Efficient Support of Position Independence on Non-Volatile Memory*

Guoyang Chen
Qualcomm Inc.
guoyangc@qti.qualcomm.com

Lei Zhang
North Carolina State University
Raleigh, NC
lzhang45@ncsu.edu

Richa Budhiraja
Qualcomm Inc.
richab@qti.qualcomm.com

Xipeng Shen
North Carolina State University
Raleigh, NC
xshen5@ncsu.edu

Youfeng Wu
Intel Corp.
youfeng.wu@intel.com

ABSTRACT

This paper explores solutions for enabling efficient supports of *position independence* of pointer-based data structures on byte-addressable None-Volatile Memory (NVM). When a dynamic data structure (e.g., a linked list) gets loaded from persistent storage into main memory in different executions, the locations of the elements contained in the data structure could differ in the address spaces from one run to another. As a result, some special support must be provided to ensure that the pointers contained in the data structures always point to the correct locations, which is called position independence.

This paper shows the insufficiency of traditional methods in supporting position independence on NVM. It proposes a concept called *implicit self-contained representations of pointers*, and develops two such representations named *off-holder* and *Region ID in Value (RIV)* to materialize the concept. Experiments show that the enabled representations provide much more efficient and flexible support of position independence for dynamic data structures, alleviating a major issue for effective data reuses on NVM.

CCS CONCEPTS

• **Hardware** → Memory and dense storage; • **Computer systems organization** → *Architectures*; • **Software and its engineering** → **Compilers**; **General programming languages**;

KEYWORDS

Compiler, Program Optimizations, Programming Languages, NVM

ACM Reference format:
Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124543

*The work was done when Guoyang Chen and Richa Budhiraja were students at NCSU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124543

1 INTRODUCTION

Byte-Addressable Non-Volatile Memory (NVM) is a new class of memory technologies, including phase-change memory (PCM), memristors, STT-MRAM, resistive RAM (ReRAM), and even flash-backed DRAM (NV-DIMMs). Unlike on DRAM, data on NVM are durable, despite software crashes or power loss. Compared to existing flash storage, some of these NVM technologies promise 10-100x better performance, and can be accessed via memory instructions rather than I/O operations. For its appealing properties, NVM has the potential to create some important impact on computing. NVM has drawn some strong interest in industry, exemplified with the 3d-XPoint memory announcement by Intel and Micron [2], the "The Machine" project by HP [8], the explorations to use it for databases [4], key-value stores [5], file systems [11, 13, 32], and so on [20].

Designs of traditional systems for general applications have been assuming a separation of memory and storage. Effectively supporting NVM hence requires innovations across the entire computing stack. Recent years have seen many studies on architectural designs (e.g., [17–19, 21, 23, 24, 33]), operating system extensions (e.g., [11–13, 22, 27, 28, 31, 32]), and programming model developments (e.g., [1, 3, 6–10, 15, 29]).

These studies have primarily concentrated on how to provide ACID (Atomicity, Consistency, Isolation, and Durability) for applications that use NVM, such as how to ensure data consistency and recoverability upon a system crash, how to minimize the needed logging overhead, how to support atomic updates.

In this paper, we concentrate on a different dimension: data reusability, which refers to how efficiently and flexibly an application can use the data previously stored on NVM by this or other applications. ACID support is important for applications to function soundly on NVM, while data reusability is critical for turning NVM persistency into computing efficiency.

More specifically, this work is focused on a fundamental aspect in enhancing data reusability: the support of position independence in pointer-based data structures. A central challenge for efficient data reuse is to enable position independence for dynamic data structures. A data object is *position independent* if all the references in the data

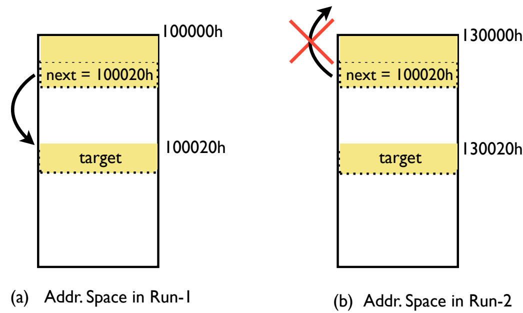

Figure 1: Issue of position dependent representation: "next" points to a wrong target in Run-2 when the data region are mapped to a different virtual address than in Run-1.

object keep their integrity (i.e., pointing to the correct addresses) regardless of where in the memory space the data object is mapped.

Dynamic data structures impose special difficulties for achieving position independence due to the usage of pointers and references in them. Consider a linked list, in which, each node contains a field "next" that carries the address of the next node as illustrated in Figure 1 (a). This implementation of pointers is not position independent. If the linked list is mapped to a different location as Figure 1 (b) shows, the value of "next" of the first node would not equal the current address of the second node, causing errors in the linked list. All data structures containing pointers are subject to this issue, including linked lists, graphs, trees, hash tables, maps, classes, and so on.

Position independence is an essential requirement for NVM. Data on NVM are supposed to be reused across runs and applications. Requiring a data object to always map to the same virtual address is impractical in general systems. The address could be already used by some other shared memories or objects. Moreover, modern operating systems have broadly adopted *address space randomization* [26], a security technique that prevents malicious attacks by varying the starting addresses of stacks and heaps at the launching time of a program.

Position independence is needed on traditional secondary storage, such as database or Java objects. The method used there is *pointer swizzling/unswizzling* [14] (happening during data *serialization/deserialization*), which uses position-independent indices (e.g., offset in a file) for pointers on the secondary storage and converts them to direct pointers when the data are loaded into memory. The drawback is the conversion overhead. It is not a major concern on traditional storage as accesses to the traditional storage are slow, and the latency largely outweighs the conversion overhead.

But as NVM has a latency orders of magnitude smaller than traditional storage, the conversion overhead becomes much more prominent in the program execution time. Measurements (Section 6) show that traversals of list, tree and other data structures are subject to 3-4X slowdowns with the use of swizzling at the loading time and unswizzling at the end. Besides pointer swizzling, *fat pointer* (or smart pointer) [10, 25] is another popular option, which is what existing NVM proposals have been using for position independence. It uses a struct or class for a single pointer, in a form like `struct{uint regionID; ulong offset}` (`offset` is the offset of the target location from the base address of the NVRegion that has an ID equaling `regionID`). It calculates the actual address from the fields of the struct or class at each pointer access. It doubles the space usage of pointers, and at the same time, incurs time overhead similarly large as swizzling causes. A better support of position independence is hence essential for data reusability on NVM.

This paper describes our solutions to the problem, which consist of two novel representations of pointers and some programming language extensions. The key in our solution is the concept of *implicit self-contained representations* of pointers. A pointer is in an implicit self-contained representation if it meets three conditions: (1) it is no larger than a normal pointer; (2) it contains all the info needed for locating the target address of the pointer; (3) programmers can use the pointer for data accesses in the same way as using a normal pointer (the type declaration could differ). Such representations minimize the space overhead of pointers in a dynamic data structure, and at the same time, allow intuitive usage of non-volatile pointers. The challenge is how to materialize the representations such that they incur the minimum runtime overhead. Concatenating the two fields of a fat pointer (regionID and offset) into one 64-bit word, for instance, can make the pointer self contained. But it would still require translations between the regionID and the address of the region at runtime, which, without a careful implementation, could easily incur large overhead.

We solve the problem by developing *off-holder* and *Region ID in Value (RIV)* pointer representations, two *implicit self-contained representations* of pointers to enable efficient support of position independence. By storing the offset of the target location and the pointer's own address, *off-holder* offers an efficient support for pointers pointing to locations within the same NVRegion as the pointers reside. RIV gives a more general solution, supporting pointers both within and across regions. To make the runtime convertions between region IDs and addresses efficient, we come up with a novel approach to realizing the convertions through some mapping functions involving only several bit transformations. We provide a comparison of the two new representations with existing alternatives both qualitatively and quantitatively.

