Supporting Legacy Libraries on Non-Volatile Memory: A User-Transparent Approach

Chencheng Ye[†], Yuanchao Xu[‡], Xipeng Shen[‡], Xiaofei Liao[†], Hai Jin[†], Yan Solihin[§]
[†] National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[‡] North Carolina State University, Raleigh, North Carolina, USA
[§] Computer Science, University of Central Florida, Florida, USA
{yecc,xfliao,hjin}@hust.edu.cn, {yxu47,xshen5}@ncsu.edu, Yan.Solihin@ucf.edu

***Abstract*—As mainstream computing is poised to embrace the advent of byte-addressable *non-volatile memory* (NVM), an important roadblock has remained largely unnoticed, support of legacy libraries on NVM. Libraries underpin modern software everywhere. As current NVM programming interfaces all designate special types and constructs for NVM objects and references, legacy libraries, being incompatible with these data types, will face major obstacles for working with future applications written for NVM. This paper introduces a simple approach to mitigating the issue. The novel approach centers around *user-transparent persistent reference*, a new concept that allows programmers to reference a persistent object in the same way as reference a normal (volatile) object. The paper presents the implementation of the concept, carefully examines its soundness, and describes compiler and simple architecture support for keeping performance overheads very low.**

I. Introduction

Non-volatile memory (NVM) provides persistency, byte-addressability, and DRAM-like access latency and bandwidth, such that programmers can read and write data in NVM as if they were in DRAM without worrying about losing data upon system failure. It has drawn a lot of attention recently [1]–[7]. Cloud service providers have started adopting NVM in production environments [8], and observed up to 10X performance gains or 38% cost reduction [9] across a wide range of data center systems.

In this paper, we focus on a pressing issue that has not received prior attention but is going to manifest as a major roadblock for the adoption of NVM, supporting legacy library. Libraries, especially those low-level libraries, are the foundation underpinning modern computing. The Ubuntu 20.04 LTS official package archive, for instance, contains 6,505 libraries [10], including Boost [11], Gnulib [12], GTK [13], CPL [14], and so on. They constitute a layer indispensable in the execution stack of almost every mainstream computing system. They form a huge codebase. In particular, the four libraries contain 5.4 million lines of C/C++ code and 342 thousand of functions in total[1].

It is hence imperative to ensure that these libraries can still work with future NVM-based applications. While the community is poised to embrace NVM with architecture support [15]–[17] and new programming models for writing code for future NVM [18]–[20], this pressing issue has however remained largely unnoticed.

Doing nothing is not an option. If nothing is done, legacy libraries will not work for NVM applications. It is due to the required persistent data types in NVM programming models. The state-of-the-art persistent programming models [21]–[25] organizes persistent data in regions or pools; we call them *persistent memory objects pools (PMOPs)* in this paper. As PMOPs are long living, they are expected to be used by different programs, and in different runs they could be mapped to different virtual addresses. As a result, pointers contained in a persistent object must be relocatable, that is, even if the objects they point to are moved to a new address, from these pointers, the program can still reach those objects. They are called *persistent pointers*.

To support relocatability, persistent pointers are represented and implemented differently from conventional pointers. All existing programming models define persistent pointers with a new data type and dereference them through special APIs. For example, Intel PMDK uses 128 bits PMEMoid as persistent pointers [21] and NVHeap adopts a similar design [22]. Some other designs [26]–[28] pack the pool id and offset into an eight-byte pointer, but still require special types to be declared and used for referencing persistent objects via persistent pointers. We call this method *explicit persistent references.*

As a result, legacy libraries will face major obstacles for working with NVM applications, for they do not accept the new data types for persistent objects, nor have invocations of those special APIs when accessing an object that could reside on NVM [29]. As NVM starts getting into mainstream computing systems, this roadblock has already been manifested in the practical adoptions of NVM. For instance, the incompatible data types and virtual addresses have been recognized [29] as two of the main challenges for adopting NVM for a production key-value store, Redis. Skilled engineers at Intel spent several months on porting Redis to NVM [30]. According to the log of the git repository, they updated 4,348 lines of code, which

[1]reported by *cloc* and *ctags* tool, respectively.

constitute 7.6% of the codebase of Redis. The migration is incomplete [31], as data structures such as `zipmap` are still unavailable; hence more effort is expected for production use. Another research project [32] that migrates only the indexing data structure of another key-value store, rocksDB, adds 4,117 lines of code.

Principled requirements. A desirable solution to the problem must meet several principled requirements.

(i) Avoiding manual code rewriting whenever possible. Consider the opposite, which is to require all legacy libraries to be rewritten with the new programming models. The changes required in the rewriting would be massive: they have to be global as all object accesses are potential places where the changes must occur. More importantly, as a library is usually going to be used in many applications, the objects passed into it via parameters could be volatile in some invocations but persistent in some other calls. The uncertainty adds even more complexities in the rewriting. Given the large codebase of legacy libraries, manual code rewriting should be avoided whenever possible.

(ii) No reliance on the features that exist in only some programming languages. Generics (or templates), for instance, could potentially simplify the treatment to the data type uncertainty issue mention earlier; a recent study, AutoPersist [33], proposed the use of *Garbage Collection* (GC) to help mitigate software migrations to NVM. These solutions however are not applicable to the large volume of libraries written in C or other languages that do not support Generics or GC.

(iii) Accommodating various usage and manipulations of pointers. It is essential for the soundness of the solution, especially important for libraries written in low-level programming languages.

(iv) Low performance overheads and costs. The solution must keep the efficiency of the libraries as much as possible. Any architecture support should be minor.

Proposed solution. In this paper, we propose a solution to meet all four requirements. The basic idea is simple: Embedding the persistence/volatility type info of a pointer into the pointer itself, and using runtime checks to discern them for different treatments. But to realize the idea effectively, we have to address concerns on both soundness and efficiency. This paper presents our answers.

More specifically, this paper makes five contributions: (i) introduce the concept of *user-transparent persistent references* for effortless migration of legacy libraries to NVM; (ii) design the storage format of persistent pointers to keep their length the same as conventional pointers but at the same time using special bits to mark the persistence; (iii) systematically analyze the soundness of *user-transparent persistent references* by examining all kinds of pointer-related operations in C, the language that allows the largest freedom (and hence complexity) in pointer manipulations; (iv) create a simple architecture support that reduces the execution time overheads of *user-transparent persistent references* to a nearly negligible level; (v) empirically demonstrate the high efficiency of the implementation and the benefits in mitigating library migration complexities.

Experiments on a popular C/C++ library Boost and a case study on a machine learning application show that (i) *user-transparent persistent references* provides a sound solution to significantly mitigate the migration burden for programs and libraries; (ii) Hardware support for efficient user-transparent persistent reference is crucial. The pure software implementation slows down the original libraries by $2.75\times$ on average, while the hardware support removes nearly the entire performance loss. (iii) User-transparency does not necessarily sacrifice performance; compared to the state-of-the-art hardware supported for explicit persistent references, our solution even gains $1.33\times$ speedup on average, thanks to the reduced numbers of memory address translations the new method entails. (iv) The overheads incurred by migrating programs to NVM are minor (less than 5%) for all benchmarks, averaging only 2% with the architecture support.

