# Preserving Addressability upon GC-Triggered Data Movements on Non-Volatile Memory

CHENCHENG YE, Huazhong University of Science and Technology, China
YUANCHAO XU and XIPENG SHEN, North Carolina State University, USA
HAI JIN and XIAOFEI LIAO, Huazhong University of Science and Technology, China
YAN SOLIHIN, University of Central Florida, USA

This paper points out an important threat that application-level Garbage Collection (GC) creates to the use of non-volatile memory (NVM). Data movements incurred by GC may invalidate the pointers to objects on NVM, and hence harm the reusability of persistent data across executions. The paper proposes the concept of movement-oblivious addressing (MOA), and develops and compares three novel solutions to materialize the concept for solving the addressability problem. It evaluates the designs on five benchmarks and a real-world application. The results demonstrate the promise of the proposed solutions, especially hardware-supported Multi-Level GPointer, in addressing the problem in a space- and time- efficient manner.

**ACM Reference Format:**
Chencheng Ye, Yuanchao Xu, Xipeng Shen, Hai Jin, Xiaofei Liao, and Yan Solihin. 2022. Preserving Addressability upon GC-Triggered Data Movements on Non-Volatile Memory. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2022), 25 pages. https://doi.org/10.1145/3511706

## 1 INTRODUCTION

Byte-Addressable Non-Volatile Memory (NVM) bridges the gap between persistent storage and DRAM, by providing better performance over traditional storage and at the same time, data persistency over DRAM. Programmers can use NVM as an alternative to DRAM while enjoying the benefits of data persistence. The benefits are largely embodied by data reusability: Data can be reused across the executions of the same or different programs, without going through object serializations and deserializations that are needed in traditional persistent storage. To materialize the opportunity, object addressability must be maintained across executions and programs, such that when a program runs sometime later, it can find the objects on NVM that it is supposed to access.

This important property of persistent objects is, however, lost when application-level Garbage Collection (GC) is used. A garbage collector is an important part of a managed programming language (Java, C#, etc.); such languages are among the most popularly used languages[1]. When a program runs, the GC automatically manages the memory used by

[1] http://pypl.github.io/PYPL.html

Authors' addresses: Chencheng Ye, yecc@hust.edu.cn, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China; Yuanchao Xu, yxu47@ncsu.edu; Xipeng Shen, xshen5@ncsu.edu, North Carolina State University, USA; Hai Jin, hjin@hust.edu.cn; Xiaofei Liao, xfliao@hust.edu.cn, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China; Yan Solihin, Yan.Solihin@ucf.edu, University of Central Florida, USA.


Manuscript submitted to ACM

a program. It detects the dead objects, reclaims the memory allocated to them, and reduces memory fragmentation by gathering the free memory space together.

Such application-level GC causes special complexities to the addressability of objects on NVM. An NVM may consist of many memory regions (called NVRegion or NVPool) with each being a standalone chunk of NVM for mapping, unmapping, and inter-process sharing. One object may be pointed to by multiple objects in different NVRegions. As an NVM object may live beyond the lifetime of a program, some cross-region references to an object may be created in some earlier executions of some programs. When a program uses an NVM object, it may not be aware of all the pointers pointing to that object—some of those pointers may be on an NVRegion that this program may not even have the permission to access. As a result, when the GC thread in this program moves that object, the GC has no way to update all the pointers pointing to that object in existing designs. Those pointers would be corrupted.

***Example.*** Figure 1 illustrates the problem with a scenario involving an actual dataset and three applications, which each corresponds to a real-world software (assumed to have been modified to utilize NVM).

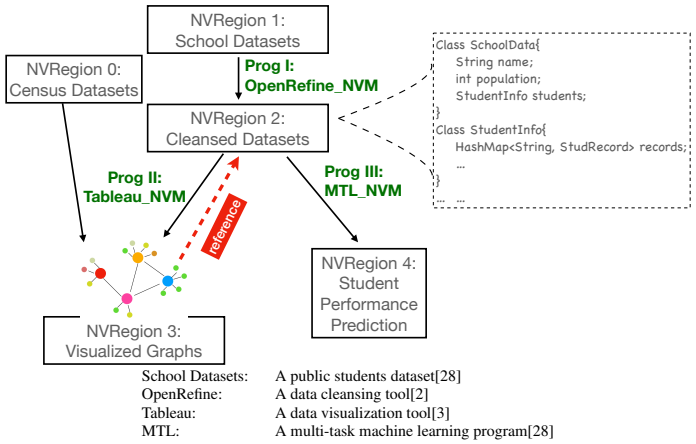



Fig. 1. An example illustrating GC-caused loss of data addressability on NVM. Each solid-lined box represents an NVRegion. When Program 3 runs, its GC packs and moves the student records in the cleansed dataset on NVM, which breaks the references from the visualized graphs to the student records in the cleansed dataset.

The first program, `OpenRefine_NVM`, takes NVRegion 1 which stores raw data and produces a new NVRegion, NVRegion 2, to hold the cleansed School Dataset in the form of Java objects. The data contains the basic information of the students, such as names and dates of birth. NVRegion 2 stores Java objects deserialized from the raw data to facilitate accesses to the datasets from Java programs.

The second program, `Tableau_NVM`, combines the cleansed data and some Census Data, such as the population of the whole schools, the relations among students, and the population of every gender, nationality, etc., to produce a new NVRegion, NVRegion 3, which holds the generated graphs that capture the correlations between income and student performance. The execution builds a reference into each graph node, linking it with the student records in NVRegion 2 such that users can view the detailed info by clicking the graph nodes. Meanwhile, Program I updates NVRegion 2 when new students are on board or when students transfer to other schools, causing creation, deconstruction, and updates of Java objects. The operations leave NVRegion 2 fragmented.

Later, Program III, `MTL_NVM`, runs to build up a student performance prediction model from the cleansed student datasets. However, as `MTL_NVM` allocates some data, the Java GC is automatically triggered. The GC packs and moves memory objects including the student records on NVRegion 2[2]. Because the view of `MTL_NVM` includes only NVRegion 2 and its own NVRegion 4, the GC does not update references in NVRegion 3, leaving them pointing to the obsolete locations of the records.

The example illustrates a phenomenon common in real-world computing systems, where persistent data (e.g., photos, contacts, logs, census data, etc.) are usually accessible by many applications, and these applications in turn generate derived data with cross references. The aforementioned reference corruption problem is not a concern on traditional file systems as persistent data are serialized and deserialized in every run. The issue immediately shows up as byte-addressable NVM gets widely adopted in the near future. Forcing all objects (by many different programs) with cross references to reside on one huge NVRegion is not a practical solution. In our example, for instance, program II may not even have the privilege to write data into Region 2, which is a situation that traditional GCs do not face.

Although there have been many recent studies on NVM (e.g., [24, 27, 34, 36, 43, 46, 49]), no work has studied this problem before. In fact, this problem has never been pointed out in previous literature. There are several recent studies on *position-independent pointers* [19] (or called *relocatable objects* [55, 56]. But the situations considered in those studies are only changes of the starting address of an entire NV region. Their solutions do not preserve addressability of objects when the objects are moved to a different location inside an NV region, as they assume that the offset of an object to the starting address of its NV region remains unchanged.

In this paper, we present the first systematic study on the problem. We propose *movement-oblivious addressing* (MOA), a scheme that preserves the addressability of persistent objects upon GC-triggered movements. Figure 2 illustrates the basic idea in a much simplified manner. MOA replaces direct references between objects in different regions with indirect references via a newly designed pointer structure, such that, at an object movement, the GC needs to update only the references inside the NV Region where the object resides, while the indirect references from the other region can still reach the object.

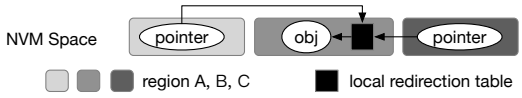



Fig. 2. Illustration of the basic idea of movement-oblivious addressing (MOA) in a much simplified manner. Three boxes show three NVRegions; two external pointers point to an object in the middle regions. By replacing direct reference with indirect reference (the white box represents a local redirecting table), GC only needs to update the local redirecting table entries, and external pointers in other regions can still reach an object even if it moves.

The basic idea is simple, but putting it to work faces some major challenges. A straightforward design may lead to 91% slowdown, due to the extra steps in object dereferences and the associated large increase of cache misses.

In this work, we conduct an in-depth exploration. We propose new address translation mechanisms that can preserve the object's information despite GC caused movements. We start with a new pointer design (OPointer) which embeds the region ID and object ID into the pointer and then looks up two separate tables to locate the object address. Such a design is inefficient when the number of objects is growing. We hence propose SGpointer to use object group to control the table size, and (inspired by multi-level page table designs) multi-level GPointer to add the flexibility of object grouping.

[2]Java GC uses reference counters to track the liveness of objects; it often moves live objects to reduce memory fragmentation.

The solutions equip traditional GCs with cross-region addressability even in the presence of object movements. The integration into JVM is straightforward: override the dereferencing operation of persistent pointers with the addressing pattern proposed in this solution. Mainstream GCs, such as Java's default G1 GC, provide reference barriers, which allow JVM developers to customize the dereference operation of a Java reference. Integrating the proposed techniques hence incurs only minor modifications to the reference barriers and interfaces of object creation, movement, and destruction. The pure software implementation and the hardware-based implementation proposed in this work offer the choices to suite different needs for runtime efficiency and ease of adoptions. Experiments demonstrate that the solutions can realize MOA while reducing runtime overhead of alternative solutions substantially (up to 60%).