Experiments show that the enabled representations provide much more efficient support of position independence for dynamic data structures than existing representations do. For single-region accesses, the overhead reduces from over 3X (swizzling or fat pointer) to 1.13X; for multi-region accesses, the overhead reduces from 2.2X to 1.4X. The exploration also reveals some other insights on supporting position independence on NVM.

Overall, this paper makes the following major contributions:

(1) It describes the first systematic study on supporting *position independence* for NVM pointers.

(2) It proposes the concept of *implicit self-contained representations*, develops *off-holder* and *RIV* as two methods to materialize the concept efficiently, and discusses the needed C/C++ type system extensions and the associated code translations.

(3) It compares off-holder and RIV with existing alternatives, both qualitatively and quantitatively. It evaluates the techniques on a set of data structures, demonstrates the promise of the techniques for removing the barriers for efficient reuses of dynamic data structures

on NVM, and reveals a series of insights on various options for supporting position independence on NVM.

2 BACKGROUND

This section provides some background on the programming paradigms and organization of NVM, offering necessary premises for the rest of the discussions.

2.1 Paradigms for NVM Programming

There are at least two paradigms for offering NVM programming support, differing fundamentally on what role they assume NVM would play in a system. One is to treat it as a second-level storage below DRAM. All accesses to NVM have to go through DRAM; data on NVM need to be copied into DRAM for accesses and copied out to NVM later for persistence. Some special storage APIs have been proposed to support this paradigm including the usage of NVM as swap space of main memory [1]. The other paradigm is to treat NVM as part of the main memory of the system in addition to DRAM. Data on NVM can be directly accessed without the need of going through DRAM. We call these two paradigms indirect and direct paradigms respectively.

The indirect paradigm allows smooth migrations of legacy code to using NVM, especially if the API is implicit (e.g., using NVM as a swap space) or resembles traditional file I/O. However, the copy-in and copy-out could incur some large (unnecessary) overhead [10, 29]. More fundamentally, it fails to fully exploit some key features (e.g., random, byte addressability) of NVM. The direct paradigm, by avoiding such copy operations, has the potential to better tap into the full potential of NVM. In this work, we assume the second paradigm: Processors use NVM as the main memory and directly loads from and stores to it.

2.2 Organization of NVM

Regarding data organizations on NVM, there have been various designs. Some assume that the entire NVM is a single unit, but more often, NVM is regarded as composed of a number of units with each being a standalone chunk of memory to be loaded and shared. Each unit is called an *NVRegion* in this paper. This latter assumption offers more flexibility for the management and reuse of data on NVM. For instance, a third party may create a set of data structures on some products. By putting them onto an NVRegion, they can be easily shared with other users, who can simply load that NVRegion into their application.

In this paper, we assume this multiple-region organization of NVM[1]. Figure 2 shows the conceptual view of the organization. It consists of a number of NVRegions. Each NVRegion is a consecutive (in virtual address space, not necessarily in physical space) chunk of memory. It contains one or more *NVRoots*. Each NVRoot is a named entity corresponding to a location in an NVRegion, from which, a set of data (e.g., array elements, tree nodes) can be reached through either regular strides or pointer chasing. The set of data reachable from an NVRoot is called the *NVSet* of that NVRoot. An NVSet may span one or more NVRegions. In Figure 2, there are seven NVRegions. The first NVRegion contains five NVRoots, each of which leads an NVSet, including a graph, an array, a linked list, and two individual variables. The second NVRegion contains an array, whose elements point to some nodes in the linked list in the first NVRegion. As a result, the linked list and the array together form the NVSet of the NVRoot of that array. Some meta data are stored in each NVRegion to indicate the locations of the roots, the data types in each, and some other attributes. Each NVRegion has a unique integer ID, stored at the beginning of each NVRegion. Through an API call, users can open an NVRegion; that maps the region to the virtual address space, and its content can then be accessed through its NVRoots (by default, no copy to volatile memory is needed.)

[1]It also covers the designs in which a single NV-region supports multiple sub-regions within it.

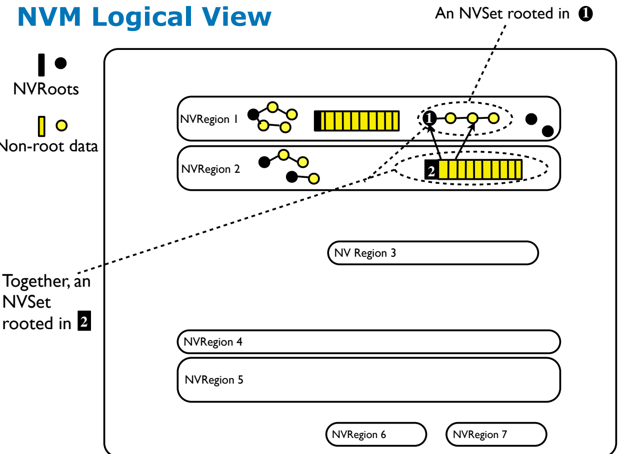

Figure 2: A conceptual view of the organization of NVM. Seven NVRegions are shown with the first two detailed. Each NVRegion contains a number of NVSets with each NVSet containing one or more NVRoots and possible some non-root data objects; an object in an NVSet can be located through at least one of the NVRoots in that NVSet. Rectangles and circles are used to represent different types of data objectives.

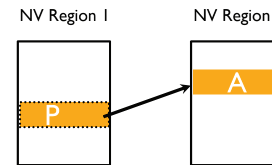

Figure 3: Example of an inter-region reference.

So overall, the assumed NVM system is as follows:

- It is directly accessed by processors as main memory.
- It consists of multiple NVRegions. Each NVRegion has a unique ID, and is a separate loading unit.
- Cross-region references (one pointer in Region 1 points to a location in Region 2) are sometimes necessary.

3 DESIRED ATTRIBUTES OF THE SUPPORT

Before presenting the various support of pointers for position independence, we first list the set of attributes of a desired support.

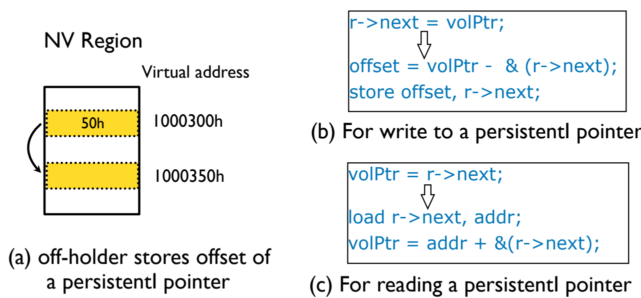

Figure 4: Example of off-holder method and its code generation.

- Efficiency in time and space. With the implementation of position-independent pointers, accesses through pointers should add minimum time overhead. Meanwhile, there should be minimum increase of space usage. Otherwise, for pointer-intensive data structures, the cost of the position-independence could make them cumbersome to use.
- Flexible to use. As Section 2 mentions, there could be needs for references both within an NVRegion and across multiple NVRegions. Therefore, position-independent pointers should support both intra-region and cross-region references, and suite the needs of various types of data structures.
- Intuitive to use. The position-independent pointers should be easy, intuitive to use. Ideally, the implementation should be transparent to users such that the use of such pointers would be not much different from the use of normal pointers. The semantics of operations on such pointers (assignment, references, dereferences, copy, function parameter passing, etc.) should be intuitive and coherent.

4 PROPOSED POINTER IMPLEMENTATION FOR POSITION INDEPENDENCE

This section presents our proposed solutions. It starts with the concept of *implicit self-contained representations* of pointers, which serves as the principle of our solutions. It then describes how the concept is materialized with *off-holder* and *Region ID in Value (RIV)*, two ways to implement position-independent pointers for dynamic data objectives on NVM. Off-holder is designed for intra-region references and features zero space overhead, while *RIV* complements off-holder with a more general support (for both intra-region and cross-region references). At the end of this section, we will discuss how, through some simple language constructs, these two may be used together to meet all possible needs.

