.	708.4	443.9	605.1	39.3

Table 4 shows the defragmentation effects on concurrent PM data structures and applications. BzTree and FPTree run with 4 threads (4T). FFCCD reduces fragmentation by 28.2–46.4%. The reduction on Echo is small because it uses a hash table and hence allocates memory with an array. This array cannot be released until all keys are removed. BzTree uses additional metadata to support PMwCAS operations, occupying more memory and receiving fewer benefits from defragmentation.



(a) Defragmentation execution time breakdown over the application

(b) Normalized execution time over the baseline

Figure 15: (a) Defragmentation execution time breakdown (b) Normalized execution time of applications.

Fig. 15 shows the overhead and normalized execution time on these four applications. Overall, SFCCD and FFCCD reduce nearly 40% and 70% overheads of data copy, and FFCCD only incurs 4.4% overhead over the application without defragmentation. BzTree has a lower overhead than FPTree due to less fragmentation in its implementation, as BzTree uses copy-on-write on internal nodes and append on leaf nodes, creating less fragmentation. Echo has fewer read barrier overhead due to a small number of references, but it also receives data copy overhead reduction from our designs.

7.4 Case study: Redis

Redis [55] is a typical PM application. It is an open-source real-world key-value store database. Redis is configured as an LRU cache with a maximum 200 MB of not-expired (living) objects. It generates 1 million random keys and values, with value sizes ranging from 240 bytes to 492 bytes. Then it performs queries on these keys. The benchmark then inserts 500K random keys and values, followed by queries until the end of execution. We use 4 KB pages in this evaluation. Mesh [69] is a recent C/C++ defragmentation work, which relocates one physical page's objects to another physical page if these two pages do not have page offset overlapped objects. After that, it changes the virtual page mapping of the first page to the destination page. The defragmentation of these two pages is finished without requiring reference updates.

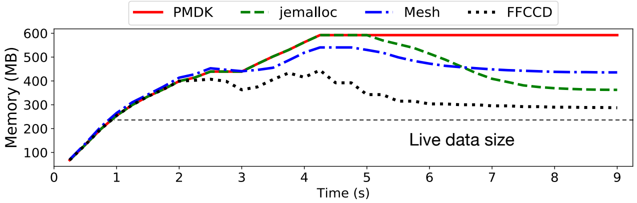


Figure 16: Memory footprints of different schemes.

Figure 16 shows Redis memory footprints for different schemes over time. Initially, the memory footprint for all schemes increases with insertions of key-value pairs until 200MB. Then, Redis expires key-value pairs with LRU policy by storing them into the disk to maintain 200MB of live data. The fragmentation increases during this process. With PMDK, fragmentation further increases after more record insertions, increasing memory usage and decreasing Redis throughput. STW defragmentation (jemalloc) will periodically introduce unacceptably long pauses.

FFCCD reduces fragmentation by 73.4% with only 4.6% overhead and low tail latencies. In contrast, STW jemalloc defragmentation only reduces fragmentation by 47.6%. Even worse are the 90th, 95th, and 99th tail latencies of STW jemalloc, which are 331ms, 442ms, and 563ms. They are more than one order of magnitude larger than those of FFCCD, which are 11.2ms, 22.1ms, and 34.8ms, respectively.

8 RELATED WORK

Garbage Collections. Beyond related work discussed in Section 2, other related work includes the following. Compacting GC has been a feature of managed languages (e.g., JAVA [28]) for a long time. G1 GC [62] is a partial concurrent GC with STW compacting. OpenJDK is developing ZGC [63], a fully concurrent GC using colored pointers. The JAVA reachability framework [47, 74] simplifies the programming of PM in JAVA, which reuses GCs in JAVA without considering crash-consistent GC. GCpersist [83] optimizes crash-consistent STW GC for PM, while STW GC incurs large application pauses and has fewer challenges in designing a crash-consistent and efficient one.

There has been work on hardware support for reference counting [39], which accelerates reference counting by introducing instructions. Maas *et al.* propose using FPGAs to accelerate the marking [56] step on JAVA. Charon [36] proposes in-memory computing to accelerate GC. Ye *et al.* work discusses how to preserve relocatability cross PMOPs in GC. In industry, commercial products focus on read/write barrier acceleration, e.g. IBM's Guarded Storage Facility [32] and Azul Vega [17, 77]. SI-TM [52] and SSP [61] explored how to use hardware to accelerate forwarding and address remapping steps.

Persistent Memory. There is a rich set of papers in literature supporting different aspects of persistent memory, including but not limited to, file systems [19, 20, 86, 87], physical organization [6, 7], persistency models [4, 19, 37, 48, 68, 73, 76, 95], logging [21, 46, 72], persistent memory debugging [54, 60].