To the best of our knowledge, this is the first work proposing MOA pointers for supporting references across NVRegions. It makes the following major contributions:

- It, for the first time, points out the addressability problem that GC causes to the reusability of persistent objects.
- It introduces the concept of movement-oblivious addressing (MOA), and develops the first set of solutions to materialize the concept for solving the addressability problem and compares them.
- It evaluates the designs on six benchmarks and shows that an adaptive approach is necessary to avoid fragmentation and compaction is necessary for highest flexibility.

## 2 PREMISES

***NVM Access Model*** Each NVRegion has a unique integer ID, stored in its head. Accesses to NVM follow the models described in a prior work [19]. An NVRegion needs to be opened through an API call before its data can be accessed; the call maps the region to the virtual address space. Accesses to NVM data, however, do not need to go through special APIs, but through direct pointer dereferences in a way similar to accesses to standard DRAM data; this is essential for productivity and code compatibility. There are three types of pointers that could point to NVM data; one for those pointing from DRAM to NVM, one from one place in an NVRegion to another place in the same NVRegion (called intra-region pointers), and another from one NVRegion to another NVRegion (called inter-region pointers). The three types of pointers have different degrees of requirements for position independence as discussed in prior work [19]. The prior work proposes an addressing approach for each type of pointers. Figure 3 illustrates the conceptual view of the organization and a code snippet for accessing a node of a durable set, NVSet. The APIs, *open_region* and *get_root_node*, are proposed by prior work. API *Deref* translates an OPointer to a virtual address.

Note that although this work assumes that a portion of the main memory utilizes NVM such as Intel Optane DC DIMM, the access model and the proposed techniques are general, and can be applied to other types of main memory.

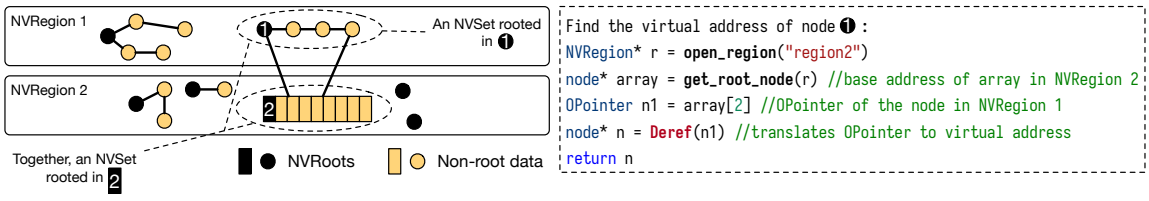



Fig. 3. A conceptual view of the organization of NVM. An NVM consists of multiple NVRegions, with possible cross-region references.The code snippet shows the APIs and the use of NVM regions.

***Garbage Collection (GC)*** There are a variety of GC algorithms, such as mark-sweep-compact GC, generational GC, and so on. They are invoked periodically in a program execution to automatically reclaim memory. Although there are non-moving GC algorithms which do not move objects [59], they cannot mitigate memory fragmentations. Most popular GCs pack live objects during garbage collections, causing data to move inside an NVRegion. GC shall not move data across NVRegions.

GC can be at the application level or system level. The most common kind is application-level GC, which is a thread living in the application address space, reclaiming memory for that application during its execution. An application-level GC is an essential component for all managed programming languages (Java, C#, Scala, etc.). A system-level GC is a process separate from applications; it tries to reclaim the space for the entire system. An example is disk space optimizers.

For NVM, both kinds of GC can be useful, in different ways. System-level GC sees all inter-region points-to relations, but it needs to go through the space of the entire system, and is hence a slow process, invoked infrequently. In order to facilitate liveness analysis and object movement, the system-level GC may have to infer the type of all data, which is sometime difficult. Application-level GC needs to run frequently to reclaim the space rapidly allocated and freed during the execution of an application. It can infer the data types with the knowledge from the running program. In the envisioned usage, application-level GC reclaims objects that are not the targets of inter-region pointers, while system-level GC is invoked occasionally to reclaim other objects (The two kinds of objects can be made distinguishable via markers associated with pointer types.). In fact, many real-world applications [42, 44, 57, 58], despite that they do not take advantage of persistent memory as of now, share objects among programs to enable fast inter-process sharing. They must either synchronize the programs before moving the data or use a centralized process to track all objects. Such cooperative solutions do not apply for NVM objects as an NVM object may be pointed by objects that all the running applications are not aware of or have no access permission to.

Note that all objects in an NVRegion (with or without inter-region pointers) are subject to move when application-level GC packs holes into large consecutive free space. As an application-level GC is usually implemented as a thread within the application address space, it maintains the addressability only for pointers within its address space. In traditional systems, if an object is in a shared memory (potentially shared by multiple processes), GC typically does not move it; in a cooperative setting, GC could work on shared objects [57], but pointer updates need to be done through explicit synchronizations among the processes.

For objects on NVM, new complexities arise. Because an object may live across the lifetime of a program and some cross-region references may be created in some earlier executions of other programs, when a program uses an NVM object, it may not be aware of all the pointers pointing to that object—some of those pointers may be on an NVRegion that this program even have no permission to access. As a result, when the GC thread in this program moves that object, the GC has no way to update all the pointers pointing to that object in existing designs. The next section presents our solution to this problem.

## 3 DESIGN AND IMPLEMENTATIONS OF MOA

In this section, we first provide a formal definition of MOA, specify the scope of our work, and then present three mechanisms to realize MOA for objects on NVM. These mechanisms share a basic idea, replacing direct references with indirect references via novel pointer structures and assisting addressing schemes. They form a progression, one built on another, with different tradeoffs and flexibility.

### 3.1 MOA

MOA is a concept we initiated in this work to describe the mechanisms which can preserve the addressability of a data object even when the object is moved. For clarity, we provide a formal definition of MOA as follows:

DEFINITION 3.1. *Let $O$ be a data object, and $S$ be the entire set of data references pointing to $O$ at time $t$, that is, $\forall p$, $p \in S \Longleftrightarrow L_1 == T\,p$, where $L_1$ is the virtual address of object $O$, and $T\,p$ returns the target address of a reference $p$. An addressing mechanism is* **movement-oblivious addressing** *if it meets the following condition: When object $O$ changes its virtual address to $L_2$ from $L_1$ ($L_2 \neq L_1$) at time $t$ when no other changes happen, the target addresses of $T\,p, p \in S$ change to $L_2$ immediately after $t$.*

The definition is general, covering all kinds of data movements incurred by all kinds of reasons (manual data movements initiated by programmers, automatic data movements triggered by runtime, etc.). In this work, we focus on GC-caused data movements. Such movements are implicit (invisible) to programmers. So unlike data movements initiated by programmers in the application code, in this case, maintaining the addressability of the moved data should be automatically supported by the underlying system rather than programmers.

Next, we present the MOA schemes that we have developed. For easy reference, we put into Figure 4 the acronyms used frequently in the following discussions. We start our description with OPointer, the simplest among all.

Name convention:
**A**: Address (offset); **G**: Group ID; **O**: Object ID; **R**: Region ID; **TB**: translation table;

**RTB**: Region ID → Address; **ARTB**: Address → Region ID;
**OTB**: Object ID → Address offset in a region; **AOTB**: Address offset → Object ID;
**GTB**: Group ID → Address offset in a region; **AGTB**: Address offset → Group ID;

Fig. 4. Acronyms reference

### 3.2 Basic Proposal: OPointer

The basic idea behind the solution *OPointer* is to localize the needed updates when there is a data movement, by replacing direct references with indirect references. The replacement is materialized through a new pointer structure and some assistant system data structures and runtime operations.

*3.2.1 Design.* The OPointer breaks the 8B-width pointer into two fields, the first 32 bits (starting from the most significant bit) are used for region ID (RID) and the remaining bits for object ID (OID). Similar to RID, which is associated with an NV region, each OID is associated with an object that needs MOA. Some prior NV pointer designs [19, 56] also store RID inside a pointer, but they store offset rather than OID of an object in the other part of the pointer. The difference is important: When a data moves, its offset in the NVRegion changes, but its OID does not.

Table 1 lists the assistant data structures and their roles in supporting OPointers. They include four mapping tables, RTB, ARTB, OTB, and AOTB, and FPool, a min-heap based pool of free OIDs. Each process has its own RTB and ARTB. They give maps between RIDs and base addresses of NVRegions. An entry is put into both whenever the process opens an NVRegion. The two tables are transient. OTB and AOTB are persistent, living in the NVRegion that they help manage, providing maps between OIDs and offsets of persistent objects on the region. FPool is per NVRegion and lives on it as well. It contains a pool of free IDs that new persistent objects on the NVRegion may choose to use. OTB, AOTB, and FPool are addressed through the region's metadata, as Figure 5 shows.