4.1 Implicit Self-Contained Representation

A pointer is in an *implicit self-contained representation* if it meets three conditions: (1) it is no larger than a normal pointer; (2) it contains all the info needed for locating the target address of the pointer; (3) programmers can use the pointer for data accesses in the same way as using a normal pointer (the type declarations could differ). The first two make it "self-contained" and the last implies that if address conversions is needed for locating the target address, the conversions must be "implicit", requiring no programmers' explicit coding. None of existing position-independent representations of pointers are implicit self-contained. A *fat pointer*, for instance, uses two fields to represent the base and the offset of a single pointer, doubling the length of a pointer and requiring explicit address conversions.

Implicit self-contained pointers have some appealing properties that can avoid the limitations of existing solutions on both efficiency and programming productivity.

- Being self-contained and no larger than a standard pointer, such a representation incurs minimum if any space overhead.
- Having a length the same as that of a normal pointer, such a pointer, when being used in a program, does not need to explicitly reference other fields or variables (hence the "implicit" in its name.) When serving as function parameters, it can be passed across functions in a way similar to the passing of normal pointers. In comparison, with existing representations of persistent pointers, a use of a persistent pointer always involves the references to multiple fields or variables.

These features give implicit self-contained pointers the potential of being used efficiently and intuitively. We next describe how the concept is materialized through two *implicit self-contained* representations that we have developed. We start with the simpler of the two, Off-holder.

4.2 Off-holder

The idea of off-holder is simple. It stores in a pointer the difference between the pointer's target location and its own address (i.e., the holder of the value). For instance, in Figure 4 (a), the value stored in the pointer is 50h rather than the virtual address of the target. From an off-holder pointer, one can easily calculate the target address. Figure 4 (b) and (c) illustrate the corresponding conversions between an off-holder pointer and its target address.

Compared to storing offset from the starting address of the NVRegion (e.g., in fat pointers), off-holder avoids the needs for carrying the starting address of the region around. Carrying that address is cumbersome and error-prone, especially across function calls as Section 5 will elaborate.

At the first glance, off-holder seems more costly than using the offset from the starting address of the NVRegion. In the latter case, the base address could be put into a register, while in the off-holder case, the base address has to be loaded each time. In practice, it is actually no less efficient because to dereference a pointer, no matter what, the pointer itself has to be located—which already provides the base address of the off-holder method. (Adding the base to the offset incurs little overhead, and is needed in both options.)

The off-holder method incurs zero space overhead, and is quite time efficient as Section 6 will show. It is however good for only intra-region pointers—that is, their targets reside in the same NVRegion as the pointers themselves do. If the target of an off-holder pointer is in a different region, the offset between the target address and this pointer's address would depend on the locations of the two NVRegions; the offset stored in the pointer in one execution could be incorrect for another execution if the NVRegions are loaded differently in the two executions.

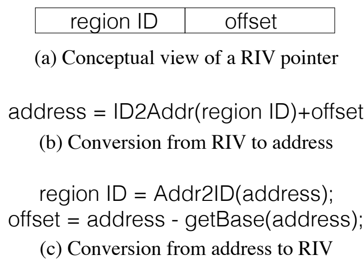

Figure 5: Illustration of a RIV pointer and the runtime address conversions.

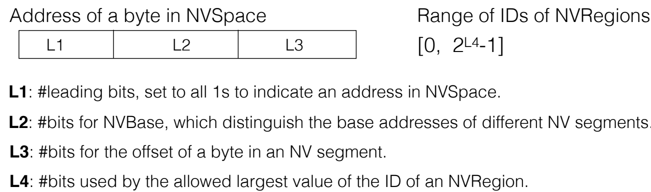

Figure 6: Illustration of an address in NVSpace, the range of allowed IDs of NVRegions, and the meanings of $L1$, $L2$, $L3$, $L4$. ($L4 \geq L2$ and $L4 + \lceil log(L2/8) \rceil \geq L3$)

4.3 Region ID in Value (RIV)

We design RIV to overcome the limitation of the off-holder method. It works for both intra-region and inter-region references. It incurs only marginal space overhead, and offers a good time efficiency through two carefully designed direct mapping tables.

Design of RIV. RIV exploits the many (often unused) bits in an address and the huge virtual memory address space in modern machines. Modern machines use 64-bit addresses. Each pointer is 64-bit long, with a range much larger than the size of physical memory equipped in a typical machine. Many bits in a pointer hence remain unused.

The basic idea of RIV is to use these unused bits in a pointer to store the ID of the NVRegion that holds the target location of a pointer. (Recall that each NVRegion has a unique integer as its ID.) The value held in a pointer now consists of two parts: the ID of the target region and the offset of the target location in that region. Some runtime conversions will be needed for using the pointers. As Figure 5 illustrates, at a dereference from the pointer, the ID part in the pointer is first used to retrieve the starting address of the target NVRegion (through ID2Addr() in Figure 5 (b)); the target address is then attained by adding the offset into that starting address. A reverse conversion is needed when an address is stored into such a pointer: The ID is first retrieved (through Addr2ID() in Figure 5 (c)) and the offset is then calculated by subtracting from the address the base address of the NVRegion (obtained through getBase() in Figure 5 (c)). The ID and the offset are then stored into the pointer. Note that using the address rather than the ID of an NVRegion in RIV won't work as the address may change across runs.

Implementation of RIV. The main challenge for implementing RIV is in making the conversion between it and the absolute target address transparent and efficient at the same time. Consider an assignment to a RIV pointer. The operation requires the ID of the NVRegion of the target address. One option could be to ask the user to use a special API (e.g., store(pointer, ID, address)) to indicate the ID explicitly. That is cumbersome and loses transparency. To allow users to write the store statement in a way same as for other normal pointers (i.e., `p=address`), we must have a method to quickly figure out what is the ID of the NVRegion to which `address` belongs.

Our solution to the challenge takes advantage of the enormousness of the virtual space in modern systems. It uses the top section of a virtual address space for NVRegions and assistant data structures. We call it *NV space*. (That adds some restrictions on *address space randomization*, but only to a minimum degree; we elaborate on this at the end of this sub-section.)

As illustrated in Figure 7 (a), an *NV space* consists of three main areas. The lowest two are used for two direct-mapping lookup tables between region IDs and their base addresses, called *RID table* and *base table*. The top part is *data area* where NV data are mapped to. Some small gap could exist between the areas.

The key question in implementing such a design is how to (1) ensure clean separations among the three areas, (2) and minimize the usage of space by the two tables, (3) and meanwhile keep the table lookups efficient. Our implementation achieves these properties through a careful usage of the bits in an address, and the design of the mapping functions for the two tables. We next describe each of the three areas in detail.

Data Area. The data area is regarded as a set of equal-sized *NV segments*, as illustrated by the left box in Figure 7 (a). The address of any byte in the NV space can be regarded as a composition of three sections, depicted in Figure 6. We use $L1$, $L2$, and $L3$ to represent the lengths of each of the three sections. The first (most significant) $L1$ bits are all 1s, indicating that it is an address in the NV space. The last (least significant) $L3$ bits indicate the offset of a byte in an NV segment. The middle $L2 = 64 - L1 - L3$ bits are called *nvbase*, which distinguishes the base addresses of different NV segments. The total number of NV segments in an NV space is slightly less than 2^{L2} because the bottom of NV space is used for the two mapping tables. Figure 6 also introduces another notation $L4$, which defines the allowed range of the IDs of NVRegions (recall that each NVRegion has an integer ID); this notation will be used later in this section.

In the basic design, one NV segment contains at most one NVRegion, and one NVRegion cannot span more than one NV segment. Hence, at a given moment, the number of opened (i.e. loaded) NVRegions in an address space cannot exceed the number of NV segments. Later in this section we will discuss how the size limitation can be alleviated. The starting address of an NVRegion is mapped to the beginning of a segment. That makes it easy to get the base address of an NVRegion from an address (i.e., `getBase()` in Figure 5 (c)): just masking the last $L3$ bits of the address.