To the best of our knowledge, this is the first solution proposed for supporting legacy libraries on NVM. Software developers can adopt the programming language support for legacy libraries by installing a compiler plugin and a persistent memory allocator. A legacy library only needs to go through a recompilation (or binary rewriting) before it can be invoked in an NVM application, either outside or inside a *persistent transaction*[2], which the user may rely on to achieve crash consistency. Although the needed architecture support adds some cost, it involves only minor changes to the store instruction; the benefit is however tremendous, considering the massive rewriting effort it could save to for a huge volume of legacy libraries.

In the rest of the paper, we first provide some background of this work in Section II, give a high-level view of the proposed solution in Section III, a careful analysis of the soundness in Section IV, the architecture-compiler support for efficiency in Section V-A, discussion of the relations with crash consistency and code optimizations in Section VI, experiments in Section VII, related work in Section VIII, and the conclusions in Section IX.

II. Background

Existing persistent programming models share lots of commonality. We give a brief explanation based on Intel `libpmemobj` [21]. Before using persistent memory, a programmer needs to create a persistent pool with a name. The pool is in the persistent memory. The operating system assigns a system-wide unique ID to the pool. Before reading from or writing to the pool, the programmer opens the pool with the name or the pool id, and the operating system maps the pool into a contiguous virtual address range in the process address space, which is similar to memory mapping a file into address space. Operating systems may map the pool to different addresses among executions of programs. However,

[2]Certainly, if a user would like to add persistent transactions into a library, she can still do that through manual or compiler-based code changes.

the offset from an object in the pool to the head of the pool stays the same. Figure 1 demonstrates the mappings.

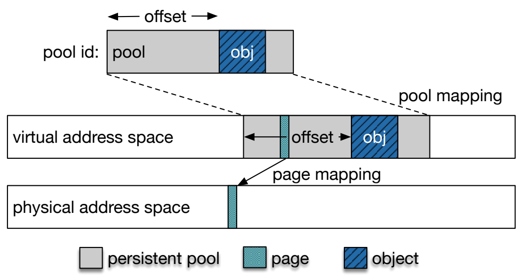


Fig. 1: Persistent pool based programming model

III. Solution at a High Level

As mentioned earlier, the representations of addresses in a persistent pointer have to differ from those in a volatile pointer, because those pointers have to stay valid even when the persistent objects are mapped to a different address in a new run. The principle of *user-transparent persistent references* is that despite the differences in representation, at the programming level, there shall be no difference when the program references a persistent object on NVM or a volatile object on DRAM.

The basic idea of our solution, *user-transparent persistent references*, is simple: embedding the type info (persistent or volatile) into the representation of pointers and then using runtime checks to discern the two types of pointers. The challenges are on ensuring both soundness and efficiency of the scheme. This section presents the solution at a high level. The next two sections focus on the soundness and efficiency respectively.

Memory Layout and Pointer Representation Figure 2 illustrates the virtual address space layout and the representation of pointers in our design.

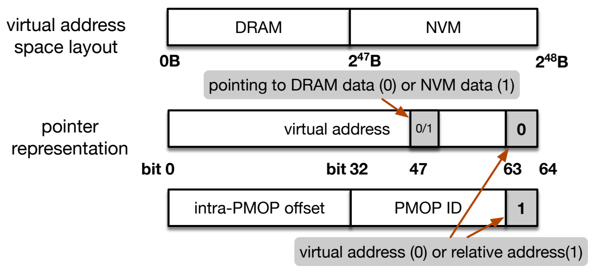

virtual address space layout
DRAM
NVM
0B
2^{47}B
2^{48}B
pointing to DRAM data (0) or NVM data (1)
pointer representation
virtual address
0/1
0
bit 0
bit 32
47
63
64
intra-PMOP offset
PMOP ID
1
virtual address (0) or relative address(1)

Fig. 2: Virtual address space layout and pointer representation

We divide the 256 terabytes virtual address of a process into two equal halves. The first half starting from byte 0 is dedicated to pages on DRAM while the second half is dedicated for pages on NVM. Given a virtual address, we determine if it points to NVM by checking the bit 47 rather than translating the address into physical address.

We use the most significant bit (MSB, bit 63) of a pointer to determine how the other bits shall be interpreted. When it is 0, the other bits constitute a virtual address corresponding to a regular/conventional pointer; for 1, a *relative address* corresponding to a persistent pointer. The relative address contains a 31-bit pool ID starting from bit 32 and a 32-bit intra-pool offset.

Runtime Checks Figure 3 uses pseudo-code to illustrate the runtime checks needed for interpreting the value of a pointer appropriately, and how it plays out in executing a pointer assignment statement. The checks are simply on the flagging bits in the pointer value. The two functions `va2ra()` and `ra2va()` are respectively translations between virtual address to relative address.

```
determineX(char* addr):              determineY(char* val):
    if bit 63 of addr is 1               if bit 64 of val is 1
        return NVM                           return Relative
    else                                 else
        if bit 47 is 1                       return Virtual
            return NVM
        else
            return DRAM
-----------------------------------------------------------------

pointerAssignment(char** to, char* p):
    if determineX(to) == NVM
        if determineY(p) == Relative → *to = p
        else → *to = va2ra(p)
    else
        if determineY(p) == Relative → *to = ra2va(p)
        else → *to = p
```

Fig. 3: Pseudocode showing the runtime checks to determine the right interpretation of a pointer value (top) and how it plays out in executing a pointer assignment statement. *va2ra(p)* translates virtual address to relative address, and *ra2va(p)* goes the other way.

Considerations There are two major concerns with the *user-transparent persistent reference* idea. The first is whether the removal of type differences at the programming level causes soundness issues to programs in all possible usage of pointers. The second is whether the overhead of the runtime checks can be minimized. Conducting the software checks at every pointer access would obviously incur large overhead. We discuss these two issues next.

IV. Soundness

To check the soundness in a complete manner, we go over ISO C11 standard [34] to enumerate all operations involving pointers, as shown in Figure 4. On each line, we list one type of operation allowed in C11 on pointers, and then provide the corresponding behavior semantics that C11 with *user-transparent persistent reference* will have. Wherever necessary, we use pxy to indicate the various kinds of pointers ($x = n, d$ for pointers on NVM or DRAM, $y = v, r$

T: any type; **I**: integer type; **i**: an integer variable; **$$**: the left hand side of an expression;
p: a pointer; **p.val**: the value of the pointer; **p.type**: the type of p; **p.va**: the virtual address of p; **p.ra**: the relative address of p;
pxy: a pointer with property x and y; **x**: **n** for the pointer is in NVM and **d** for DRAM; **y**: **v** for pxv.val is a virtual address and **r** for relative address; For example, **pxr** is a pointer either in NVM or DRAM and its value is a relative address;

operation	semantic
cast operator	
(T*)p	$$.val = p.val
(T*)i	$$.val = i.val
(I)pxv	$$.val = pxv.val
(I)pxr	$$.val = ra2va(pxr.val)
unary operator	
++p, --p, !p, ~p	$$.val = op p.val
&p	$$.val = p.addr
*pxv	$$ = *(pxv.val)
*pxr	$$ = *(ra2va(pxr.val))
sizeof p alignof p	$$.val = op p.type
assignment operator	
★ pny = pxv	$$.val = va2ra(pxv.val)
★ pny = pxr	$$.val = pxr.val
★ pdy = pxv	$$.val = pxv.val
★ pdy = pxr	$$.val = ra2va(pxv.val)
p = NULL	$$.val = NULL.val
pxy += i, pxy -= i	$$.val = pxy.val op i
logical operator: &&, ||	
p op i, i op p	$$.val = (I)p op i; $$.type = I
p op p'	$$.val=(I)p op (I)p'; $$.type=I
conditional operator	
p ? expr : expr	$$.val = (I)p ? expr : expr