Table 1. Assistant Data Structures for OPointers. The last column shows the operations use the data structures, P for pointer dereferencing, A for pointer assignment, N for newly object allocation, and F for object free.

| Acronym | Role | Implementation | Sharing | Operations |
|---|---|---|---|---|
| RTB | map RID to base addresses of region and OTB | direct address table | across whole system | P |
| ARTB | map base address of region to RID | sorted binary tree | across whole system | A N |
| OTB | map OID to intra-region offset | direct address table | per NVRegion | P F |
| AOTB | map intra-region offset to OID | hash table | per NVRegion | A N F |
| FPool | manage free OIDs | min-heap | per NVRegion | A N F |

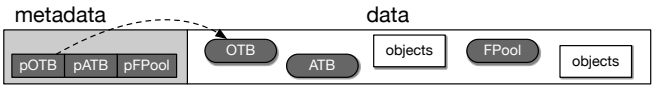
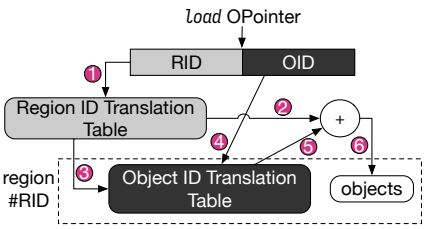



Fig. 5. OTB, AOTB and FPool reside on NVRegions, addressed through the region's metadata (Sec. 3.2).

*3.2.2 Operations and Enabled MOA.* A dereference of an OPointer consists of several operations, which, by leveraging RTB and OTB, translate the RID and OID contained in the OPointer into the base address of the target NVRegion and the offset of the object in that region respectively, and then sum them into the address of the target object. Figure 6 shows an example and the pseudo-code. A translation from an object's address to an OPointer consists of a reverse process via ARTB and AOTB.




```
Deref(OPointer):
1  RID = OPointer >> 32
2  region_addr = RTB[RID*2]
3  INT64* OTB = (INT64*)RTB[RID*2+1]
4  OID = OPointer & 0xFFFFFFFF
5  object_offset = OTB[OID]
6  object_addr = object_offset +
                 region_addr
7  return object_addr
```

Fig. 6. Illustration of dereferencing an OPointer (Sec. 3.2); circled numbers correlate with code line # on the right.

OPointer helps enable MOA by localizing the needed changes when a persistent object moves. The only needed updates at a data movement are to the two tables, OTB and AOTB, updating their entries with the new offset of the object in that NVRegion. As both OTB and AOTB reside on that NVRegion, the GC can simply make the changes as part of the GC process. The value of an OPointer that points to that object needs no changes; a dereference of it still reaches that object after the move.

OIDs are managed through FPool. FPool is a min-heap containing a set of IDs that new persistent objects may take on. When a new persistent object is created that is visible to cross-region references, the runtime requests an OID from FPool, and puts the OID into OTB and AOTB to associate the OID with the offset of the object in the NVRegion. FPool always returns the minimum free OID to minimize the fragmentation in OTB. If FPool is empty at an ID request, the runtime resizes the OID table by doubling its size, and then adds the new indices available into FPool; AOTB is resized as well. The use of min-heap ensures logarithmic computational complexity of an ID request; deleting a persistent object puts its ID back to FPool.

*3.2.3 Limitations.* The major weakness of OPointer is the frequent but slow accesses to OTB. It assigns an OID to each object referenced from other regions. The size of OTB hence can be large. Every dereference needs to access OTB. The large size of OTB may entail many cache misses and hence slow dereferences.

### 3.3 Proposal II: Multi-Sized GPointer (SGPointer)

Multi-Sized GPointer is a variant of OPointer. It reduces the size of OTB by grouping the objects and sharing the group ID (GID) across the grouped objects.

In Multi-Sized GPointer, an object group is a consecutive memory space filled with two types of memory blocks: (1) the free blocks that can be allocated to objects and managed by memory management; (2) the grouped objects whose headers are in the group. The object groups have the following properties:

- An object group is the smallest unit for data movement;
- An object could span beyond the end of the group it belongs to;
- One group cannot overlap another;
- The group size is pre-defined.

The first property ensures that the offsets of the objects to the starting address of the group stay unchanged after data movements. With this design, a pointer can now consist of three parts, RID GID Offset, where RID is the NVRegion ID, GID is the Group ID of the object, and Offset is the offset of the object to the base address of the group. The large OTB and AOTB in OPointer are now replaced with much smaller GTB (Group IDs to the base addresses of the groups) and AGTB (reverse GTB), leading to better data locality and hence cache performance. At a data movement, the only updates are to the group's base addresses in GTB and AGTB. As data offsets in a group remain unchanged, they can be retrieved through the original GPointer.

The second property ensures that GPointer works even for objects larger than the pre-defined group size. Figure 7b illustrates a 256B group. The last object in the group spans beyond the boundary.

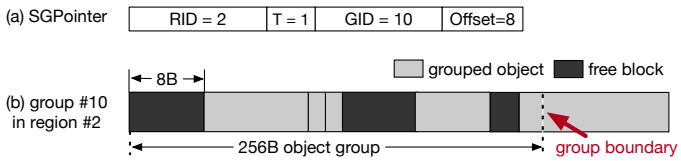



Fig. 7. SGPointer (a) and its associated group (b). The SGPointer points to the first object in the group.

The third property is easy to understand. The final property is necessary for efficient group and object management. A question is what size should be used. A large size reduces the sizes of GTB and AGTB and hence improves cache performance, but leaves a group more likely to be fragmented. (Fragmentation happens when an object is freed in the middle of a group.)

Multi-Sized GPointer addresses the issue by providing multiple size options for an object group (1B, 256B, 4KB, and 64KB in our implementation). To materialize the flexibility, a Multi-Sized GPointer format is designed to contain four fields: the first 30 bits constitute RID, and the following two bits constitute a T field. T can be 0–3, respectively corresponding to the sizes of 1B, 256B, 4KB and 64KB in our implementation. The remaining bits constitute the G and O fields, for GID and intra-group offset.

The assistant data structures need to support the multi-sized design. It uses a sorted binary tree for AGTB (similar to AOTB, it maps intra-region offset to GID) and use four replicas of some data structures, one for each group type. Specifically, the GPointer uses four GID tables (GTB) and four FPools (denoted as GTB$i$ and FPool$i$ for group type $i$, $i = 0, 1, 2, 3$). In addition, the GPointer installs the base addresses of the four GTBs of an NVRegion into the RTB table, along with the base address of that region. The GTBs store the offsets of the groups to the base of the region; they are shared cross programs. The metadata of the NV region is extended according to the increasing number of the data structures. As an optimization, the Multi-Sized GPointer uses a single AGTB rather than four AGTBs. It maps each intra-region offset to a value composed of the group type and the GID, by padding the GID to 32 bits and concatenating the group type with the GID. Section 3.6 will explain how to choose the appropriate group a newly created object should be placed into.

### 3.4 Proposal III: Multi-Level GPointer (LGPointer)

Although Multi-Sized GPointer offers some flexibility in what groups an object may get into, it has a rigid design. The size of a group is fixed. If a large group happens to suffer severe fragmentation, it cannot be broken down into smaller ones to mitigate the issue.

We introduce Multi-Level GPointer to augment GPointer with dynamic adaptivity in group sizes. In this design, at runtime, an object group may be split into multiple smaller groups, and multiple groups may merge into a large group, with all happening transparently to the applications, programmers, or users.

*3.4.1 Pointer and Data Structures.* The box on top of Figure 8 shows the structure of a Multi-Level GPointer. The first 32 bits define the RID, the following 32 bits are divided into four fields, namely P0-P3, which are the indices of associated GTBs, detailed next.

A key enabler of the flexible group sizes in Multi-Level GPointer is the first (most significant) bit of a GPointer. We call that bit the *type bit*. P0 is the index to the first level GTB, namely GTB0. An entry in GTB0 may be one of two types. If its type bit is 0, the entry is the offset of a next level GTB (GTB1) in the NVRegion; if the bit is 1, it is the offset of a 16MB-object group in the NVRegion, and the suffix of the pointer P1 P2 P3 form the offset of the object in that 16MB-object group. GTB1 and GTB2 have the same design as GTB0, except that their corresponding object group sizes are 64KB and 256B, and the suffixes are P2 P3 and P3 respectively. Figure 8 gives a simple illustration. Each entry in GTB3 can be only the offset of a 1B-object group. (As it is 1B in size, no intra-group offset is needed.)

For an NVRegion, there is only one GTB0, but the number of GTBi is $n_{i-1}$ ($n_{i-1}$ is the total number of entries in all GTB$i-1$; $i = 1, 2, 3$).

The GTBs are kept in the region mixed with other data, as the bottom box in Figure 8 shows. The runtime walks through the GTBs according to the fields in a Multi-Level GPointer.

The usage of other assistant data structures in Multi-Level GPointer are similar as in the Multi-Sized GPointer.

The described design of Multi-Level Pointer entails four possible group sizes, 1B, 256B, 64KB and 16MB, corresponding to the four levels of object groups. The full ID of an object group is the concatenation of the group ID fields in a pointer. For instance, if the group is a level-2 group, the concatenation of P0 and P1 forms its ID.

*3.4.2 Group Splitting and Merging.* An appealing property of Multi-Level GPointer is that it allows easy efficient splitting and merging of object groups. The enabled dynamic adjustability of group sizes opens opportunities for enhancing the tradeoff between locality in GTB accesses and fragmentation in a group.