Lookup Tables. The runtime builds the two lookup tables to facilitate the address translation. The tables are populated as the program opens NVRegions.

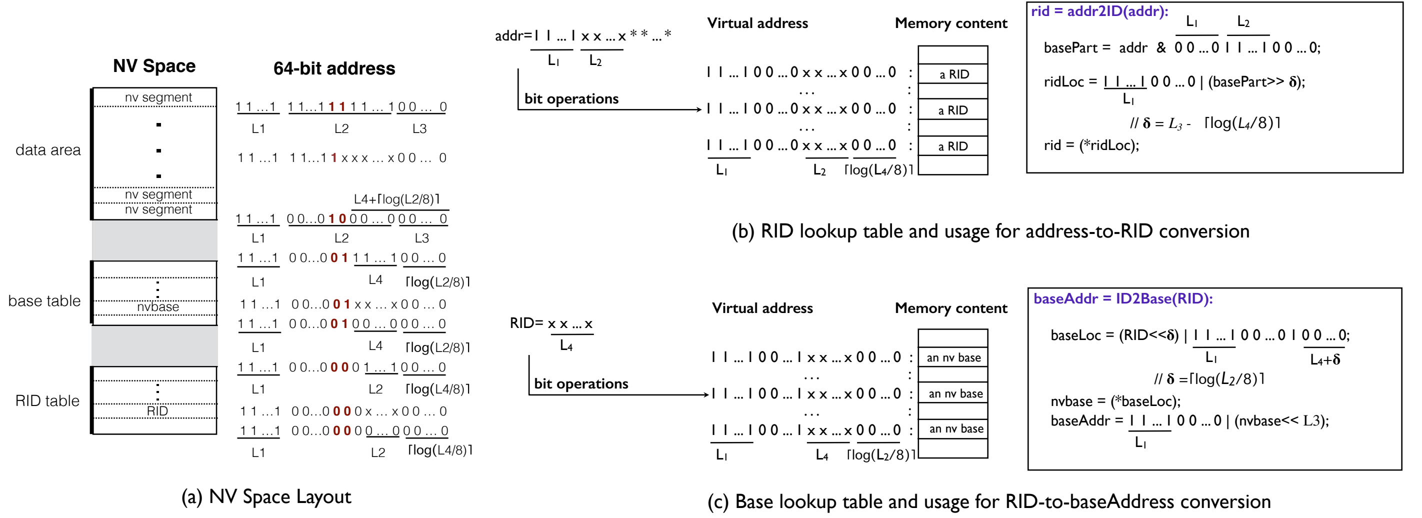

Figure 7: NV space layout and the use of the lookup tables for efficient conversions between Region IDs and base addresses. (See Figure 6 for definitions of L1,L2,L3,L4.)
($\lceil L2/8 \rceil$ and $\lceil L4/8 \rceil$ are the numbers of bytes taken by one entry in the base table and the RID table respectively; their logrithms determine the numbers of bits needed for representing the strides between the locations of two adjacent entries in the tables.)

The content of the RID table (the bottom area in Figure 7 (a)) is only the IDs (which are integers) of the opened NVRegions. The key for the table lookup to be efficient is in the design of the placement of the IDs in the memory address space. The ID corresponding to a base address x is put into a memory location y such that y is the result of several simple bit transformations applied to x, as shown by the first two lines in the code box in Figure 7 (b). That placement essentially creates a direct mapping between the base addresses and the IDs, whereby, from the base address, the ID can be directly retrieved through just the several bit transformations on the base address.

In a similar vein, the base table (the middle area in Figure 7 (a)) is built as a directly mapped table as well, as illustrated in Figure 7 (c). It contains only some *nvbase* values, which are put at some carefully determined memory addresses such that the base values can be retrieved through just several bit transformations of the ID of the NV region.

In the design, care is needed to ensure the two table areas do not overlap. The solution is to limit the range of possible ID values of NVRegions. Let that range be [0, $2^{L4}-1$]; $L4$ should be set to no smaller than $L2$ to avoid overlaps of the two table areas—the "base table" would be a region within the "RID table" region in Fig 7 (a). The relation, $L4 > L2$, means that there could be more NVRegions in a system than the number of NV segments opened in a single application, which is not an issue as very rarely, an application needs to open all NVRegions contained in a whole system at the same time. The system can impose a limit on the number of concurrently opened NVRegions to less than 2^{L2} if necessary (similar limits exist on files).

Example. We use an example to illustrate how the lookup tables work. Assume $L1$=4; $L2$=28; $L3$=32; $L4$=32. Suppose that when a program opens an NV region whose RID is 8, the runtime loads the region into an NV segment that has a base address as 0Xfffffffd00000000. At that loading time, the runtime will put into the *base table* a table entry, the content of which is 0Xffffffd, the *nvbase* of the base address of that NV segment. Specifically, the runtime puts that entry at location 0Xf000000400000020 (within the *base table*), which it attains by applying the first line of code in Figure 7 (c) to the RID (i.e., 8). Because of that placement, the *nvbase*, when needed, can be retrieved from RID through the same bit transformation, realizing the conversion from RID to the base address. At the same loading time, the runtime will also put an entry into the *RID table* ; the entry content is the RID (i.e., 8), and the location is 0Xf00000003ffffff4. It attains that location by applying the first two lines of code in Figure 7 (b) to the NV segment base address (i.e., 0Xfffffffd00000000). That placement ensures that for an arbitrary absolute address in that NVRegion (e.g., 0xfffffffd12345678), applying the same two lines of bit transformations on that address will lead to the entry that contains the RID, realizing the conversion from an absolute address to RID.

Discussions. Several points are worth noting. First, the formats of the addresses in the three areas of the NV space are carefully designed to avoid any overlap among them. An overlap would cause ambiguity in the meaning of the value at the overlapped locations. Our design prevents overlaps. In Figure 7 (a), for instance, the highlighted bits (we call them "flagging bits") on the left of the "L4" section of each "base table" address is set to 1, which ensures that the base table entries won't appear in the range of the RID table ($L4+\lceil log(L2/8)\rceil \geq L2+\lceil log(L4/8)\rceil$ because $L4 \geq L2$). Similarly, the flagging bits "11" (or "10") highlighted in the data area addresses prevents the data area from overlapping with the two tables—for this to work, our design requires $L4+\lceil log(L2/8)\rceil \geq L3$ such that that the flagging bits "11" (or "10") will reside in the "L2" section of the addresses in the data area. In addition, $L4+\lceil log(L2/8)\rceil \leq 62-L1$

is required to ensure there is space for the flagging bits in the addresses.

Second, the mapping functions involve only several bit transformations, making the table lookups time efficient. Meanwhile, the addresses of the tables differ in the precisely calculated minimum number of rightmost bits (e.g., bit $\lceil log(L4/8) \rceil + 1$ to bit $\lceil log(L4/8) \rceil + 1 + L2$ for the RID table entries), which minimizes the usage of the virtual address space by the two tables ($2^{L4} * \lceil L2/8 \rceil + 2^{L2} * \lceil L4/8 \rceil$ bytes in total). The physical memory space overhead is linear to n, the number of actually opened NVRegions; it is totally $n * (\lceil L4/8 \rceil + \lceil L2/8 \rceil)$ bytes. Consider an example: $L2 = 24$, $L4 = 32$, 20 NVRegions opened. The space overhead is only 140 bytes.

Third, the values of $L1, L2, L3, L4$ determine the upperbound of an NVRegion size and the number of NVRegions allowed in a system. For example, {L1=2; L2=24; L3=38; L4=58}, allows 2^{58} total (up to 16 millions loadable at one moment) NVRegions with each up to 256GB. To allow more flexibility in region size, one could support in a single system two levels of NVRegions, small and large, using one extra bit (represented with L0) to distinguish them. For instance, the small regions could have {L0=1; L1=2; L2=28; L3=34; L4=57} for small (up to 16GB) NVRegions, and {L0=1; L1=2; L2=21; L3=40; L4=57} for large (up to 1TB) NVRegions.