operation	semantic
additive operator: +, -	
pxy op i	$$.val = pxy.val op i; $$.type = pxy.type
i + pxy	$$.val = i + pxy.val; $$.type = pxy.type
pxv - pxv'	$$.val = pxv.val op pxv'.val; $$.type = I
pxr - pxv pxv - pxr	$$.val = ra2va(pxy.val) op pxv.val; $$.type = I
pxr - pxr'	$$.val = pxr.val - pxr'.val; $$.type = I
postfix operator	
p++, p--	$$.val = p.val op
p[i],	$$.val = *(p+i)
p.identifier, p->identifier	$$.val = *(p + OffsetOf(identifier))
pxv(argument list)	$$.val = pxv.val (argument list)
pxr(argument list)	$$.val = ra2va(pxr.val) (argument list)
relational and equality operator: <, >, ≤, ≥, =, ≠	
pxv op pxv'	$$.val = pxv.val op pxv'.val
pxr op pxr'	$$.val = ra2va(pxr.val) op ra2va(pxr'.val)
pxr op pxv pxv op pxr	$$.val = pxv.val op ra2va(pxr.val)
pxv op i i op pxv	$$.val = pxv.val op i
pxr op i i op pxr	$$.val = ra2va(pxr.val) op i
p op NULL	$$.val op NULL

filled boxes for modified semantics

Fig. 4: A full list of operations ISO C11 standard allows on pointers and the corresponding semantics if *user-transparent persistent reference* is used.

for normal virtual address representation or relative address representation).

By checking the table against the specifications in ISO C11 standard, one can easily confirm that with *user-transparent persistent reference*, the returned value of every operation that involves pointers is consistent with the specifications in the ISO C11 standard, despite whether the actual representation of the pointer is a relative address format for a persistent pointer or an absolute address format for a volatile pointer. The reason is that the dynamic format checks automatically resolve the differences whenever necessary, as shown by the filled boxes. Intuitively, the hardware support converts between absolute addresses and the relative ones just before using/storing them. That means that we could safely convert a relative address to an absolute address to every NVM pointer before doing any operation to it. It is possible to do better than that in terms of performance by opportunistically keeping pointers as proper relative addresses so we do not have to convert them back before storage, as what our proposed scheme does. Preserving these relative pointers is “just an optimization”, so we can do it opportunistically without impairing the soundness.

V. Efficiency

Naively adding the dynamic checks at every pointer access would incur large time overhead and code size increase. We have explored two ways to reduce the overhead, one based on novel architecture support, the other based on pure compiler analysis and transformations.

A. Hardware-based Method

This hardware-based solution introduces a new instruction, some additional semantics to memory instructions, and some assistant hardware components. It dramatically reduces the overhead of the dramatic checks. The use of it requires only lightweight support of the compiler.

a) Instruction semantics: Our 64-bit pointers have two formats. A persistent pointer uses relative address format while conventional pointer uses virtual address format. For quick identification, the MSB (bit 63) distinguishes them ("1" for relative address, "0" for virtual address). Furthermore, our 48-bit virtual address space is split equally into volatile region (bit 47 is "0") and persistent region (bit 47 is "1"). Let us denote source and destination registers as Rs and Rd, respectively, and *ra2va* and *va2ra* as functions that convert relative (or virtual) address to virtual (or relative) address, respectively. Table I shows the semantics of memory reference instruction:

TABLE I: Instruction semantics

Instruction	Semantics	When Rx_y = 0, 1, or any (-)			
		Rd_{63}	Rd_{47}	Rs_{63}	Rs_{47}
load Rd, (Rs)	Rd = Mem[Rs]	-	-	1	-
	Rd = Mem[ra2va[Rs]]	-	-	0	-
storeD (Rd), Rs	Mem[Rd] = Rs	0	-	-	-
	Mem[ra2va[Rd]] = Rs	1	-	-	-
storeP (Rd), Rs	Mem[Rd]=va2ra[Rs]	0	1	0	1
	Mem[Rd]=Rs	0	1	1	-
	Mem[ra2va[Rd]]=va2ra[Rs]	1	-	0	1
	Mem[ra2va[Rd]]=Rs	1	-	1	-
	Mem[Rd]=Rs	0	0	0	-
	Mem[Rd]=ra2va(Rs)	0	0	1	-
	Error	other combinations			

For a *load* instruction, if Rs bit 63 is "1", the relative address must first be converted to virtual address before issued to the TLB or cache. Similarly, this applies to Rd for *storeD* instruction. Note that additionally there is a new store instruction (*storeP*). While *storeD* is a regular store instruction intended to store data into a memory location, *storeP* is intended to store a pointer value to a memory location. Our architecture provides two types of stores because for storing pointer value, the pointer may need to be converted into a relative address before written to memory. A program can use *storeP* to write an address value into a persistent memory location. For example, we can use *storeP* to replace `pointerAssignment` function presented in Figure 3 and *storeD* for `*(int*)p = i`.

The semantics for *storeP* largely reflects cases of whether or not Rs contains virtual address (hence needing conversion to relative address) or relative address, and whether or not Rd represents virtual address or relative address (hence needing conversion to virtual address).

b) Changes to the pipeline: Figure 5 (left) shows that we assume the compiler chooses *storeD* or *storeP* whenever it stores data or pointer to memory, respectively. Specifically, an LLVM optimization pass detects pointer operations and translates them to combination of LLVM IRs according to the semantics.

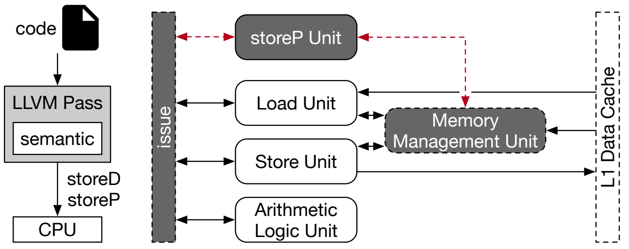


Fig. 5: Overview of compiler assumption (left) and modifications on CPU pipeline (right)

Figure 5 (right) shows the CPU pipeline modification. Load and Store functional units execute *load* and *storeD* instructions, respectively, as in a regular processor, but with an additional step to convert relative address to virtual address during effective address generation at the execute stage. To translate relative address to virtual address, we adopt a translation structure analogous to the page table and TLB called *persistent object table* (POT) and *persistent object lookaside buffer* (POLB) [26]. After translation, virtual address is used in the rest of the pipeline, hence mechanisms such as load bypassing, load forwarding, memory dependence checking, and speculation are not affected.

The *storeP* instruction requires additional processing to convert pointer format, so we introduce a separate functional unit for it. The storep unit must translate virtual to relative address. To achieve that, it relies on a finite state machine to keep track of the translation. The unit derives the pointer properties of its two operands with hardware logic implements the `determineX` and `determineY` in Figure 3.