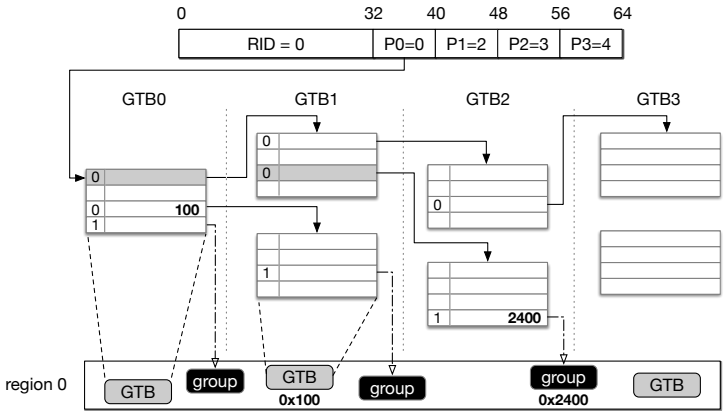



Fig. 8. Multi-level GTBs (Sec 3.4).

*Splitting.* The splitting operation can be employed anytime even when the program is running. The runtime first locates the GTB entry (say $e$) of the object group that is going to be split. It then prepares a next level GTB with the offset of each sub-group being filled. It finally updates $e$ by replacing it with the intra-region offset of the new GTB and sets the type bit of $e$ to 0.

Figure 9 illustrates the procedure of splitting group 0x01A0. At first, the system allocates a new third level table GTB2 showed on the right side of the figure, then it fills the table with the intra-region offsets, starting from 0xABC and increase in a step of 0x100, which equals to the size of the new group, 256B. After that, it updates the entry of GTB1@1 to the intra-region offset (0xFC123) of the new table and updates the first bit to 0. The new table is named as GTB2@A0 in the graph.

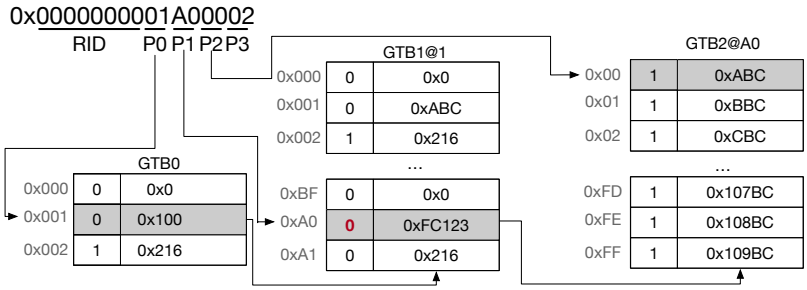



Fig. 9. Split the group 0x01A0 to sub groups with ID starts from 0x01A000 to 0x01A0FF (Sec. 3.4).

The GIDs of new small groups are from 0x01A000 to 0x01A0FF; the largest common prefix of the GIDs is the GID of the large group.

*Group Merging.* Group merging merges small groups into a large group to reduce the number of GIDs. The process is the reverse of group splitting. Merging needs to avoid group conflicts in space. Consider the object group showed in Figure 7. The last object of the group exceeds the group boundary. If the runtime places another group right after the showed group, the object could overlap with another object in the new group, causing space conflicts.

A set of object groups $S$ can be merged if and only if they meet the following conditions:

- they are of the same size;
- their IDs form a consecutive sequence;
- $L$ in binary presentation is the largest common prefix of the IDs in $S$, where $L$ is the binary presentation of the summation of the sizes of all groups in $S$. For example, group 0x0010 can be the merging result of groups 0x001000 to 0x0010FF only.
- Merging would not cause conflicts to any two groups in $S$.

Intuitively, the conditions ensure that $S$ can be merged into one single group at a higher level.

### 3.5 Hardware Support

The three methods can be applied to existing hardware via system software. With extra hardware support, they can be made even more performant. The three types of pointers all center around the use of pointer redirection and are hence all subject to redirection overhead. We propose hardware features to accelerate dereferences of the pointers. The main ideas are two, to avoid software-based bit manipulations in dereference, caching the translation with dedicated look-aside buffers. The corresponding changes to instruction set are no more than those in prior NVM hardware support [55], that is, the addition of instructions *nvld* and *nvst* for NVM accesses.

*Hardware Support for OPointer.* A region-object translation look-aside buffer (ROTLB) is introduced in this design, as shown in Figure 10. It translates OPointers to virtual addresses and then passes them to TLB. Every entry in ROTLB is an OPointer-virtual address pair. If an OPointer matches no entry in ROTLB, the translation takes a slow path. In the slow path, the hardware ❶ derives from OPointer the region ID and the object ID, ❷ loads the address of region ID translation table from a new register *ncr.0* which stores the address of per-process RTB, and finds the virtual address of the region and the virtual address of the OTB (object ID translation table) of the region, ❸ loads the intra-region offset from OTB (object ID translation table), and ❹ sums the region address and the offset to produce the virtual address of the object. It then caches the virtual address in ROTLB, and sends it to TLB for further operations.

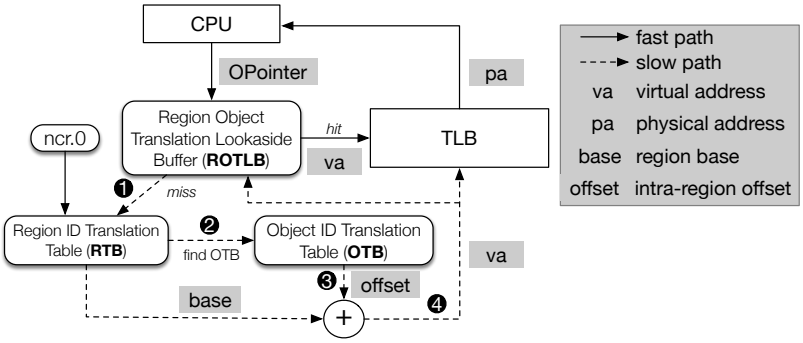



Fig. 10. region object translation look-aside buffer

ROTLB is hierarchical. A small fast L1 ROTLB is backed by a large slow L2 ROTLB and both ROTLBs are set-associative. (Section 4 gives sensitivity study on their sizes.) RTB and OTB are accessed through virtual addresses, allowing operating system to swap OTB content when necessary.

ROTLB hits have the same latency as TLB hits. A L2 ROTLB miss incurs one memory access to each of RTB and OTB. We leverage a previous design named POTB [55] to cache RTB to further reduce memory accesses.

*Hardware Support for SGPointer.* This design introduces a region multi-sized group translation look-aside buffer (RSGTLB) as showed by Figure 11. It translates SGPointers of all group sizes to virtual addresses of object groups. All the three multiplexers introduced by the design take the 2 bits type field of the SGPointer as selector input. They select different bits from the pointer. Specifically, *MUX1* extracts 7 least significant bits from GID field, which are bits 0-6 for a pointer in a 1B group or bits 8-14 for a pointer in a 256B group. *MUX2* generates the tag for searching by selecting all fields but the intra-group offset. *MUX3* selects the intra-group offset field. These operations add negligible latency as the circuit depths of the multiplexers are less than three.

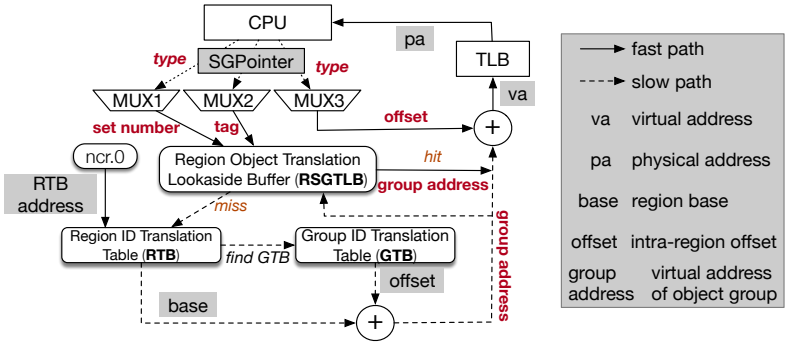



Fig. 11. Region- multi-sized group translation look-aside buffer

The other parts of the design are similar to the hardware support of OPointer (Figure 11), except that GTB replaces OTB. The two designs hence share similar performance.

*Hardware Support for LGPointer.* For LGPointer, RSGTLB does not apply because the group size is decoupled from the pointer. The design uses four buffers, with each for one of the four levels of pointers. At a pointer translation, the hardware simultaneously searches in all the buffers. It forwards the virtual address to TLB if the search hits any buffer, and takes the slow path otherwise.

***Hardware Overhead.*** Similar to prior hardware proposals that add address translation for NVM [55], the hardware changes in this work requires no modification to cache coherence. We use CACTI [10] to estimate the hardware area overhead. The hardware overhead of the design consists of a few logical circuits and a look-aside buffer (ROTLB/RSGTLB/RLGTLB). The size of the buffer is set to 9KB in total such that the access latency meets the target configuration (one cycle for 1KB L1 and seven cycles for 8KB L2). The area cost is marginal, $0.065mm^2$ according to CACTI.

### 3.6 Special Complexities

This part discusses three special complexities and our solutions. The first two relate with both SGPointer and LGPointer, and the third is specific to LGPointer.

*Grouping.* For both kinds of GPointers, a complexity is in deciding which objects should be grouped together. The primary consideration is memory fragmentation. As aforementioned, grouping objects helps reduce the size of assistant data structures, but could result in memory fragmentation within a group when some objects are freed before the rest of the group. GC are not allowed to pack objects within a group as the entire group is the smallest unit for movement. So the principle for grouping objects is to group objects that have similar lifetimes.

There are many prior studies that propose methods to predict objects lifetime [14–16]. A simple and effective heuristic commonly found in prior work is that the objects created at the same allocation site tend to have similar lifetimes. Our implementation uses this grouping strategy.