Finally, although reserving the upper part of address space for NVM adds some restrictions to address space randomization, the influence is minor: NVRegions could be still mapped to different locations in different runs, as long as they fall into the NV space.

4.4 Extensions to the Type System

The new types of non-volatile pointers can be used through macros that realize the needed address conversions. But a more transparent approach is to add some minor extensions to the type system of the programming language such that compilers can automatically recognize such pointers in a program and generate the needed address conversions at their accesses. It will also allow the compiler to better detect type safety issues. We present our extensions to C/C++ in this part.

Our extension includes two modifiers; *persistentI* for off-holder pointers ("I" for intra-region), and *persistentX* ("X" for cross-region) for RIV pointers. With these types indicating the NV pointers, the compiler can generate the corresponding address conversion code at the assignments and dereferences of those pointers. To programmers, the usage of these pointers will be similar to that of normal pointers; no special APIs would be needed for pointer assignment or dereference.

The syntax of using the two modifiers is the same as the other type keywords in C/C++ (e.g., const). As a clear semantic definition is critical for the types to be used correctly, we provide a formal definition of the semantics of the two types of pointers in Figure 8, which can be useful for the development of compiler support. Figure 8 (a) formally defines the basic implications of the two types. The type *persistentI* implies that the pointer itself and the location it points to reside on the same NVRegion, while *persistentX* allows them to reside on the same or different NVRegions. Figure 8 (b) gives the semantic domains. We include three main attributes for a variable: its value (*val*), address (*addr*), and type (*type*). Figure 8 (c)

- i ∈ persistentI ⇒ ∃ r∈ NVRegions, i.addr ∈ r, i.val+i.addr ∈ r.
- x ∈ persistentX ⇒ ∃ w,v ∈ NVRegions, x.addr ∈ w, x2p(x) ∈ v.

(a) Implications of persistentI and persistentX

i ∈ persistentI; x ∈ persistentX; p ∈ normal pointers; v ∈ non−negative integers;
$$: the left hand side of a rule; $1, $2: the right hand side operands (left to right);
b.val: value of a variable b; b.addr: memory address of a variable b; b.type: type of variable b
op: mathematical operators

(b) Semantic domain

Statement	Semantics	Statement	Semantics
p = i	$$.val = $1.val + $1.addr	i op v	$$.type = $1.type
p = x	$$.val = x2p($1.val)	v op i	$$.val=$1.val op v.val
i = x	tmp = x2p($1.val);	x op v	$$.type = $1.type
	$$.val = tmp.val − $$.addr	v op x	$$.val=p2x(x2p(x) op v.val)
x = i	tmp = $1.val+$1.addr;	&i	$$.val = $1.addr
	$$.val=p2x(tmp.val)	*i	$$.val = *($1.val + $1.addr)
i = p	$$.val = $1.val − $$.addr	&x	$$.val = $1.addr
x = p	$$.val = p2x($1.val)	*x	$$.val = *x2p($1.val)

(c) Semantics of related operations

Figure 8: Semantics of *persistentI* and *persistentX* types of pointers and their related operations. Dynamic type safety checking can be generated for the type castings; they are ommitted here. Functions $x2p()$ and $p2x()$ are explained in Section 4.3.

lists the primary evaluation rules. The first six show the operational semantics of implicit type conversions (through pointer assignments) of the different types of pointers. Functions $x2p()$ and $p2x()$ define the converstions from persistentX to normal pointers and from normal pointers to persistentX respectively. They have been explained in the previous subsection. The rules "i op v" and "x op v" define the operational consequences of mathematical operations on persistentI and persistentX pointers. Both return a type identical to the type of the pointer type involved in the expression, with the point-to location reflecting the consequence of the operations. The final four rules show the semantics of address-of and dereference operations.

In addition to the two types, there could be some extra modifiers for volatile pointers (i.e., pointers that themselves reside on volatile memory). For instance, in our designed programming system, there is a type modifier "persistent" for a volatile pointer to distinguish volatile pointers that point to volatile memory locations and those pointing to persistent memory locations. Such a distinguishment helps maintain the consistency of the data "persistent" pointers point to (e.g., help compilers insert code to ensure ACID properties for accesses to persistent memory.) Without the modifier, runtime checks (of the initial bits of an address, for instance) may be needed for telling that the access is to volatile memory or persistent memory. Because these pointers themselves are not persistent and will be initiated at the beginning of an execution and be discarded at the end of an execution, they store absolute addresses, needing no position independence support.

With the type extensions, the persistent pointers can be used intuitively. Figure 9 shows the traversal of a cross-region linked list. Each node in the list contains two pointers; one is of type "persistentI", the other is of type "persistentX". The former points to the next node in the same region, while the latter points to a NVRegion that contains the information of a product. The bold font highlights the accesses to the two fields. Both are as intuitive as

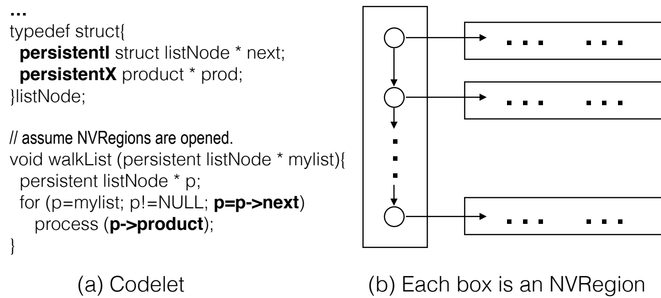

Figure 9: An example of a cross-region linked list.

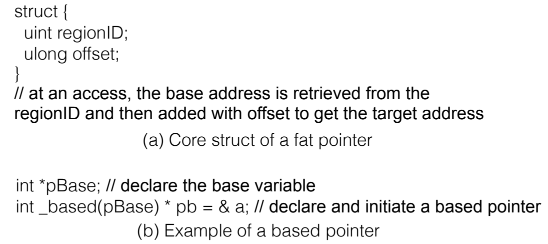

Figure 10: Examples of fat pointers and based pointers.

accessing normal pointers, requiring no accesses to any base or other assistant variables that other methods would need.

The extended type system also offers conveniences for automatic type safety checks. For instance, if there is a statement assigning a *persistentX* pointer to *persistentI* pointer, the compiler can recognize such a case through the types of the variables, and then insert a runtime check to ensure that the pointer and the address it points to indeed reside on the same NVRegion (hence the compatibility of the two sides of the assignment.) Such a check can be made optional to fit users' different levels of safety and efficiency needs.

It is worth noting that it is also reasonable to design the type system more conservatively by disallowing some of the possibly risky type conversions (e.g., "i=x"). In that case, static types checks by compilers can be sufficient for detecting the type violations.

With the semantics clearly defined, it is straightforward to implement the two types of pointers and the associated code translations in a compiler. We implement a prototype on LLVM [16] for this study based on the definitions in Figure 8.

In NVM usage, we expect that most persistent pointers will be persistentI (intra-region); persistentI can be made the default type for persistent pointers, and programmers need to pay special attentions only to persistentX pointers. As cross-region pointers are expected to be uncommon, the extra burden added to programmers will be small in most cases.

Consider a forest consisting of some trees. Each tree could be put into a region. Cross-region pointers are needed only for the few connections between trees. All other pointers would be the default persistentI pointers.

If a tree grows too large to fit into a basic NVRegion, it could be migrated to a higher-level larger NVRegion (if multi-level NVRegions are adopted), or prompt a runtime allocation error when migration is impossible.

5 COMPARISONS WITH OTHER METHODS

During the design, we have examined existing methods, and found three main options. This section gives qualitative comparisons with them; next section will compare them quantitatively.