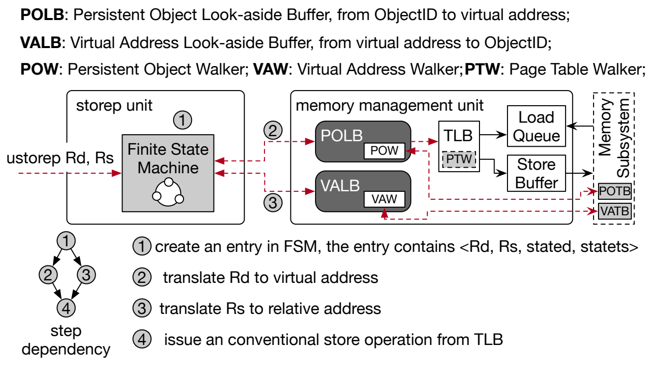

POLB: Persistent Object Look-aside Buffer, from ObjectID to virtual address;
VALB: Virtual Address Look-aside Buffer, from virtual address to ObjectID;
POW: Persistent Object Walker; VAW: Virtual Address Walker; PTW: Page Table Walker;
storep unit
ustorep Rd, Rs
Finite State Machine
memory management unit
POLB
POW
VALB
VAW
TLB
PTW
Load Queue
Store Buffer
Memory Subsystem
POTB
VATB
step dependency
1 create an entry in FSM, the entry contains <Rd, Rs, stated, statets>
2 translate Rd to virtual address
3 translate Rs to relative address
4 issue an conventional store operation from TLB

Fig. 6: Design of storep function unit and dataflow of operation $pny = pxv$. Solid filled blocks indicating the modified modules. Dashed arrows are for data path. *ustorep* refers to micro operation of *storep* instruction.

Since virtual to relative address conversion may incur a variable latency, we use a buffer to hold outstanding *storeP* instructions so that multiple instructions can be executed concurrently. Each buffer entry contains four fields: placeholders for virtual address in Rd and relative address in Rs, and the state of translations for Rs and Rd. Each entry has a finite state machine that keeps track of the progress of both translations. The *memory management unit* (MMU) is added two lookaside buffers to implement *ra2va* and *va2ra* translations using POLB [26] and the new *virtual address lookaside buffer* (VALB), respectively. VALB translates virtual address in Rs to relative address, in two steps: it retrieves PMO ID given a virtual address, and concatenates it with the offset portion of the virtual address.

Each VALB entry has three fields: PMO starting address (64 bits), PMO size (32 bits), and PMO ID (32 bits). VALB may use a *ternary content addressable memory* (TCAM) to find an entry with the longest prefix match, allowing quick retrieval of PMO ID. Just as POLB is backed by kernel table POTB [26], VALB is backed by a kernel table called *virtual address table* (VATB). Since PMOs have different sizes and starting addresses, VATB adopts a B-Tree based range table structure that was proposed for Range TLB [35]. Note that POLB and VALB do not contain permission bits as permission control is already enforced by the TLB [36].

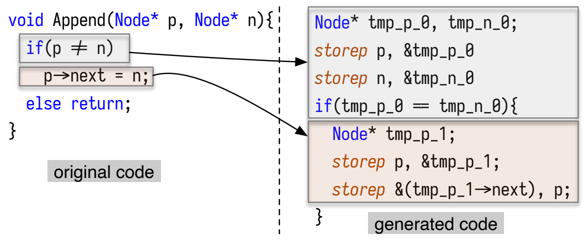
When a *storeP* instruction is executed, the bottom portion of Figure 6 illustrates the steps tracked by the state machine. Both Rd and Rs are translated (if needed) to virtual address and relative address, respectively, simultaneously. Once both are completed, the virtual address can be forwarded to the TLB for permission check. In addition to protection and permission fault, *storeP* may incur a fault for errors listed in Table I).

c) Hardware complexity and storage overheads: The storeP unit acts as a functional unit which could have its own reservation stations, hence it does not affect the critical path delay of other instructions. The POLB and VALB are accessed prior to the TLB hence they add small delay to the critical path of address translation in the MMU (modeled in the simulation). Some prediction mechanisms can be deployed to accelerate this, to predict non-PMO accesses that bypass the POLB/VALB, but we leave this out for future work. The total on-chip storage costs for the hardware structures (FSM, POLB, and VALB) is minor, at less than 2KB, as detailed in Table II. Others (POTB and VATB) are software data structures in kernel memory. We use Cacti [37] to evaluate the die area needs with 45nm process. The total on-chip storage consumes only 0.059% die area of a 45nm octal core Nehalem processor [38].

TABLE II: Space cost of hardware implementation

Structure	Entry Size (Bytes)	Num Entries	Total (Bytes)	Area (mm^2)
FSM	16	32	512	0.0205
POLB	12	32	384	0.0137
VALB	12	32	384	0.0137
Total size: 1,280 bytes; Total area: 0.0479 mm^2				

d) Compiler support: Unlike in the software solution, with the architecture support, the compiler's role changes to: (1) generating *storeP* instruction for assignment operations rather than calling functions; (2) generating *load* and *storeD* instructions for data load and assignment as the instructions now accept *user-transparent persistent references* as input; (3) leveraging *storeP* to convert a relative to virtual address. Figure 7 illustrates an example linked list Append() function utilizing *storeP* instructions, inserted by an LLVM pass for hardware supported user-transparent persistent reference. The LLVM pass for hardware supported user-transparent persistent reference is invoked after all other optimization passes as none of them recognizes *storeP* instruction.



```
void Append(Node* p, Node* n){
  if(p ≠ n)
    p→next = n;
  else return;
}
```

```
Node* tmp_p_0, tmp_n_0;
storep p, &tmp_p_0
storep n, &tmp_n_0
if(tmp_p_0 == tmp_n_0){
  Node* tmp_p_1;
  storep p, &tmp_p_1;
  storep &(tmp_p_1→next), p;
}
```

Fig. 7: Generated code for linked list appending function

The first two *storeP* instructions store pointer `p` and `n` into temporary variables `tmp_p_0` and `tmp_n_0`. Since stack variables are in volatile memory, according to the semantic for pointer assignment $pdy = pxr$, the *storeP* instructions translate `p` and `n` into virtual addresses if they are relative addresses.

B. Compiler-based Method

Besides the hardware-based method, we have also explored the use of pure compiler-based static analysis to infer the actual types of the pointers in a program, and hence avoid insertions of the checks at the places where the actual type of the pointer can be precisely determined.

Compiler-based type inference is a well-studied topic in programming languages. For this task, it is possible in some cases based on the contexts. For example, for statement `void* p = pmalloc(1024)`, the compiler can determine that: (1) p must be on DRAM as it is a stack variable, and (2) the persistent memory allocation function must return a relative address per its definition. It is however not always possible for C programs due to the difficulties in function parameter passing and point-to aliases and hence cannot remove all dynamic checks.

Figure 8 outlines this approach. We implement a pass in LLVM to conduct the inference based on its type inference module. The pass first marks a predefined set of functions accepting or returning relative addresses, including `pmalloc` and `pfree`. Starting from the arguments or return values of the functions, it uses backward dataflow analysis to propagate the properties to other variables as much as possible. The propagation follows the pointer properties change of the semantic in Figure 4. For example, assigning a pxr pointer to a stack variable makes the variable a pxv pointer.