*Selection of Group Size or Level.* For SGPointers and LGPointers, when a new group is to be created, a decision has to be made on how large (or at what level) the new group should be. We present our solution by explaining the whole process when a new object is allocated.

At the allocation of a new object, the allocator will check whether the group sitting right before this object (called candidate group) has the same lifetime as it has, and if so, it tries to assign that group ID to this object (by setting some bits of its address) if that group still has enough space; if the space is short, it creates a new group (by getting a free group ID from FPool), and assigns that new group ID to this object. Note, the new group is set to a size one level higher than the candidate group if possible; the rationale is the filling of the candidate group indicates that a lot more objects are likely to be created at that allocation site. On the other hand, if the candidate group has a different lifetime, a new group of the smallest size is created for this object. An alternative scheme is to replace the adaptive group size with a fixed group size. Section 4.5 gives empirical comparisons of the schemes.

*Selection of Group ID.* For SGPointers, a newly created group can use any free group ID. But it is more complicated for LGPointers. A wrongly selected GID may prevent two groups from merging in the future. Consider two adjacent 16B objects, with each being put into a 1B group. (Recall an object can span beyond its group boundary). If the IDs of the two groups are 0 and 1 respectively, their P3 fields would equal 0x0 and 0x1. Although that works fine when they are 1B groups, when the two groups were merged into a higher-level group, their P3 fields would be regarded as the offsets of the two objects in the larger object group, which would lead to mistakes (the second pointer would points to the middle of the first object rather than the second object). So, for Multi-Level GPointers, when picking the ID for a new group, the runtime ensures that the gap of the new ID from the candidate group (say $X$) equals to the actual span of group $X$ (i.e., *max(EndOfObjects, EndOfGroup)-StartOfGroup* of group $X$).

*Size Constraints.* The formats of an NVPointer form some constraints on NVRegion sizes, which is the case for prior relocatable pointers as well [19, 55, 56]. As suggested in a prior work [19], to increase flexibility, one could design multiple formats with different number of bits for the region ID and other fields such that in the memory system, larger NVRegions and smaller NVRegions could coexist. We leave this extension to the future.

### 3.7 Computational and Space Complexity

Table 2 shows the computational complexity of the proposed techniques. The most common operations on persistent pointers are pointer dereference. The overhead of dereferencing is measured in number of table-based redirections. Multi-Level GPointers could have several extra redirections depending on at what level the object group is, but overall the techniques have similar time complexities. In practice, the overhead of dereferencing is mainly determined by cache performance of accesses to the tables.

For pointer assignment, the time complexity for OPointer is $O1$, only one hash look-up on AOTB. For GPointers, the runtime has to request a new OID from FPool. As FPool is a min-heap, the time complexity is $O\log n$, $n$ is the number of IDs in FPool. The access to AGTB also has a time complexity $Ologn$ as AGTB is implemented in a sorted binary tree. For Multi-Level GPointer, group splitting and merging have a time complexities $Om$ where m is the size of the merged GTBs.

Table 2. Computational complexity of the operations; the complexity of dereferencing is measured in the number of indirections.

| | OPointer | Multi-Sized GPointer | Multi-Level GPointer |
|---|---|---|---|
| Dereferencing | 2 | 2 | 2-5 |
| Initialization | O(1) - O($\log n$) | O($\log n$) | O($\log n$) |
| Remove | O($\log n$) | O($\log n$) | O($\log n$) |
| Splitting | — | — | O($m$) |
| Merging | — | — | O($m$) |

The memory space taken by RTB and ARTB is marginal, because they each need only one entry for one NVRegion. For OPointer, the primary space overhead comes from OTB and AOTB. For $n$ NVObjects on an NVRegion, the sizes of OTB and AOTB are both $8n$B. For SGPointer and LGPointer, the primary space cost is GTB; if $g$ is the group size, the space cost of GTB is $1g$ to 1/1 of that in OPointer, depends on the object sizes. However, the incurred object size increase is marginal compared to the average object sizes in Java programs (149–744B [9]).

## 4 EVALUATION

This section evaluates the efficacy of the proposed MOA solutions, in five aspects. The first aspect is soundness, that is, whether they can indeed support MOA in presence of data movements. On this aspect, all the solutions have no differences; they all provide sound MOA solutions, verified through observations of the values returned by memory accesses when arbitrary data movements are injected into the benchmarks. We hence focus the discussions in this section on the other four aspects, which are all related with the efficiency of the solutions:

- Time cost: As all proposed solutions replace direct references with indirect references, the redirections introduce time overhead, which is measured through this metric.
- Cache performance: As the solutions materialize redirection through some intermediate tables, this metric measures the effect on cache by the increased space usage.
- Performance sensitivity to NVM access latency.
- Memory fragmentation: This is specific to the two variants of GPointers. As objects are put into groups and are not allowed to move within a group, memory fragmentations occur when some objects are freed in the middle of a group. This metric measures how serious the fragmentations are.

### 4.1 Methodology

***Benchmarks and Data*** We evaluate our techniques with five Java benchmarks on some important data structures, as listed in the top part of Table 3. We pick these benchmarks because they exhibit different memory access patterns which are essential for comprehensively understanding the performance of the various solutions, given their focus on memory accesses. In addition, we apply the technique to a real-world application, a 10440-line key-value store ClauDB[3], which will be elaborated in Section 4.6. The real-world application is expected to exhibit more complex memory accessing patterns than the micro benchmarks, as it has multiple kinds of tasks. This improves the assessment of the proposed techniques.

Note that programming support for NVM is still preliminary for Java; Intel's Persistent Java Collection (PCJ) is currently the most developed support [6]. PCJ actually reuses Intel C library (PMDK) through JNI rather than offers

[3]https://github.com/tonivade/claudb

Table 3. List of benchmarks

| Benchmark | Description |
|---|---|
| Seq | Sequentially access an array with 179 million objects, each object is 8B and pointed by a pointer. |
| Rand | Similar to Seq but the pointers are randomly shuffled |
| List | Access 134 million nodes in a linked list. Each node is an 8B object. |
| BTree | Insert 50 millions random keys into a BTree, followed by 100 million search queries. Each BTree node has two arrays for integer keys and pointers to other nodes, and the minimum degree is 256. |
| HATTrie | Insert 23 million Wikipedia page titles[4] into a combination of trie and array hash, followed by 100 million search queries. The trie stores the prefix of the strings and the array hash stores the suffix of the strings. The hash is split into trie node and smaller hash on demand. |
| ClauDB | A real world JAVA in-memory key-value store database used as an LRU cache, with three workloads yielding 1%, 5% and 10% miss ratios respectively. |

support customized to Java. As a result, programs with it suffer from very large execution time overheads from cross-language function calls, and they make the overheads from MOA pointers look trivial (less than 1% in all cases). Thus, we implement a prototype Java NVM program performance measurement framework. The framework replaces NVM accesses in the program with native implementations, rather than a JNI function call, and hence avoids the artificial cross-language overheads from JNI.

All benchmarks have both a single-region and a four-region version. We assume the whole GC-managed heap is on NVM and the rest (e.g., stack) is on DRAM. The size of the dataset increases in proportion to the number of regions and the data is distributed to the NVRegions in a round-robin manner. Seq and Rand take up 1.5GB and 2GB memory, respectively.

***Machine Configuration*** We run real-system experiments (for all our software schemes) as well as simulation experiments (for all our hardware schemes), with configurations shown in the top part of Table 4. We use real systems to evaluate all the software implementations and a processor simulation model to evaluate the hardware implementations. The real system is an Intel i7-6700K CPU system. Our implementation leverages an instruction `bextr` to efficiently extract a number of continuous bits from a word. We use gcc 8.2 compiler and set the parameter `-mbmi` to enable the compiler support for the instruction. The hardware runs Ubuntu OS 18.04.2 LTS with 4.15.0 kernel. We load the kernel module PMEM driver and map 16GB as persistent memory. We use low-level API from Intel PMDK to manage the NVM regions. We use openjdk 1.8.0_242 and G1 garbage collector (GC) for Java programs. G1 GC is a generational GC implemented with C++.

To evaluate our architecture support, we use a trace-based simulator, Champsim[1], with parameters shown in the bottom of Table 4. The accuracy of Champsim was validated in recent works [32, 61, 62]. The simulator models an out-of-order processor and the detailed operations of TLB, cache, and memory subsystems. Page table walk latency is modeled as a fixed TLB miss penalty of 100 cycles. We also simulate the OTB/GTB walk with fixed latencies. We simulate 1 billion instruction execution for each program.