Fat Pointer

In the *fat pointer* method, a pointer is represented with a struct, as illustrated in Figure 10 (a). The `regionID` field records the ID of the NVRegion in which the target address resides; the `offset` field records the offset of the target location relative to the base address of the NVRegion. When an NVRegion is loaded, an entry is put into a hashtable to record the mapping between the regionID and the base address of the region. At a use of a fat pointer, the base address is retrieved from the `regionID` of the fat pointer through the hashtable and is then added to its `offset` to get the absolute address. The PMEM library [25] uses *fat pointers* for persistent pointers; the *smart pointers* used in NV-Heap [10] have a similar design (through C++ class).

Efficiency: Fat pointers are not efficient in either space or time. It doubles the space usage for a pointer. At the same time, a use of a fat pointer (reference or dereference) usually needs accesses to both fields and the hashtable lookup, resulting in substantial time overhead. The overhead could be reduced if the retrieved base address of the NVRegion is cached in a register. However, if multiple NVRegions are accessed in an application, the effectiveness of the caching method can be largely throttled as next section will show.

Productivity: Fat pointers rely on programmers to put in code to explicitly conduct address conversions in their references and dereferences. The lack of user-transparency makes the use of fat pointers wordy and error-prone.

Based Pointer

Based pointers explicitly store the "base" of a memory region into a variable (which is usually a global variable). Each declaration of a *based pointer* includes a base variable as part of the type of the based pointer, indicating the base address of the memory region of the target location. The based pointer itself only stores the offset of the target location relative to the base. As Figure 10 (b) illustrates, references and dereferences of a based pointer are similar to normal pointers, requiring no explicit address converting code to be put in by programmers. The compiler makes inferences based on the type definitions of the pointers and puts in address converting code automatically. Based pointers are used in Microsoft C++.

Efficiency: Based pointers are efficient in both space and time. Space-wise, all pointers on a memory region share one single base variable; the extra space usage is just one base variable for all the pointers on a region. Time-wise, as the base variable often stays in a register, the time overhead at a reference or a dereference of a based pointer is typically just one extra addition operation.

Productivity: Even though based pointers typically need no explicit address translations, they are subject to some productivity issues.

The first and also the most important limitation is its insufficient support for cross-region references. An array of based pointers must have the same base, which is the one used in the declaration of the array, such as the base *b* in declaration "long __based(b) * ptrArray[100]". If the elements in the array shall point to locations in

different NVRegions, with based pointer method, they cannot be put into a single array. In comparison, RIV is not subject to such limitations given that it does not include base in its type definition.

Second, based pointer is not self-contained, causing its usage wordy, frequently counter-intuitive, and error-prone. Because the declaration of a based pointer must include the base as part of the type, the declaration may have to carry multiple bases, making its usage wordy (consider a declaration of a based pointer pointing to another based pointer). Consider the following simple codelet:

```
1. int *b=1000;
2. int __based(b) *p = 3000;
3. int * q=3000;
4. p=q;
5. p++;
6. q++;
```

Both lines 2 and 4 are assignments to a based pointer. Although the right hand sides of the two lines have the same value, the values that *p* gets differ: 3000 at line 2, but (3000-1000=2000) at line 4. Lines 5 and 6 increase pointers *p* and *q* by 1 respectively. However, the actual semantics differ: at line 6, the behavior is to increase the *target address* of *q by sizeof(int)*, while at line 5, the behavior is to increase the *value* in *p by 1*. In more complex code (e.g., involving arrays of pointer arrays or other pointer-based collection data structures), the code can be even more confusing.

Moreover, base pointers are especially error-prone to use in function parameter passing. Figure 11 shows three attempts to pass a based pointer to a function. The first one gets a compilation error, and the second one gets a runtime error. The third one works, but requires explicit passing of not only the pointer, but also its bases.

Serialization and Deserialization

Another option is the method of (de)serialization. In this method, the pointers in a data structure are converted into references based on position independent indexes (e.g., name or index/offset in a file) when the data structure is stored into persistent storage, and are converted back when the data structure is loaded. To ensure the consistency of the content of the data structure, all the operations in the conversion of a data structure usually have to be atomic. This method is the default way Java uses for storing and moving its data objects.

This method is not efficient. The conversions need to traverse the entire data structure. In addition, it adds complexities into the maintenance of persistency and consistency of the persistent data structure. Throughout the program execution, the data structure is in a form that is not position-independent. If there is a power loss or system breakdown before the data structure is converted back into a position-independent form, the updates to the data structure could all get lost.

Off-holder and RIV

Off-holder and RIV both are implicit self-contained representations, and are both intuitive to use.

Off-holder is efficient as Section 6 will show. It by itself does not suffice as it cannot support inter-region references. RIV can work for both intra-region and inter-region references, but its reliance on lookup tables makes it less efficient than off-holder for intra-region references. Overall, using off-holder and RIV together could overcome all the major drawbacks of existing methods.

```
int __based(b1) * __based(b2) * p1;
// try to call foo() with p1 as (one of) the parameter(s).
a) foo(p1);
   void foo (int __based(int *a1) * __based(int  *a2) * p){
     **p; // compilation error
   }
b) foo(p1);
   int *a1, *a2;
   void foo (int __based(a1) * __based(a2) * p){
     **p; // runtime error
   }
c) foo (p1, b1, b2);
   void foo (int ** p, int * a1, int * a2){
     int __based(a1) * __based(a2) * q;
     q = (int __based(a1) * __based(a2) *) p;
     **q; // correct
   }
```

Figure 11: Three attempts to pass a based pointer to a function.

6 EVALUATION

This section gives a quantitative comparison of the various implementations of pointers: off-holder, RIV, fat pointers, based pointers, and pointer swizzling. Because space efficiency and usability have already been covered in the previous section, this section concentrates on time efficiency.

We first summarize the overall observations as follows:

- Based pointer has the highest efficiency, but as the previous section explains, it is subject to some serious usability limitations, especially for cross-region accesses.
- Fat pointer incurs large (over 3X) time overhead. Caching can reduce the overhead, but is not effective when the accesses move across multiple NV regions frequently.
- Pointer swizzling causes overhead similarly large as fat pointer does unless the data are reused for many times.
- RIV and off-holder always give an efficiency close to the best. Considering their near zero space overhead and general usability, they together offer an option that meets the needs in all situations (single or multiple NV regions, various data structures of different sizes).

6.1 Methodology

We conduct our evaluations with four dynamic data structures: linked list, binary tree, hash set, and trie. They are the primary data structures used in many applications. The linked list is a single-direction linked list of a number of nodes. The binary tree is just a common tree with two children per node. The hash set contains N entries with each key's values stored in a linked list; new values are put to the end of the corresponding linked list. The trie data structure we use is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. In our experiments, we try to store many English words in a trie. Each node is a letter, and each path from the root to a leaf node represents an English word. Two words sharing the same prefix share the same subpath.

In our experiments, we populate the data structures with some random content such that each data structure contains 10000 elements. For each timing result, we repeat the measurement for 10 times and get the average. The implementations using normal (volatile) pointers are used as the baseline.

For each of the data structures, we made two kinds of implementations of the various pointers. The first is based on the PMEM.IO library [25]. The library provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming. It uses fat pointer in implementing persistent pointers. In our experiment, we replace the fat pointer with other types of persistent pointers. Code written in PMEM.IO usually runs substantially slower than code not using it, due to the extra op[illegible]ns for tracking the a[illegible]sses to NVRegions t[illegible]eve [illegible]e trans[illegible]nal store [illegible]nti[illegible] [illegible]ata [illegible]tegrity upon cra[illegible]es. T[illegible] of these extra operations on the measured overhead of persistent pointers, we made another implementation, which avoids those extra operations (hence, the transactional store semantics). We call the two versions “transactional” and “non-transactional” respectively.

We in addition implement *wordcount* application with each of the methods and report the performance.