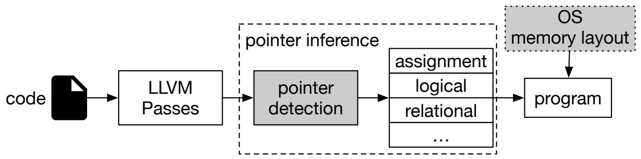
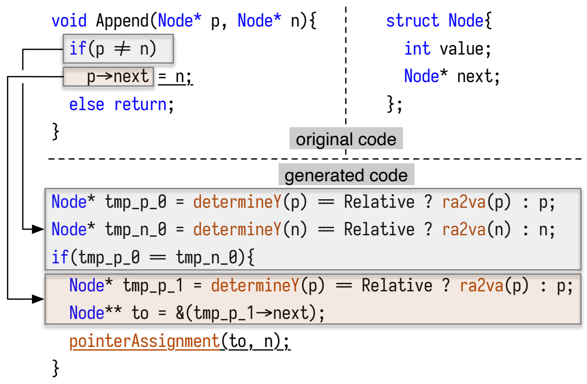


Fig. 8: Components and workflow of the compiler-based approach

For the pointer variables that compiler cannot determine properties at compile time, the compiler inserts the dynamic checks. Figure 9 demonstrates the generated code for a linked list appending function. For the relational operation `p != n`, the compiler inserts `determineY` for the pointers on both sides and converts them to virtual addresses accordingly; for assignment operation, the compiler inserts `pointerAssignment` function call.



```
void Append(Node* p, Node* n){          struct Node{
  if(p != n)                              int value;
    p->next = n;                          Node* next;
  else return;                          };
}
```

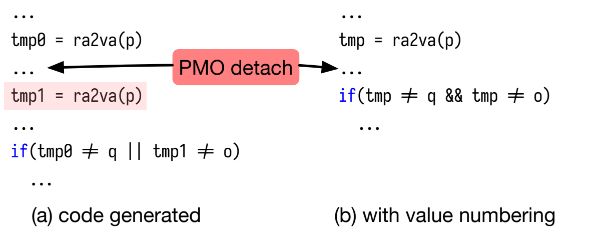
```
Node* tmp_p_0 = determineY(p) == Relative ? ra2va(p) : p;
Node* tmp_n_0 = determineY(n) == Relative ? ra2va(n) : n;
if(tmp_p_0 == tmp_n_0){
  Node* tmp_p_1 = determineY(p) == Relative ? ra2va(p) : p;
  Node** to = &(tmp_p_1->next);
  pointerAssignment(to, n);
}
```

Fig. 9: Linked list appending function generated by the LLVM pass

VI. Discussion

With the solution proposed in this work, to make a legacy library callable in an NVM application, it just needs to go through a recompilation (or binary rewriting) such that the `store` instructions in the code can be replaced with either `storeD` or `storeP` instructions. The library functions can then be invoked in the application, regardless of whether the objects passed into the library function are persistent or volatile. If the call is enclosed in a persistent transaction in the application code, it will be the job of the compiler of the application program to insert necessary runtime logging instructions into the application and the library function to ensure crash consistency. Obviously, our solution does not prevent a user from changing the library if she would like to take a full advantage of NVM by, for instance, adding persistent transactions inside the library.

Both software and hardware implementation insert the code generation pass after all code optimizations. The order prevents scalar optimizations, such as value numbering, lazy code



```
...                                     ...
tmp0 = ra2va(p)                         tmp = ra2va(p)
...                                     ...
tmp1 = ra2va(p)                         if(tmp != q && tmp != o)
...                                       ...
if(tmp0 != q || tmp1 != o)
  ...
```

Fig. 10: Generated code for p != q && p != 0. Pool detach causes inconsistent program semantics.

motion, or common subexpression elimination. For example, for `p != q && p != o`, the software user-transparent persistent reference compiler may generate a `ra2va(p)` function call for each `p` of the two statements. Trivial value numbering can easily remove the redundant `ra2va(p)` if it was employed after the user-transparent persistent reference pass.

However, changing optimization order results in inconsistent program semantic. Figure 10 illustrates a case that the pool associated with p detached [39] during program execution. The code generated for the software method interrupts at the second `ra2va` function call while the value-numbering optimized code executes normally.

The two methods can be potentially used together. The compiler conducts type inference, and at places where it is uncertain about, inserts the extended hardware instructions rather than software-based dynamic checks. Our experiments reported next however show that it is not necessary thanks to the superior performance of the hardware-based method.

Under the proposed solution, considering that most hardware prefetchers use physical addresses [40], they work as usual as our proposal does not change data placement in the physical address space. Virtual address based stride prefetchers [41] may lose performance when data in persistent memory pools are mapped to distributed virtual addresses. It is however the consequence of persistent memory pool address mapping introduced by the persistent memory programming model [21] rather than by our proposed techniques.

VII. Evaluation

This section evaluates the soundness and effects of the proposed technique from four aspects:

- Soundness: the semantic of the program must be preserved; the pointers in NVM shall hold the correct relative addresses.
- Productivity: the number of lines needs to be modified for libraries or applications to migrate them to NVM shall be largely reduced.
- Performance: the performance of the program generated with the proposed techniques shall be comparable if not better than the default *explicit persistent reference* cases.
- Breakdown: the sensitivity analysis to latency of new hardware unit VALB and the breakdown of the cost and benefits.

A. Methodology

Benchmarks As the complexities and compatibility of libraries for NVM is a main motivation of this work, we focus our evaluations on commonly used C/C++ libraries. As none of the existing NVM benchmarks are designed for evaluating library performance, we developed a benchmark suite ourselves to do the study.

Specifically, we concentrate on Boost [42], a popular set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudo-random number generation, multithreading, image processing, regular expressions, and unit testing. Table III lists the six commonly used data structures in Boost that our experiments focus on. Together they have 22,206 lines of source code.

To measure the performance of the libraries, we need a harness to invoke them. We create the harness based on Intel PMDK [43], a key-value store framework for NVM. The core operations in the harness are mapping keys to value records. When evaluating one of the data structures listed in Table III (except `LL`), the harness replaces the mapping scheme with the corresponding implementation in the data structure of interest and then runs a number of key-value store operations and measures the performance.

The harness uses YCSB [44] to generate the workload for the key-value store. Specifically, we use *workloadd* (one of the preset workloads) in YCSB to generate 10,000 key-value pairs and 100,000 GET and SET operations. Both key and value are 8 bytes text string. 95% of the operations are GET and 5% are SET. All SET operations insert new key-value pairs into the key-value store such that the key-value store updates the indexing data structures and pointers. The operations access keys with *latest* distribution, which is zipfian distribution with more recently inserted records are more likely to be read.

All of the data structures in Table III can play in the role of mapping and hence are compatible with the key-value store harness, except the linked list `LL`. We hence create a separate harness for evaluating `LL`, which generates 10,000 nodes with each node containing two pointers and a 16-byte randomly generated integer value, and then iterates the linked list and accumulates the values.

We in addition conduct a detailed case study on a machine learning application `KNN` from a machine learning algorithm collection MLPack [45], which also uses two other C/C++ libraries, Armadillo [46] and Boost [42]. It implements the *k-nearest neighbor* (KNN) algorithm that is widely used in data mining. We use `iris` dataset [47] as input. The dataset contains 150 samples of three types of iris plants.

We persist all the content of six data structures by explicitly specifying pmalloc as the memory allocator, while leaving other data volatile. Each data structure requires only one line of code modification in the application; no code change is needed in the Boost library. On the KNN benchmark, we persist two matrices with two lines of code modified; the modification is the in the KNN application rather than the MLPack, Armadillo, or Boost libraries.