When measuring Java program performance on real machines, we warm up the runs before measuring the steady execution time to avoid the non-deterministic behavior of Java runtime. In our experiments, we collect the steady execution

Table 4. Real Machine and Simulation Configurations.

| **Real Machine Platform** (for eval. software solution) | |
|---|---|
| Processor: | Intel i7-6700K, quad core, 3.4 GHz (turbo 4GHz) |
| TLB: | L1: 4-way, 64 entries; L2: 4-way, 1536 entries |
| Cache: | L1: 8-way, 32KB; L2: 4-way, 256KB; L3: 16-way, 8MB; |
| **Simulation Model** (for eval. hw-based solution) | |
| CPU: | 4Ghz; ROB: 352 entries; load queue: 128 entries; store queue: 72 entries; branch predictor: bimodal |
| TLB: | L1: 4-way, 64 entries, 1 cycle; L2: 4-way, 512 entries, 8 cycles; page table walk: 100 cycles; |
| Caches: | L1: 8-way, 64KB, 5 cycles; L2: 8-way, 256KB, 10 cycles; L3: 8-way, 2MB, 20 cycles |
| Memory: | DRAM: CAS 12.5 ns (50 cycles), RCD 12.5 ns, RP 12.5 ns ; NVM: 75 ns (300 cycles) |
| ROTLB/ RSGTLB/ RLGTLB | L1: 4-way, 64 entries, 1 cycle; L2: 4-way, 512 entries, 8 cycles; table walk: ROTLB 120 cycles, RSGTB 110 cycles; RLGTLB 150 cycles; |

time (after 3 warm-up runs) repeatedly for 10 times; marginal variations are observed and hence the average performance is reported.

### 4.2 Execution Time Overhead

For performance comparison, we use the previously proposed relocatable NVPointer (called *RPointer* in this paper) [55] as the baseline. An RPointer consists of RID Offset, where RID is the ID of an NVRegion and Offset, which is the offset of the object in that region. This scheme supports the relocation of the entire NVRegions, but not GC-triggered data movements within an NVRegion, hence it does not support MOA pointers. By comparing to this baseline, we measure the additional overhead introduced by the MOA support. We choose RPointer as it provides the state-of-the-art performance for retrieving objects in relocatable NVRegions.

Figure 12 reports the time overhead of our three MOA pointers without hardware (parts (a) and (b)) and with hardware support (part (c)). In the experiments, we limit LGPointer to use groups no larger than 64KB; otherwise, the lower-level tables are rarely used, and LGPointer would show performance similar to that of RPointer.

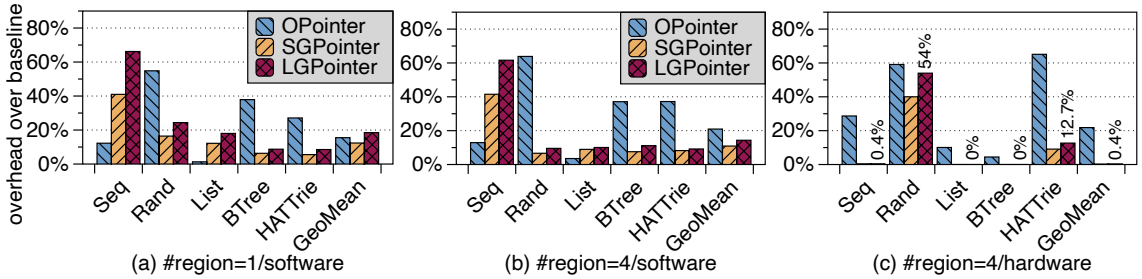



Fig. 12. Execution time overhead of MOA in software (a and b) and with hardware support (c), over RPointer[55].

Parts (a) and (b) of the figure show that the results with one or four NV regions are similar, incurring 14-21% overheads on average. Compared to OPointer, SGPointer lowers the overheads significantly, especially for `Rand, BTree, and`

`HATTrie`, thanks to grouping objects. However, OPointer performs better than SGPointer and LGPointer on `Seq` and `List`. For `Seq`, thanks to the regular data access and hence data locality, the main overhead comes from the additional operations from MOA pointers rather than cache misses; OPointer has the least extra operations. For `List`, the nodes are randomly placed in the memory and chained up in the order of node creation. Therefore, accesses to the nodes incur a random access pattern while the accesses to the assistant data structure follow a sequential access pattern. Hence, OPointer performs better as cache misses on the OTB are hidden by the slow data accesses, and it has fewer bitwise operations than SGPointer and LGPointer do.

Part (c) of the figure shows that the hardware support is very effective; it reduces the cost of MOA pointers in SGPointer to an average overhead of 0.3% (or 0.4% for LGPointer). As the hardware support exploits spatial or temporal locality but `Rand` produces a completely random access pattern, it hence represents the pathological case for hardware performance, especially for OPointer which has a large OTB.

Figure 13 reports the increase in L3 cache misses (collected via PAPI [51]) for each benchmark. The figure shows that SGPointer and LGPointer achieve a much smaller increase than OPointer, indicating the effectiveness of grouping.

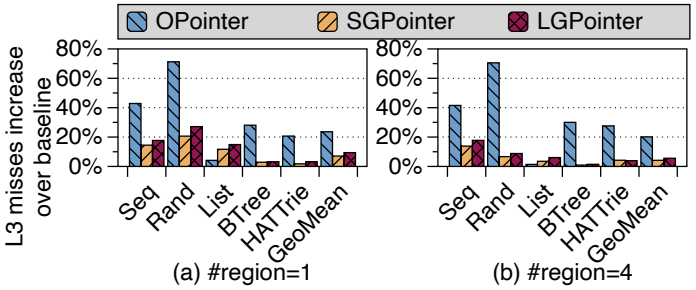


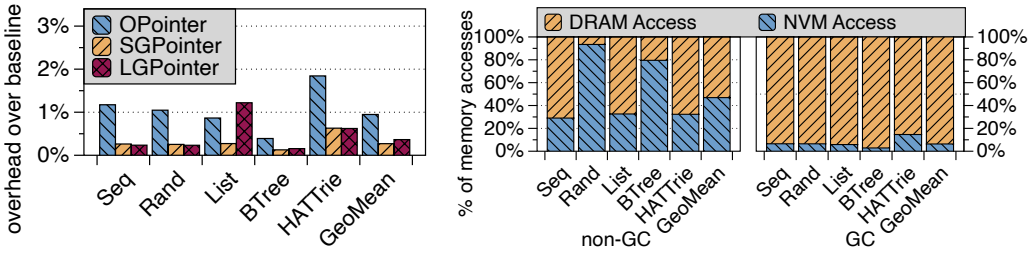

Fig. 13. The increase in the number of L3 cache misses, relative to RPointer, for the software MOA schemes on real machine.

### 4.3 Garbage Collection Overheads

This subsection details the overheads incurred by MOA pointers on the G1 GC used in JVM, with our hardware support. During the GC execution, memory accesses are emitted into a memory trace, which becomes the input to the simulator. The GC is prompted to run 10 times during the execution of each benchmark via calling *System.gc()*; the portion of objects that are destroyed increases from 10% to 100%. Table 3 shows the total number of objects before destruction. Figure 14a reports the execution time overheads of GC execution over RPointer. The figure shows nearly negligible overheads (less than 1% for SGPointer and LGPointer, slightly higher for OPointer) that are lower than for benchmark execution. Figure 14b shows that GC incurs a much smaller portion of NVM accesses vs. non-GC execution, hence it is much less affected by the MOA pointer overheads.

### 4.4 Sensitivity Study

Figure 15 reports the overheads of hardware MOA as we vary the NVM access latencies. The figure shows that the overheads of SGPointer and LGPointer remain low ($< 10\%$ in most cases) and insensitive to NVM latency, even the latency is five times of DRAM.




(a) Time overhead on GC (w/ hardware extension). (b) Memory access breakdowns by GC and application (non-GC).

Fig. 14. Overhead on GC.


Fig. 15. The impact of NVM access latencies (in cycles) on execution time overheads for OPointer, SGPointer, and LGPointer, with hardware extension.

### 4.5 Memory Fragmentation

While grouping helps GPointers reduce memory consumption, it may also increase memory fragmentation. To study the effect, we conducted an experiment to measure memory fragmentation. We investigate the use of two fixed group sizes (1KB, 8KB) and the adaptive size described in Section 3.6.

The experiment borrows a synthesized benchmark called `Fragger` [45], which maximizes fragmentation by first filling the memory with small objects of 200B or larger, then freeing some, and measuring the quality of defragmentation through allocating as many large objects as possible. We follow the first step, except that we assign every created object with a group ID as required by the GPointers. Then we extend the benchmark by repeatably freeing the $m$ objects at the end of every $n$ $m$ objects (the default `Fragger` is a special case of our method with $n = m = 1$.) As the last step, instead of allocating large objects, we emulate GC by moving the groups without live objects together to form a large free memory space (note that GC moves groups but not objects inside a group.) The memory would be composed of some free spaces, each surrounded by some live objects. Let $H$ be the size of the largest free space, and $A$ be the total free space. A metric used to measure memory fragmentation is:

$$\texttt{Fragmentation} = 1 - \frac{H}{A}$$

In our experiments, we set memory space to 4GB and the size of a small object to 256B. Table 5 shows the fragmentations of GPointers and the corresponding numbers of groups when a fixed group size (1KB, 8KB) is used or an adaptive size is used. For reference, we also show in an "OPT" column the number of groups that minimizes fragmentation, obtained based on the full knowledge on the placement and the lifetime of the created objects. The different $n$ and $m$ values create different memory freeing patterns. We further add an additional pattern of object freeing, which randomly frees half of all the objects; it is represented as “RandomHalf" at the bottom of the table.

Because GC does not move objects within a group, a larger group tends to suffer more serious fragmentation. It explains the larger fragmentation 8KB has over 1KB. On the other hand, the smaller group size of 1KB entails about 8 times more object groups (and hence worse OID cache performance) over the use of 8KB groups. The "adaptive" method strikes a tradeoff between them. When the values of $n$ and $m$ happen to make the empty spaces perfectly align with the boundaries of fixed-size groups (e.g., $n = m = 64$), the fixed-size group schemes work well. But in general cases, the "adaptive" scheme works better as its adaptive method is conscious of the objects spanning over the boundary of a group (Section 3.6).