We use the Intel Persistent Memory Emulator Platform (PMEP) for all performance measurements. Intel built PMEP to enable the performance study of persistent memory software for a range of latency and bandwidth points interesting to the emerging NVM technologies. As the previous publication describes [13], PMEP partitions the available DRAM memory into emulated NVM and regular volatile memory, emulates configurable latencies and bandwidth for the NVM range, allows configuring NVM wbarrier latency (set to 115ns in our experiments), and emulates the optimized clflush operation. PMEP is implemented on a dual-socket Intel® Xeon® processor-based platform, using special CPU microcode and custom platform firmware. Each processor runs at 2.6GHz, has 8 cores, and supports up to 4 DDR3 Channels (with up to 2 DIMMs per Channel). The custom BIOS partitions available memory such that channels 2-3 of each processor are hidden from the OS and reserved for emulated NVM. Channels 0-1 are used for regular DRAM. NUMA is disabled for NVM channels to ensure uniform access latencies. It has 16GB DRAM and 256GB PM.

6.2 Non-Transactional Implementation

We first report the results obtained on the non-transactional implementation on PMEP. The results reflect the overhead caused by the pointer implementations without the influence of the extra operations related with the transactional semantics. This experiment uses only one NVRegion in each execution. The test cases used for this experiment use only intra-region pointers because the purpose was to compare the performance of all the methods, which all (including RIV) can support such pointers.

Figure 12 reports the running times of the traversals of the four data structures when the payload in each data element is 32-byte large. The times are normalized to those when normal (volatile) pointers are used. The implementations of all the persistent pointers shown in the bar graph are through the method corresponding to the legends of the bars. Next subsection includes the measurements on multiple NVRegions.

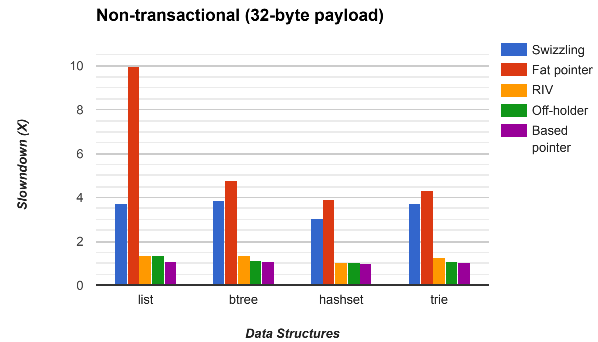

Figure 12: Slowdown (non-transactional)

Both swizzling and fat pointer cause more than 3.6X slowdowns to the traversals. RIV, off-holder, and based pointer incur much smaller slowdowns, with the average being 1.24X, 1.13X, and 1.03X respectively. Based pointer is the fastest thanks to its use of registers to store the base addresses of the NVRegion. Unfortunately, it is subject to some important usability limitations as Section 5 mentions. Off-holder has an efficiency close to that of based pointers, while avoiding much of its usability limitations.

The careful design of RIV helps it avoid the substantial hashtable lookup overhead that the fat pointer method has. Its smaller pointer size helps data locality and cache performance. These make it efficient. Its translations between region ID and base address makes it slightly more costly than off-holder and based pointer. We examine the overhead of RIV by breaking down a RIV-based read into three steps: (1) getting the region ID and offset fields from a RIV value; (2) calculating the baseAddr from the ID; (3) reading the baseAddr and calculating the actual target address by adding the offset to the baseAddr. On average, the three parts contributes 32%, 23%, and 48% of the total overhead.

When the payload size of each data element changes, the weight of pointer accesses in the overall running time changes, which leads to changes in the overhead by the pointers. When the payload increases to 256 bytes. The overhead of all types of pointers becomes smaller than in the 32-byte payload case. However, swizzling and fat pointer still cause more than 3X slowdowns on average. The average slowdowns by RIV, off-holder, and based pointer become 1.15X, 1.07X, and 1.01X.

Unlike other types of pointers, the swizzling method incurs a one-time overhead for a data structure in one execution. After the pointers are swizzled, the accesses to them do not incur any extra overhead. Therefore, if the data structure is accessed repeatedly, the swizzling overhead can weigh less. Table 1 gives quantitative measurements. As the number of traversals of the data structure increases to 10 and 100, the overhead drops from over 3X to about 1.3X and then to a negligible level. These results indicate that swizzling is not a good option for NVM pointers unless the data structures are repeatedly accessed for many times.

Table 1: Overhead (X) of Pointer Swizzling Method (32-byte payload, non-transactional).

# traversals	1	10	100
List	3.76	1.29	1.05
BTree	3.85	1.34	1.06
Hashset	3.07	1.20	1.01
Trie	[illegible]	[illegible]	[illegible]

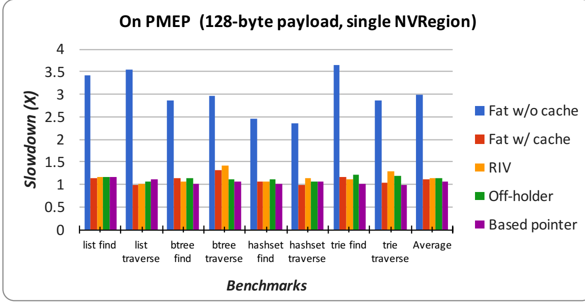

Figure 13: Slowdown (transactional, single NV Region)

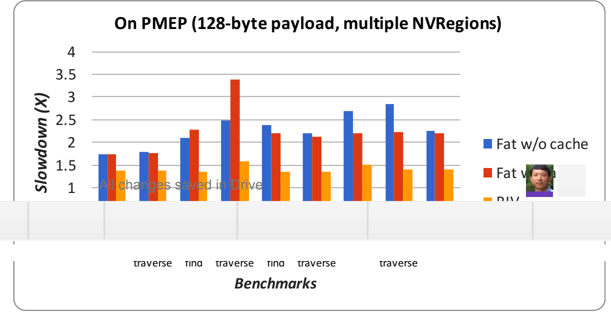

Figure 14: Slowdown (transactional, 10 NV Regions)

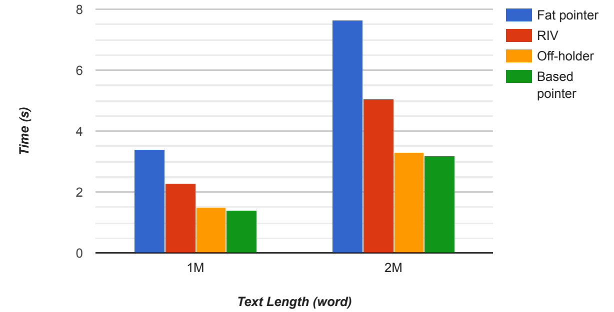

Figure 15: Execution times of *wordcount*.

6.3 Transactional Implementation

In this part, we present the results from the transactional implementation on Intel PMEP. The implementation builds on the PMEM.IO library, which creates some wrapping structure for each data item on NVM with some metadata (e.g., type info) about that data item recorded. The metadata are necessary for the library to materialize the transaction [illegible] store ser[illegible] each data [illegible] including the metadata is 12[illegible] large. The metadata and extra operations by PMEM.IO are something actual NVM programs would typically incur for getting the ACID (atomicity, consistency, isolation, durability) property on NVM. Therefore, the measurements in this experiment reflect the overhead of the persistent pointers in this practical setting. The baseline of the comparison is the performance when normal (volatile) pointers are used for the data structures working with the PMEM.IO library on PMEP.

Besides traversals, we also experiment with search for random elements in the data structures. In addition, we add an optimized fat pointer implementation, which uses two global variables, *lastID* and *lastAddr*, to store the most recent NVRegion ID and its base address. At each fat pointer access, the program first compares the target NVRegion ID with *lastID* and uses *lastAddr* if they are identical and avoids a hash lookup needed otherwise. We call this method "fat pointer with cache", and the basic fat pointer "fat pointer without cache".