TABLE III: Benchmarks in use

Benchmark	Description
Implemented with Boost Library	
LL	linked list with 10,000 nodes inserted and iterated 10 times.
Hash	chained hash table. Number of buckets equals to the number of keys.
RBTree	red-black tree, a self-balancing binary tree with wide range of applications.
AVLTree	AVL tree, used for database transactions that frequent lookups are required.
Splay	Splay tree, used in garbage collector that same element is retrieved multiple times in a short period.
SG	Scapegoat tree, a self-balancing tree with low space cost and high rebalancing performance.
Implemented with MLPack, Armadillo, and Boost	
KNN	k-nearest neighbor algorithm widely used for object classification in data mining.

Hardware Simulator We evaluate all implementations with an interval-based timing accurate hardware simulator, Snipersim [48]. The simulator uses Pin as the frontend. We implement the new instructions as magic instructions in Snipersim and add latency to them. Table IV lists the simulator parameters. We set the capacity and latency of POLB according to the original work [26], [49]. We use CACTI [37] to ensure that the latency fit into the number of cycles. For VALB, by default, we use the same latency as POLB; we conduct separate sensitivity studies on the influence from its latency.

TABLE IV: Simulated architecture

Component	Parameter
ISA	64-bit X86, Gainestown architecture
CPU	1 core, 2.66Ghz, 64B cache line
Branch predictor	Pentium M, miss penalty 8 cycles
L1 data TLB	4-way, 64 entries, 1 cycle
L2 shared TLB	4-way, 1536 entries, 7 cycles for hit, 30 cycles for miss
L1 data cache	8-way, 64 entries, 4 cycles
L2 cache	8-way, 256KB, 12 cycles
L3 cache	8-way, 2MB, 40 cycles
Memory	120 cycles (45ns) for DRAM, 240 cycles for NVM
Buffer	
POLB	3 cycles for hit, 30 cycles for miss
VALB	3 cycle for hit, 30 cycles for miss

a) Versions to Compare: In the performance comparison, we focus on the following versions:

- **HW:** the version of the NVM program written with our hardware-based *user-transparent persistent references*.
- **SW:** the version of the NVM program written with our compiler-based *user-transparent persistent references*.
- **Explicit:** the version of the NVM program written with a representative *explicit persistent references* programming model [26]. In this version, special APIs are used in the program for accessing NVM data objects. At every access, the object ID (i.e., relative pointers) is converted to the virtual address through a hardware extension.
- **Volatile:** the version of the native implementation of the program with normal pointers without considerations

of NVM. This version is subject to no NVM-caused overhead. It cannot work on real NVM systems, but offers a clean reference point for examining the overhead of the other versions.

B. Soundness Evaluation

We first check the soundness of the *user-transparent persistent references*. We examine the execution of programs using both the proposed HW and SW versions empirically, including both whether their outputs are consistent with the outputs of the original programs, and the correctness of every pointer in the data structures. We confirm that all the programs outputs are correct. Moreover, all the pointers in the persistent objects are in the correct relative pointer format and carry the correct values throughout the entire executions of the programs. The results confirm the analytical results in Section IV.

Moreover, we test the soundness of the technique on a production-quality LLVM test-suite [50]. The test-suite contains test programs written in C, C++, bitcode, and LLVM *intermediate representation* (IR). We use only C test programs; there are 267 application tests and 1518 regression tests (1483 from gcc-torture test-suite [51]). We use the SW version, *user-transparent persistent references*, to test the semantics of pointer operations. All tests were passed.

To use persistent memory, we persist the entire heap by replacing the default memory allocator with libvmmalloc [52], a library that transparently overrides the malloc function to allocate memory on NVM; the stack memory remains volatile. It is a common practice of prior studies [7], [31], [53], [54].

C. Performance Evaluation

Figure 11 reports the execution time of the programs normalized to the *volatile* time. We make the following observations from the figure.

(1) The *HW* version outperforms the *explicit* version by 1-3X. This observation was initially surprising to us. We expect to see productivity improvement over *explicit* methods, but not speedups. After a detailed analysis, we found out the reason. The *explicit* version, due to the use of the special APIs and object IDs rather than pointers, requires address conversion at every access to a persistent object. In contrast, with the *HW* version, the conversion results are naturally assigned to normal pointers and hence get reused in later accesses to the object through those pointers. Figure 12 illustrates it through an example codelet. This benefit comes as a side effect of the removal of the differences of persistent and volatile pointers at the programming level.

(2) The *SW* version suffers from significant time overhead. Although the compiler-based type inferences try to avoid the insertions of dynamic checks whenever possible, the complexities in pointer and alias analysis leave a substantial amount (about 42%) of dynamic checks in the code. These dynamic checks contain conditional statements that add lots of branch mispredictions. Figure 13 reports the number of branch mispredictions normalized to those from the *volatile* version. The branch mispredictions in the *SW* version is $6.7 - 2944\times$

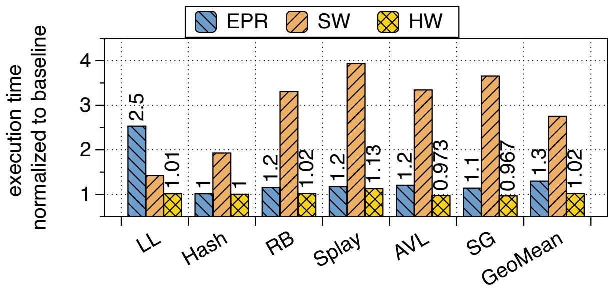
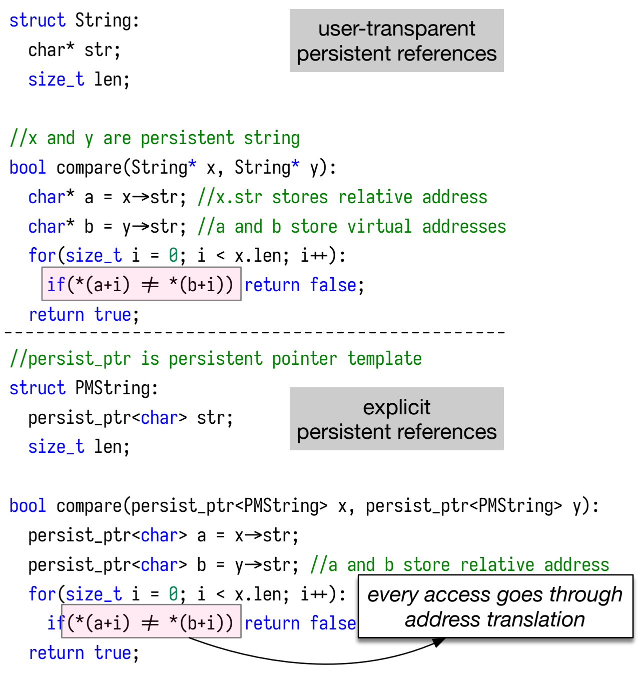


Fig. 11: Simulated execution time normalized to native execution. EPR denotes explicit persistent references. Less is better.



user-transparent persistent references
explicit persistent references
every access goes through address translation

```
struct String:
  char* str;
  size_t len;

//x and y are persistent string
bool compare(String* x, String* y):
  char* a = x->str; //x.str stores relative address
  char* b = y->str; //a and b store virtual addresses
  for(size_t i = 0; i < x.len; i++):
    if(*(a+i) != *(b+i)) return false;
  return true;
------------------------------------------------
//persist_ptr is persistent pointer template
struct PMString:
  persist_ptr<char> str;
  size_t len;

bool compare(persist_ptr<PMString> x, persist_ptr<PMString> y):
  persist_ptr<char> a = x->str;
  persist_ptr<char> b = y->str; //a and b store relative address
  for(size_t i = 0; i < x.len; i++):
    if(*(a+i) != *(b+i)) return false
  return true;
```

Fig. 12: (a) A codelet in *user-transparent persistent references*, where, the accesses to the persistent object naturally reuse the virtual address of the object attained at the first conversion. (b) A codelet in *explicit persistent references*, where, every access to the persistent object need to go through a conversion from relative address to the virtual address.

higher compared to the *HW* version. Its overheads on `LL` and `Hash` are relatively smaller because these two programs have relatively less data locality, and hence the inserted dynamic checks weigh less in the overall time.