Table 5. Memory fragmentation of fixed group sizes (1KB, 8KB), adaptive size, and optimal size (OPT). The lower the better; range is [0,1]. The #groups are 4,194,304 and 524,288 for 1KB and 8KB cases respectively.

| Mem release pattern param. | | Mem. Fragmentation | | | #Groups | |
|---|---|---|---|---|---|---|
| | | Size of a group | | | | |
| n | m | 1KB | 8KB | Adaptive | Adaptive | OPT |
| 1 | 1 | 1 | 1 | 0 | 8,388,608 | 8,388,608 |
| 31 | 31 | 0.10 | 1 | 0.71 | 1,262,800 | 2,706,004 |
| 64 | 64 | 0 | 0 | 0.33 | 589,824 | 262,144 |
| 150 | 100 | 0.02 | 0.30 | 0.19 | 536,871 | 738,199 |
| RandomHalf | | 0.87 | 1 | 0.81 | 2,593,267 | 6,709,760 |

### 4.6 Real World Application: ClauDB

The evaluation uses three workloads for ClauDB which yield 1%, 5% and 10% miss rates, respectively. Each workload is a mix of SET and GET operations, the performance is measured in the number of GETs per second; we evaluate the performance degradation incurred by the MOA solutions.

Fig. 16 shows the results on the three workloads. All the software MOA solutions suffer slightly more degradation on the *intensive* workload due to more persistent objects creation and deconstructions. SGPointer slightly outperforms LGPointer, while both retain 97% of performance. OPointer incurs 5% degradation for its large assistant data structures. All hardware MOA solutions retain 99% baseline performance across all workloads while OPointer incurs about twice overhead of the other two solutions.

## 5 RELATED WORK

Self-healing barrier [22] is a GC technique for addressing data movements in non-blocking GC. It creates a forwarding table for all objects moved during GC, such that pointers not yet updated can find the objects through the forwarding table. Once all pointers are updated, the forwarding table is removed. This technique is insufficient for the problem in this work. As GC is not aware of pointers from regions outside the view of this current application, it would not be able to remove entries in the forwarding table. The table would keep growing, facing the issues that OPointer faces.

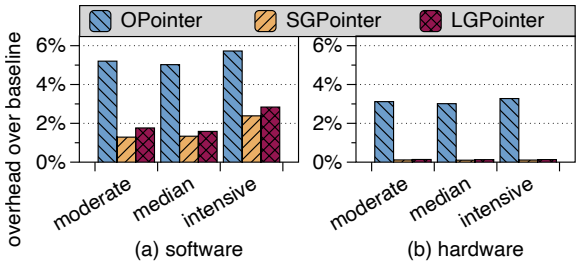



Fig. 16. Performance overheads of ClauDB with software and hardware MOA solutions; lower is better. For hardware MOA, the NVM access latency is 600 cycles, 3× that of DRAM.

There are several recent studies on position-independent pointers [7, 12, 13, 19, 41, 55, 56, 68, 70], such as Intel PMDK [7] and Twizzlers [12]. But the situations considered are only changes of the starting address of an entire NV region. Their solutions do not preserve object addressability when the objects are moved to different locations inside an NV region, as they assume that the offset of an object to the starting address of its NV region remains unchanged. For instance, in the previous work by Wang et. al. [55], the authors store in a pointer the region ID and the offset of the object in that region, and use hardware to accelerate the translation to virtual address. PMDK [7] and Twizzler [12] adopts the same pointer format. The methods break in the presence of GC: The pointer would become invalid if the object is moved in that region by GC. In contrast, this work proposes a new concept, movement-oblivious addressing, enables the property by redesigning the pointer structure and translation mechanisms, and further develops four novel methods that alleviate the space and time cost of movement-oblivious addressing.

Chakrabarti et. al. [18] mentioned NVRegions should be garbage collected upon failure and envisioned the basic use without giving a design. NV-Heaps [23] uses a reference based GC to check the reachability. Makalu [11] explored post-failure recovery time. Cohen et. al. [24] mentioned that data migration is necessary due to fragmentation or object size change; they used a fully offline copying GC that scans the objects connected to the root objects; the method supports cross-region pointers only if the two regions are loaded simultaneously when the regions with cross-region pointers could be essentially considered as a single entity. DwarfGC [36] is specifically designed for crash consistency without cross-region pointers considered.

A relevant problem is GC when shared memory is used. The programs using shared memory are cooperative processes. The GC is made possible through coordinated synchronizations and a holistic control over the participating processes. For example, XMem [57] synchronizes all programs before moving the shared object, and Skyway [44] migrates shared objects between servers and uses a centralized server to track all objects. Other systems either use non-moving GC [8, 58] or full-system GC [47]. In contrast, an NVM object may be pointed by objects in many other NV regions, and the GC of a program that uses that object may not be aware of or have the access permission to those pointers in other regions. The cooperative solutions hence do not apply.

There are many other prior studies on NVM programming. They aim at other aspects, including fault tolerance [23, 31, 53, 69], programming model [26, 39, 40, 71], algorithms [52], security [65, 66], and so on. They make important contributions, but do not address the problem in this work.

AutoPersist [48] and GCPersist [63] enables object movement between DRAM and NVM for automatic data persisting, but considers no cross-region pointers from other regions created by other applications. Espresso [64] provides crash-consistency heap for Java without tackling the addressability problem studied in this paper.

Current Java support for NVM from Intel (PCJ [6]) is incomplete. It manages persistent objects as off-heap data that are not garbage collected, and provides persistency and addressability through C library PMDK. Other projects either have similar constraints [5] or are research projects providing no runnable framework to public [48, 64].

Combining DRAM with NVM[20, 21, 38] to form a flat main memory is a state-of-the-art architecture that combines persistence and performance. DRAM is used to cache durable data before eventually updating the NVM. In terms of updating durable data in NVM, the hybrid DRAM/NVM architecture faces the same addressability issues as the NVM-only memory. As long as the programmers adopt the access model introduced in Section 2, the solutions proposed in this work is also effective for the hybrid architecture.

Pointer analysis [29, 30, 54, 60] infers from the source code which variable a pointer refers to. They may potentially augment the proposed techniques by offloading some pointer detection work to static code analysis. However, pointer analysis is hard perform precisely for general programs due to memory ambiguity, aliases, and other code complexities. The analysis is also challenging when the runtime system can move objects.

Indirection tables are used in other work [17, 37] for other purposes, such as crash consistency guarantee and fast data persistence. For example, HOOP [17] employs an out-of-place(OOP) update method, which puts the updates to a durable object on another NVM location, namely the OOP region. When the program accesses the addresses of the original object, HOOP redirects the accesses to the OOP region through a redirection table. The use of redirection table in OTB is different. First, HOOP employs physical-to-physical address mapping while OTB employs OID pointer-to-virtual address mapping—which is necessary in the context of GC-caused movements. As a result, HOOP places the redirection table subsequent to TLB, but OTB must put it before TLB in the CPU pipeline. Second, HOOP can transparently remove entries from the redirection table by applying the out-of-place updates from OOP region to the original data. It therefore can keep the redirection table small in size. Other out-of-place update work [37] adopts similar optimizations. OTB, on the other hand, does not allow such a user program-transparent operation as it can remove the OTB entry of an OID pointer only when the pointed-to durable object is deconstructed or the GC guarantees to swizzle all the OID pointers into virtual addresses, which is difficult to realize as illustrated by Figure 1. Hence, OTB is much larger than the redirection table in HOOP, facing the unique challenges in efficient address translation.

Some of the ideas in the proposed designs drew inspirations from translation look-aside buffer (TLB) [50, 67]. It however solves a new problem, GC on NVM. The proposed designs differ fundamentally from a TLB: they translate the address in the granularity of objects of various sizes rather than memory pages of a fixed size. The difference causes special complications on efficiency and correctness. For example, an object may span across the group boundary as shown in Figure 7. We therefore propose a set of optimizations to handle the complexities, including directly indexed table for OPointer, grouping and splitting mechanisms for LGPointer, and other considerations as described in Section 3.6.

## 6 DISCUSSION

*Moving and non-moving GC.* Whereas non-moving GCs are efficient and simpler to implement, they do not alleviate memory fragmentation. Memory fragmentation becomes more common and severe on NVM as the lifetime of durable objects are expected to be long. Any destruction, changing in sizes, and creation of durable data may increase memory fragmentation which gets worse over time. Past experience [4] on disk-based file systems, such as the study on a realistic workload [25], has shown that file systems can easily become severely fragmented over time. The durable objects on NVM have a similar lifetime as files, yet are subject to much more frequent updates than files, hence the importance of moving GCs for NVM.

*Cost of GC over plain programs.* Prior studies [16, 33, 48, 63] showed that GC incurs from 1.01% to 34.8% runtime overheads for an NVM program. In general, hardware accelerated GCs, such as P-Inspect [33], incur marginal overheads. The techniques proposed in this work incur 0.4% (the hardware solutions) or 21% (the software solution) performance overheads, as shown in Section 4.

*Write endurance.* The proposed techniques introduce negligible extra write traffic to NVM. They update the data structures listed by Table 1 only on durable object creation, movement, and destruction, which are much less frequent and much cheaper than object updates. On SPECjvm 2008 [35], for instance, those operations incur 24 bytes in write traffic per object, much smaller than the mean object size of 303.2 bytes. There are orders of magnitude more object updates than that.