Figure 13 reports the normalized running times when one single NVRegion is used. The results on the search and the traversal are similar. Both swizzling and the basic fat pointer still incur over 3X overhead in most cases. (As swizzling shows large slowdowns as in the non-transactional cases, for legibility, we omit its bars in the graph.) The caching optimization for fat pointer is effective, removing most of the overhead. It and the other three methods (RIV, off-holder, based pointer) all perform well, with average overhead being 1.11X, 1.15X, 1.13X, and 1.06X respectively.

When there are multiple NVRegions, the results on these pointers become significantly different. To measure the effects, we place the elements of each data structure across 10 NVRegions in a round-robin manner. Figure 14 reports the normalized (over the use of normal pointers) running times. (Off-holder and based pointers are not applicable to cross-region usage and are hence not shown.) The benefits of caching for fat pointers diminish because in most cases, the NVRegion to access next differs from the just accessed one; most of the lookups are not saved anymore. As a result, fat pointer with caching suffers a similar degree of slowdowns as the fat pointer without caching. In some cases (e.g., btree traverse), it even runs slower due to the extra checks it does. The RIV method, on the other hand, performs much better.

We experiment with smaller numbers (2,4,8) of NVRegions. The slowdowns caused by fat-pointer (cached and uncached) are similar to those in the 10-NVRegion case (2.1-2.5X for cached, and 2.3-3X for uncached.) The cached method is still subject to misses because the next target region differs from the just accessed one, while the hashing overhead in the uncached method is largely insensitive to the number of NVRegions.

Wordcount Application. We in addition implement *wordcount* application with each of the methods. As an important step for many document analytics, *wordcount* uses a Binary Search Tree to count word frequency in an input file. The tree is put on an NVRegion. A new node is inserted into the tree when a word is enountered for the first time; a comparison function is used to decide the location in the tree for inserting a new node. Figure 15 reports the execution times when the input contains 1Million and 2Million words to count.

Off-holder still shows a similar performance as based pointer, cutting the time of Fat Pointer by about half. RIV reduces the time by about a third.

7 RELATED WORK

Although there have been many studies on NVM programming support in the recent several years, they primarily focus on the ACID properties of executions on NVM [1, 3, 6–10, 15, 29]. Position independence and efficient reusability of NVM data objects have not yet received systematic studies.

A recent work proposes hardware support [30] to the translation of object IDs to virtual addresses, and reports promising speedups of the translations compared to prior software-based implementations. The hardware changes bring some extra complexities, including coherence among multi-core processors, that may need further explorations. Our work offers new software-based methods, which, by employing the careful designs of pointer representations, removes address translation overhead substantially. How to effectively combine the new software methods with hardware support is worth future investigations.

There have been a number of proposals on NVM programming models. Mnemosyne [29] and NVHeaps [10] are two programming models proposed for C++ and C languages respectively. Mnemosyne requires NVregions to be loaded into the same virtual address at every usage, not meeting the needs of modern operating systems for address randomization. NVHeaps uses smart pointer to support position independence. As a variant of fat pointer, it adds significant overhead into dynamic data structure usage as we have shown in the previous sections. Pointer swizzling is used to support position independence in Java and database. It is not efficient enough for NVM usage as reported earlier. The Atlas programming model [8] in "The Machine" project from HP Labs also relies on smart pointers for persistent objects. OpenNVM is an effort for designing some open-source API for taking advantage of Flash memory for programming [1]. Flash memory is an access-by-block device. The designs in OpenNVM were mostly about hiding the block accesses in programming. They treat flash memory as a level of storage below the main memory. The Storage Networking Industry Association (SNIA) have released some specifications on NVM programming models [3], which describes the expected behaviors of NVM operations at a high level, without descriptions on how the behaviors can be implemented.

Besides treating NVM as main memory for computing, there are a body of work that try to develop file systems on NVM [11–13, 22, 27, 28, 31, 32]. Accesses to NVM would need to go through storage API, limiting the flexibility of NVM for general computing.

Position independence in paging has been enabled through hardware support (e.g., page tables, TLB). This work is to provide a pure software solution that is directly applicable without the hardware cost and complexities. Overlay systems use relative addressing. Like `based pointers`, it is not self-contained, requiring the use of the base address for location calculations. That method, when being used for general programming, tends to cause programming difficulties as Section 5 shows on `based pointers`.

8 CONCLUSION

This paper examines the weaknesses of various implementations of persistent pointers for supporting data position independence on NVM, and proposes off-holder and RIV to overcome their drawbacks. It concludes that based pointer is efficient but is subject to some serious usability issues. Fat pointer incurs large time and space overhead; caching can help but not in multi-region cases. Pointer swizzling is expensive in general unless the data are reused many times. RIV and off-holder together offer a more satisfying solution. They incur small time overhead and near zero space overhead; off-holder is more efficient, while RIV is more generally applicable.

ACKNOWLEDGEMENT

We thank the MICRO-50 reviewers for their helpful suggestions. This material is based upon work supported by the DOE Early Career Award(DE-SC0013700), the National Science Foundation (NSF) under Grants No. 1455404, 1525609 and 1455733(CAREER), IBM CAS Fellowship, and Google Faculty Award. The experiments benefit from Intel Persistent Memory Emulator Platform (PMEP). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE, NSF, IBM, Google, or Intel.

REFERENCES

[1] [n. d.]. OpenNVM. http:opennvm.github.io. ([n. d.]).
[2] 2015. Intel and Micron Produce Breakthrough Memory Technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology. (2015). [Online; accessed 2016/11/08].
[3] 2015. NVM Programming Model (NPM). http://www.snia.org/tech_activities/standards/curr_standards/npm/. (2015). [Online; accessed 2016/11/08].
[4] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.
[5] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. 2013. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. ACM, 4.
[6] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust shared objects for non-volatile main memory. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[7] Dhruva R. Chakrabarti and Hans-J. Boehm. 2013. Durability Semantics for Lock-based Multithreaded Programs. In *USINEX Hotpar*.
[8] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 433–452.
[9] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.
[10] 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 ASPLOS*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380
[11] 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.
[12] Jonathan Corbet. 2014. Supporting filesystems in persistent memory [LWN.net]. https://lwn.net/Articles/610174/. (2014). [Online; accessed 2016/11/08].
[13] 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*. ACM, 15.
[14] H. Garcia-Molina, J. Ullman, and J. Widom. 2009. *Database Systems: The Complete Book* (2nd ed.). Pearson Prentice Hall.
[15] J. Izraelevitz, H. Mendes, and M. L. Scott. 2016. Linearizability of Persistent Memory Objects under a Full-System-Crash Failure Model. In *Proceedings of*

30th Intl. Symp. on Distributed Computing (DISC). ACM.

[16] C. Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Ph.D. Dissertation. Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

[17] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 1 (2010).

[18] 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. *SIGPLAN Not.* 49, 4 (Feb. 2014), 455–470. https://doi.org/10.1145/2644865.2541957

[19] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*. 216–223.

[20] Calif Milpitas. 2015. SanDisk and HP Launch Partnership to Create Memory-Driven Computing Solutions. https://www.sandisk.com/about/media-center/press-releases/2015/sandisk-and-hp-launch-partnership. (2015). [Online; accessed 2016/11/08].

[21] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. 401–410.

[22] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 12.

[23] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[24] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4-5 (2008), 465–480. https://doi.org/10.1147/rd.524.0465

[25] Andy Rudoff et al. [n. d.]. PMEM library website. http://pmem.io/nvml/. ([n. d.]).

[26] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. 298–307.

[27] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[28] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 14.

[29] 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 (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[30] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. 2017. Hardware Supported Persistent Object Address Translation. In *The Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46*.

[31] Matthew Wilcox. 2014. Support ext4 on NV-DIMMs. http://lwn.net/Articles/588218/. (2014). [Online; accessed 2016/11/08].

[32] Xiaojian Wu and AL Reddy. 2011. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 39.

[33] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: closing the performance gap between systems with and without persistence support. In *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*. 421–432.