Persistent memory security received more attention recently, as PM data is long-lived and corruption is also persistent. Several studies proposed to maintain crash consistency and improve the performance of encryption in PM [8, 9, 16, 53, 93, 94, 99]. Some work proposed faster Merkle tree mechanisms to verify the integrity of PM [10, 23, 24, 24, 27, 97, 98]. Another branch of work reduces the exposure window of PM to reduce the attack surface of PM corruptions [88–90], even more so as cross-process attacks are feasible [58].

9 CONCLUSION

In this paper, we discussed the persistent fragmentation problem in PM. We proposed to build defragmentation on top of GC to keep the PM working set size low. With the proposed software and architecture support, our PMGC offers the first-known PM solution for C/C++ applications, providing fence-free crash-consistent, concurrent, and efficient compacting GC. Results show that FFCCD reduces PMOP fragmentation by 28.2-73.4% on microbenchmarks and by 75.1% fragmentation on Redis, while incurring 4.1% execution time overhead on average.

ACKNOWLEDGMENTS

We thank our shepherd, Alaa Alameldeen, and all the anonymous reviewers for their insightful feedbacks to improve the final version of the paper. We also thank Derrick Greenspan, Martin Maas and Naveed Ul Mustafa for their valuable feedback and support. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-2124010, CNS-1717425, CNS-1900724, CNS-2106629, and and Office of Naval Research (ONR) under grant No. N00014-20-1-2750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[2] Shoaib Akram, Jennifer B Sartor, Kathryn S McKinley, and Lieven Eeckhout. 2018. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices* 53, 4 (2018), 62–77.

[3] 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)*. IEEE, 111–124.

[4] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique. In *Proc. of the International Symposium on Computer Architecture*.

[5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[6] A. Awad, S. Blagodurov, and Y. Solihin. 2015. Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems. In *Proc. of the International Symposium on Performance Analysis of Systems and Software*.

[7] A. Awad, S. Blagodurov, and Y. Solihin. 2016. Write-Aware Management of NVM-based Memory Extensions. In *Proc. of the International Conference on Supercomputing*.

[8] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. In *Proc. of the International Symposium on Architecture Support for Programming Language and Operating Systems*.

[9] A. Awad, Y. Wang, D. Shands, and Y. Solihin. 2017. ObfusMem: a Low-Overhead Access Obfuscation for Trusted Memories. In *Proc. of the International Symposium on Computer Architecture*.

[10] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. 104–115.

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[12] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. *ACM SIGPLAN Notices* 51, 10 (2016), 677–694.

[13] Koustubha Bhat, Erik Van Der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2021. FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 363–375.

[14] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

[15] 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.

[16] S. Chhabra and Y. Solihin. 2011. i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption. In *Proc. of the International Symposium on Computer Architecture*.

[17] Cliff Click, Gil Tene, and Michael Wolf. 2005. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. 46–56.

[18] 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.

[19] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.

[20] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*.

[21] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*.

[22] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–9.

[23] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2020. Persist-Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Non-Volatile Memory. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture*.

[24] Alexander Freij, Huiyang Zhou, and Yan Solihin. 2021. Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1227–1240.

[25] glibc Wiki. 2021. *Malloc Internals*. Retrieved February, 2021 from https://sourceware.org/glibc/wiki/MallocInternals

[26] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[27] Xijing Han, James Tuck, and Amro Awad. 2021. Dolos: Improving the Performance of Persistent Applications in ADR-Supported Secure Memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1241–1253.

[28] Wilfred J Hansen. 1969. Compact list representation: Definition, garbage collection, and system implementation. *Commun. ACM* 12, 9 (1969), 499–507.

[29] Swapnil Haria, Mark D Hill, and Michael M Swift. 2018. Devirtualizing memory in heterogeneous systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 637–650.

[30] Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/3445814.3446710

[31] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.

[32] IBM. 2017. *IBM's Guarded Storage Facility*. Retrieved February, 2021 from https://www.ibm.com/support/pages/java-sdk

[33] Intel. 2021. *Persistent Memory Programming*. Retrieved February, 2021 from http://pmem.io/

[34] Intel. 2021. *pmemkv 1.5.0*. Retrieved February, 2021 from https://github.com/pmem/pmemkv

[35] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[36] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. 2019. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 726–739.

[37] Jungi Jeong and Changhee Jung. 2021. PMEM-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–529.

[38] Xiaowei Jiang, Yan Solihin, Li Zhao, and Ravishankar Iyer. 2009. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 169–180. https://doi.org/10.1109/PACT.2009.31

[39] José A Joao, Onur Mutlu, and Yale N Patt. 2009. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the 36th annual international symposium on computer architecture*. 418–428.

[40] Mark S Johnstone and Paul R Wilson. 1998. The memory fragmentation problem: Solved? *ACM Sigplan Notices* 34, 3 (1998), 26–36.

[41] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

[42] E Kaczmarek and L Yi. 2015. Taming gc pauses for humongous java heaps in spark graph computing. *Spark Summit 2015* (2015).

[43] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 804–818.