## 7 CONCLUSION

The paper points out a data addressability problem for NVM in the presence of GC-triggered data movements. It proposes the concept of MOA, and develops solutions to localize the needed updates inside an NVRegion to keep the full addressability of an NVM object, even if the object is moved by the GC to a different locations in the NVRegion. Both pure software and hardware-based solutions are proposed. Experiments show that MOA can be realized efficiently through Multi-Sized GPointer and Multi-Level GPointer.

## 8 ACKNOWLEDGEMENT

We thank anonymous reviewers for their feedback. This work is supported jointly by National Natural Science Foundation of China (NSFC) under grants No.61832006, 62072198, 61825202, 61929103, and the National Science Foundation (NSF) under Grants CNS-2107068, CNS-1717425, 1900724, and 2106629. 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.

## REFERENCES

[1] 2017. 2nd Cache Replacement Championship. https://crc2.ece.tamu.edu/
[2] 2021. Data cleansing tool in Java. https://openrefine.org/
[3] 2021. Data visualization tool in Java. https://www.tableau.com/
[4] 2021. File system fragmentation. https://en.wikipedia.org/wiki/File_system_fragmentation
[5] 2021. Managed Data Structures. https://github.com/HewlettPackard/mds
[6] 2021. Persistent Collections for Java. https://software.intel.com/en-us/articles/java-support-for-intel-optane-dc-persistent-memory
[7] 2021. Persistent Memory Development Kit. https://pmem.io/pmdk/
[8] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA'15)*. Association for Computing Machinery, New York, NY, USA, 123–132.
[9] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. Controlling Fragmentation and Space Consumption in the Metronome, a Real-Time Garbage Collector for Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (San Diego, California, USA) *(LCTES'03)*. ACM New York, NY, USA, 81–92.
[10] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization* 14, 2 (2017), 14.
[11] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA'16)*. Association for Computing Machinery, New York, NY, USA, 677–694.
[12] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell DE Long, and Ethan L Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, 65–80.
[13] Daniel Bittman, Peter Alvaro, and Ethan L Miller. 2019. A persistent problem: Managing pointers in NVM. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada). Association for Computing Machinery, New York, NY, USA, 30–37.

[14] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA'01)*. Association for Computing Machinery, New York, NY, USA, 342–352.

[15] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: Automatic Profiling for Object Lifetime-aware Memory Management for Hotspot Big Data Applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware'17)*. Association for Computing Machinery, New York, NY, USA, 147–160.

[16] Rodrigo Bruno, Duarte Patricio, Jose Simao, Luís Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys'19)*. Association for Computing Machinery, New York, NY, USA, 16 pages.

[17] Miao Cai, Chance C Coats, and Jian Huang. 2020. Hoop: efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)* (Virtual Event). IEEE Press, 584–596.

[18] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA'14)*. Association for Computing Machinery, New York, NY, USA, 433–452.

[19] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-volatile Memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO'17)*. Association for Computing Machinery, New York, NY, USA, 191–203.

[20] Renhai Chen, Zili Shao, Duo Liu, Zhiyong Feng, and Tao Li. 2019. Towards efficient nvdimm-based heterogeneous storage hierarchy management for big data workloads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO'19)*. Association for Computing Machinery, New York, NY, USA, 849–860.

[21] Renhai Chen, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan. 2016. vFlash: Virtualized flash for optimizing the I/O performance in mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 7 (2016), 1203–1214.

[22] 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* (Chicago, IL, USA) *(VEE'05)*. Association for Computing Machinery, New York, NY, USA, 46–56.

[23] 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. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 105–118.

[24] Nachshon Cohen, David T Aksun, and James R Larus. 2018. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–22.

[25] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A Bender, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, et al. 2017. How to Fragment Your File System. *login Usenix Mag.* 42, 2 (2017). https://www.usenix.org/publications/login/summer2017/conway

[26] Alan Dearle, Graham NC Kirby, and Ron Morrison. 2009. Orthogonal Persistence Revisited. In *International Conference on Object Databases*, Vol. 5936. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–22.

[27] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI'18)*. Association for Computing Machinery, New York, NY, USA, 46–61.

[28] Lei Han and Yu Zhang. 2015. Learning tree structure in multi-task learning. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia). Association for Computing Machinery, New York, NY, USA, 397–406.

[29] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (Snowbird, Utah, USA) *(PASTE'01)*. Association for Computing Machinery, New York, NY, USA, 54–61.

[30] Ming-Yu Hung, Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq-Kuen Lee. 2012. Support of probabilistic pointer analysis in the ssa form. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2366–2379.

[31] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS'16)*. Association for Computing Machinery, New York, NY, USA, 427–442.

[32] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 2017. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy, In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (Xi'an, China). *SIGARCH Comput. Archit. News* 45, 1, 737–749.

[33] 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'20)* (Colorado Springs, Colorado, USA). Association for Computing Machinery, New York, NY, USA, 509–524.

[34] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Language-level persistency. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)* (Toronto, ON, Canada). Association for Computing Machinery, New York, NY, USA, 481–493.

[35] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE'17)*. Association for Computing Machinery, New York, NY, USA, 3–14.

[36] Heting Li and Mingyu Wu. 2018. DwarfGC: A Space-Efficient and Crash-Consistent Garbage Collector in NVM for Cloud Computing. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE'18)*. 192–197.

[37] 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.

[38] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified working memory and persistent store architecture. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 455–470.

[39] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*. USENIX Association, Santa Clara, CA, 4.

[40] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS'17)*. Association for Computing Machinery, New York, NY, USA, 693–708.

[41] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD'18)*. 413–422.

[42] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference (Eurosys'19)* (Dresden, Germany). Association for Computing Machinery, New York, NY, USA, 1–15.

[43] 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 *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS'17)*. Association for Computing Machinery, New York, NY, USA, 135–148.

[44] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Association for Computing Machinery, New York, NY, USA, 56–69.

[45] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: Fragmentation-tolerant Real-time Garbage Collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI'10)*. Association for Computing Machinery, New York, NY, USA, 146–159.

[46] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys'17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages.

[47] Yuxin Ren, Gabriel Parmer, Teo Georgiev, and Gedare Bloom. 2016. CBufs: Efficient, System-wide Memory Management and Sharing. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (Santa Barbara, CA, USA) *(ISMM'16)*. Association for Computing Machinery, New York, NY, USA, 68–77.

[48] 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* (Phoenix, AZ, USA) *(PLDI'19)*. Association for Computing Machinery, New York, NY, USA, 316–332.

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

[50] George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB slice—a low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual international symposium on Computer Architecture* (Seattle, Washington, USA). Association for Computing Machinery, New York, NY, USA, 355–363.

[51] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[52] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California) *(FAST'11)*. USENIX Association, USA, 5–5.

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104.

[54] Shao-Chung Wang, Lin-Ya Yu, Li-An Her, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2021. Pointer-Based Divergence Analysis for OpenCL 2.0 Programs. *ACM Transactions on Parallel Computing* 8, 4, Article 20 (2021), 23 pages.

[55] 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* (Cambridge, Massachusetts) *(MICRO'17)*. Association for Computing Machinery, New York, NY, USA, 800–812.

[56] Tiancong Wang, Sakthikumaran Sambasivam, and James Tuck. 2018. Hardware Supported Permission Checks on Persistent Objects for Performance and Programmability. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)* (Los Angeles, California). IEEE Press, 466–478.

[57] Michal Wegiel and Chandra Krintz. 2008. XMem: Type-safe, Transparent, Shared Memory for Cross-runtime Communication and Coordination. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI'08)*. Association for Computing Machinery, New York, NY, USA, 327–338.

[58] Michal Wegiel and Chandra Krintz. 2010. Cross-language, Type-safe, and Transparent Object Sharing for Co-located Managed Runtimes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA'10)*. Association for Computing Machinery, New York, NY, USA, 223–240.

[59] Paul R Wilson. 1992. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management (Lecture Notes in Computer Science, Vol. 637)*. 1–42.

[60] Robert P Wilson and Monica S Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)* (La Jolla, California, USA), Vol. 30. Association for Computing Machinery, New York, NY, USA, 1.

[61] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO'19)*. Association for Computing Machinery, New York, NY, USA, 996–1008.

[62] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA'19)* (Phoenix, Arizona). Association for Computing Machinery, New York, NY, USA, 1–13.

[63] 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* (Lausanne, Switzerland) *(VEE'20)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[64] 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 (ASPLOS'18)*. Association for Computing Machinery, New York, NY, USA, 70–83.

[65] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 987–1000.

[66] 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)* (Virtual Event). IEEE Press, 680–692.

[67] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Hardware-Based Address-Centric Acceleration of Key-Value Store. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Virtual Event). IEEE Press, 736–748.

[68] 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)* (Virtual Event, Spain). IEEE Press, 443–455.

[69] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez. 2011. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE Press, 466–477.

[70] Lu Zhang and Steven Swanson. 2019. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 897–912.

[71] Mingzhe Zhang, King Tin Lam, Xin Yao, and Cho-Li Wang. 2018. SIMPO: A Scalable In-Memory Persistent Object Framework Using NVRAM for Reliable Big Data Computing. *ACM Transactions on Architecture and Code Optimization* 15, 1, Article 7 (March 2018), 7:1–7:28 pages.