(3) With the hardware support, the *user-transparent persistent pointers* remove almost all the overhead incurred by the use of persistent pointers. The largest overhead is on `Splay`, about 12%, compared to the *volatile* case where no persistent pointers are used.

Table V shows the number of dynamic checks and conversions.

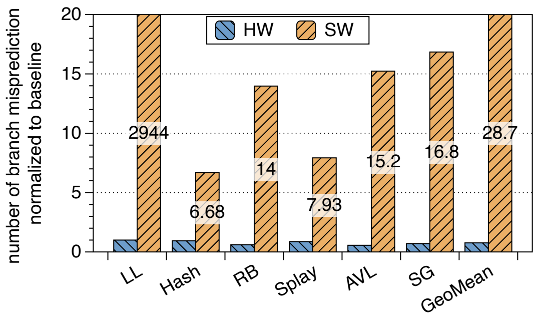


Fig. 13: The number of branch mispredictions, normalized to those of the *volatile* version. Lower is better.

TABLE V: The number of dynamic checks, conversions from absolute address to relative, and vice-versa for each benchmark. abs. refers to absolute address, rel. refers to relative address.

Benchmark	dynamic checks	abs. to rel.	rel. to abs.
LL	8,200,020	199,999	3,999,960
Hash	2,577,030	29,944	444,844
RB	14,518,568	50,805	8,392,150
Splay	25,572,050	850,878	10,949,713
AVL	14,407,532	55,327	8,287,941
SG	18,137,435	29,885	11,830,136

D. Sensitivity Analysis to VALB Latency

The latency of VALB and VAW generates marginal impact on performance. Even with 50 cycles of latency set for VALB and VAW, which implies that every three VALB lookups access memory once, the execution time of all benchmarks increases by less than 10%. Figure 14 shows the execution time normalized to the execution time of explicit persistent references.

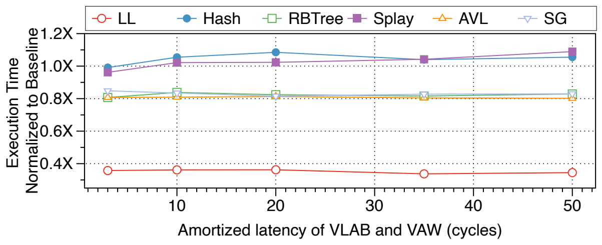


Fig. 14: Execution time to amortized latency of VALB and VAW, normalized to the execution time of explicit persistent references

Figure 15 demonstrates the reason for the observed small impact. Only 0.38% of memory access instructions are *storeP* ; only 0.22% accesses VALB or VAW. In comparison, 12.6% of memory accesses access POLB and POW.

E. Case Study with KNN on Productivity and Performance

In this part, we provide a detailed case study on KNN, which uses multiple libraries including Armadillo [46] for matrix creation, and Boost [42] for computation.

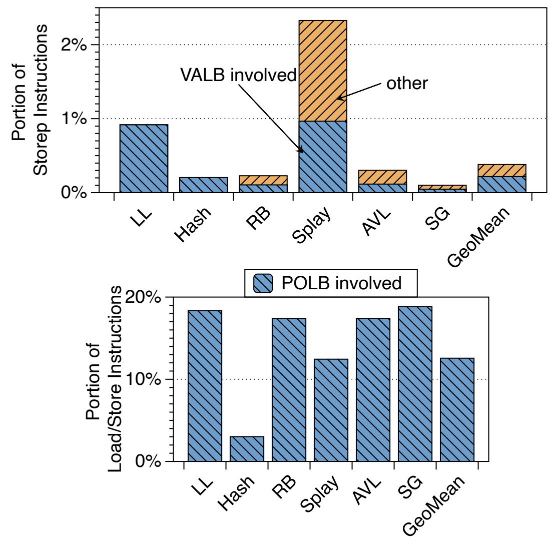


Fig. 15: Portion of load/store instructions that involve VALB or POLB.

The KNN algorithm uses four matrixes, one for input, one for internal uses, and two for outputs. Each matrix is a compound data structure defined by the Armadillo library. A matrix contains a pointer to a data array and a set of metadata, which indicate for instance whether the matrix is column-major or row-major. Any of the matrices can be in NVM or DRAM. For example, another data collection program forwards its output to KNN, or KNN persistifies its output. If a matrix is in NVM, the pointer in it should be converted to persistent pointer. We allocate all matrices on NVM except for the input matrix. Instead of interacting directly with the matrices, the Boost library takes an object as input and encases a matrix.

We assume that all matrices need to be persistified. With our proposed technique, only seven lines of the code need to be modified, to replace malloc and free with the persistent versions (which can be automated by a compiler). In the *explicit* version, in contrast, the changes include 863 lines of code, more than 10 data objects, and over 32 functions.

As KNN itself may serve as a machine learning library function, to make it usable for all the possible situations on a system equipped with both DRAM and NVM, it would need to be able to handle any combination of the data objects in terms of their placements on DRAM or NVM. The four matrices correspond to 16 possibilities. In the *explicit* case, 16 versions of the KNN would need to be created. The total number of lines of code that needs to be changed quickly grow to thousands. All the resulting code versions would then need to be maintained in the future.

The performance of the *HW* version has marginal difference from that of the baseline as only 0.22% of the load instructions incur address translation. The *SW* version sees 7.56X slowdown.

TABLE VI: Related work

programming framework	language	user-transparent persistent references?	allow object relocation?	relocation overhead	solution for relocation
Espresso [19]	Java	Yes	Yes	High	update all pointers via pointer tracing
AutoPersist [33]	Java	Yes	Yes	High	update all pointers via pointer tracing
go-pmem [55]	Go	Yes	Yes	High	update all pointers via pointer tracing
PMDK [21]	C/C++	No	Yes	Low	position independent pointer (fat pointer)
Breeze [20]	C++	No	Yes	High	update all pointers via pointer tracing
Atlas [56]	C	Yes	No	–	No
iDO [57]	C++	Yes	No	–	No
Our work	C/C++	Yes	Yes	Low	position independent pointer (8 bytes pointer)

F. *Overall Observations*

Overall, through the case study, we conclude the following:

- *User-transparent persistent references*, with either HW or SW support, do not affect the soundness of programs.
- *User-transparent persistent references* remove most of the programming burdens in migrating code and libraries to NVM.
- The proposed *hardware-based support* significantly improves the performance of the NVM programs compared to the use of *explicit persistent pointers*, reducing the execution time overheads to a nearly negligible level.

VIII. Related Work

While industry recently finds a single type system for persistent and volatile objects is essential for practical persistent programming [55], legacy libraries expectedly benefit from such type system as they can run without distinguishing persistent and volatile pointers. However, existing proposals provide limited supports, summarized in Table VI.