[44] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R Ganger. 2018. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 691–704.

[45] Takayuki Kawahara, Riichiro Takemura, Katsuya Miura, Jun Hayakawa, Shoji Ikeda, Y Lee, Ryutaro Sasaki, Yasushi Goto, Kenchi Ito, Toshiyasu Meguro, et al. 2007. 2Mb spin-transfer torque RAM (SPRAM) with bit-by-bit bidirectional current write and parallelizing-direction current read. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 480–617.

[46] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. *ACM SIGPLAN Notices* 51, 4 (2016), 385–398.

[47] Apostolos Kokolis, Thomas Shull, Jian Huang, and Josep Torrellas. 2020. P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 509–524.

[48] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. *ACM SIGPLAN Notices* 51, 4 (2016), 399–411.

[49] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 256–267.

[50] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 705–721.

[51] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

[52] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. 2014. SI-TM: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 383–398.

[53] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 310–323.

[54] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 411–425.

[55] Redis Ltd. 2020. *Redis*. Retrieved February, 2021 from https://redis.io/

[56] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 138–151.

[57] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[58] Naveed Ul Mustafa, Yuanchao Xu, Xipeng Shen, and Yan Solihin. 2021. Seeds of SEED: New Security Challenges for Persistent Memory. In *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*.

[59] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 135–148.

[60] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. {AGAMOTTO}: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1047–1064.

[61] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L Miller. 2019. SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 836–848.

[62] Oracle. 2015. *Garbage-First Garbage Collector*. https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm#JSGCT-GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573

[63] Oracle. 2018. *Open JDK ZGC*. Retrieved February, 2021 from https://wiki.openjdk.java.net/display/zgc/Main

[64] Oracle. 2020. *ZGC Oracle Dev Live*. Technical Report. Online; accessed February, 2022.

[65] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[66] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 679–692.

[67] David Pech, Magali Brunet, Hugo Durou, Peihua Huang, Vadym Mochalin, Yury Gogotsi, Pierre-Louis Taberna, and Patrice Simon. 2010. Ultrahigh-power micrometre-sized supercapacitors based on onion-like carbon. *Nature nanotechnology* 5, 9 (2010), 651–654.

[68] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 265–276.

[69] Bobby Powers, David Tench, Emery D Berger, and Andrew McGregor. 2019. Mesh: compacting memory management for C/C++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 333–346.

[70] Steve Scargall. 2020. Persistent memory architecture. In *Programming Persistent Memory*. Springer, 11–30.

[71] John Schoenick. 2017. *ARE WE SLIM YET?* Technical Report. https://areweslimyet.com/faq.htm.

[72] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 178–190.

[73] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the long latency of persist barriers using speculative execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 175–186.

[74] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 316–332.

[75] Keith A Smith and Margo I Seltzer. 1997. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 203–213.

[76] Yan Solihin. 2019. Persistent Memory: Abstractions, Abstractions, and Abstractions. *IEEE Micro* 39, 1 (2019), 65–66.

[77] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*. 79–88.

[78] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. 2008. *CACTI 5.1*. Technical Report. Technical Report HPL-2008-20, HP Labs.

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

[80] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362.

[81] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. 2017. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 800–812.

[82] Wikichip. 2010. *Nehalem*. Retrieved February, 2021 from https://en.wikichip.org/wiki/intel/microarchitectures/nehalem_(clien%t)

[83] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. 2020. GCPersist: an efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 1–14.

[84] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 70–83.

[85] Zhong-Shuai Wu, Khaled Parvez, Xinliang Feng, and Klaus Müllen. 2013. Graphene-based in-plane micro-supercapacitors with high power and energy densities. *Nature communications* 4, 1 (2013), 1–8.

[86] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[87] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[88] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 987–1000.

[89] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. 2022. Temporal Exposure Reduction Protection for Persistent Memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 908–924.

[90] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 680–692.

[91] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. 2021. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 343–358.

[92] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Supporting legacy libraries on non-volatile memory: a user-transparent approach. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 443–455.

[93] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 33–44.

[94] Shougang Yuan, Yan Solihin, and Huiyang Zhou. 2021. PSSM: achieving secure memory for GPUs with partitioned and sectored security metadata. In *Proceedings of the ACM International Conference on Supercomputing*.

[95] Ardhi Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and fast lazy persistency on gpus. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

[96] Yiran Zhang, Fan Zhang, Bolin Yang, Guorui Xu, Bin Shao, Xinjie Zhao, and Kui Ren. 2019. Persistent fault injection in fpga via bram modification. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 1–6.

[97] Kazi Abu Zubair and Amro Awad. 2019. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. 157–168.

[98] Kazi Abu Zubair, Sudhanva Gurumurthi, Vilas Sridharan, and Amro Awad. 2021. Soteria: Towards Resilient Integrity-Protected and Encrypted Non-Volatile Memories. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1214–1226.

[99] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 479–492.