Espresso [19], AutoPersist [33], and Go-pmem [55] mitigate explicit persistent references for managed languages such as Java and Go. While the semantic of Java or Go reference (pointer) operations are much simpler than that of C as Java or Go does not support pointer casting to integer or most of the other operations supported in C, the major obstacle for those techniques to be applied for C-based libraries is that they depend on garbage collection to enable reusability of persistent objects, which has been proposed by a number of prior studies [21], [22], [26], [27], [58], [59].

In particular, Espresso [19] uses conventional pointers for persistent objects but scarifies security on address space layout randomization. Its Java virtual machine loads persistent heap and maps to the same address space among different programs or executions. If the mapping fails as the address space is occupied, Espresso map the persistent heap to other address space and updates all pointers within that persistent heap.

AutoPersist [33] uses object reachability as a criterion of whether an object should be persistent. It automatically moves objects from DRAM to NVM with JVM and updates the pointers to the objects, hence it does not need special pointer type for persistent objects at programmers' end. However, it applies only to managed programming languages running in a virtual machine. Further, it is unclear how it tackles cross-run object reusability problem.

Early proposals such as Atlas [56] and iDo [57] disallow NVM objects to be mapped to different addresses in different runs.

For the restrictions and loss of security protection opportunities, most other studies assume that NVM objects can be mapped to different addresses in different runs. NV-Heaps [22] and PMDK use 128 bits fat pointer composed of 64 bits pool ID and 64 bits intra-pool offset. Every pool hence can be up to 2^{64} bytes. Chen et al. [27] point out that fat pointer incurs substantial overhead. They proposed 64 bits position-independent RIV pointer and software solution to translate from RIV pointer to virtual address. Wang et al. [26] proposed hardware acceleration for 64 bits relative pointers that are composed of 32-bit pool ID and 32-bit intra-pool offset. They use special instructions to access data through relative pointers. All of them require programmers to use a special pointer type and APIs for persistent objects, belonging to the *explicit* category as evaluated in this work.

Chen et al. [27] proposes a type system for conversion between relative pointers and conventional pointers. The semantics proposed in this work provides user-transparent addresses conversion and reusability, which selects correct form for the pointer to be retained in NVM. Cai et al. [23] implements the pointer conversion of offset-based pointer, which is called off-holder by Chen et al. [27]. Twizzler [24] requires programmers to call functions to explicitly dereference a relative pointer or convert a conventional pointer to a relative pointer. Those works can be enhanced with the semantics of pointer operations proposed by this work to implement compiler support to automatically insert those dereferences and conversions.

Proposals for C/C++ [20], [60] use a separate type system for persistent and volatile objects. Breeze [20] requires programmers to annotate persistent pointers and objects. PMDK [60] provides a persistent pointer template, *persist_ptr*, that offers identical interfaces for operations persistent pointers and normal pointers. As such, programmers define persistent pointers as an instance of the *persist_ptr* template and later use the persistent pointers as if they were normal pointers. However, this method requires programmers to define persistent pointer as a type other than that of normal pointer and is hard to materialize for programming languages such as C and Go where template is absent.

The idea of using different pointers has been explored in the context of managed language, such as generational garbage collection [61] and persistent object stores [62], [63]. The assumptions are different from those of persistent pointers in other languages. For example, C/C++ allows more kinds of pointer operations, lacks rich runtime support, and outweighs performance consideration.

A relevant technique in implementing user-transparent persistent pointer is Java read and write barriers [64]. The technique is orthogonal to the semantics of pointer operations or hardware support proposed by this work. In particular, the read and write barriers are functions that Java compiler or runtime transparent inserted into a program to instrument every memory accesses. The content of the functions is defined by developers of Java runtime or compiler. One can easily implement the semantics of pointer operations proposed by Section IV as Java read and write barriers. Furthermore, such software scheme suffers from substantial runtime overhead. For reference, the *conditional read barrier* implemented with only two assembly instructions can slowdown a Java program by 10% [64]. The operations on user-transparent persistent pointer are complex and apply for both read and write memory accesses, which makes the hardware support an appealing solution to preserve the performance.

IX. Conclusion

As libraries form a fundamental layer in the modern computing stack, the lack of support of legacy libraries on NVM would pose a major obstacle for broad practical adoptions of NVM. This paper has introduced the first known solution to the problem. It introduces *user-transparent persistent reference*, a concept for effortless migration of legacy libraries to NVM. It carefully analyzes the soundness of the design against all kinds of pointer-related operations. It proposes a lightweight architecture support to eliminate the runtime overhead of *user-transparent persistent reference*. It reports the effectiveness of the solution on a popular C/C++ library Boost and a case study on a machine learning application. The results show that the proposed solution is promising in removing the major roadblock that legacy libraries pose for the adoption of NVM.

X. Acknowledgement

We thank anonymous reviewers for their feedback. This work is supported by the National Key Research and Development Program of China under grant No.2017YFB1001603, the National Natural Science Foundation of China under grant No. 61832006, 61825202, 61702202, and the National Science Foundation (NSF) under Grants CNS-1717425 and 16406268. 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] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.

[2] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "SLM-DB: single-level key-value store with persistent memory," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019, pp. 191–205.

[3] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[4] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory," in *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems*, 2017.

[5] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[6] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019, pp. 31–44.

[7] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 462–477.

[8] "Intel persistent memory partners," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[9] "Gain with intel persistent memory," https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[10] "Library section in ubuntu 20.04 lts packages," https://packages.ubuntu.com/focal/.

[11] "Boost C++ Library," https://www.boost.org/.

[12] "The gnu portability library," https://www.gnu.org/software/gnulib/.

[13] "GTK library," https://www.gtk.org/.

[14] "Common pipeline library," http://www.eso.org/sci/software/cpl/.

[15] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: durable hardware transactional memory," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 452–465.

[16] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015, pp. 672–685.

[17] M. Cai, C. C. Coats, and J. Huang, "Hoop: efficient hardware-assisted out-of-place update for non-volatile memory," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020, pp. 584–596.

[18] *SNIA NVM Programming Model*, https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf.

[19] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "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*, 2018, pp. 70–83.

[20] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *Proceedings of the IEEE 36th International Conference on Computer Design*, 2018, pp. 413–422.

[21] *Intel PMDK*, https://pmem.io/pmdk/libpmemobj/.

[22] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 105—118.

[23] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020, pp. 60–73.

[24] D. Bittman, P. Alvaro, P. Mehra, D. D. Long, and E. L. Miller, "Twizzler: a data-centric OS for non-volatile memory," in *Proceedings of the USENIX Annual Technical Conference*, 2020, pp. 65–80.

[25] Y. Solihin, "Persistent memory: Abstractions, abstractions, and abstractions," *IEEE Micro*, vol. 39, no. 1, pp. 65–66, 2019.

[26] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 800–812.

[27] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu, "Efficient support of position independence on non-volatile memory," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 191–203.

[28] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020, pp. 680–692.

[29] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 427–439.

[30] "Pmem-redis," https://github.com/pmem/pmem-redis.

[31] W. Zhang, S. Shenker, and I. Zhang, "Persistent state machines for recoverable in-memory storage systems with NVRAM," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 1029–1046.

[32] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proceedings of the USENIX Annual Technical Conference*, 2020, pp. 17–31.

[33] T. Shull, J. Huang, and J. Torrellas, "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*, 2019, pp. 316–332.

[34] *C11 Standard*, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf.

[35] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 66–78.

[36] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall, CRC Computational Science, 2015, ISBN-13 978-1482211184.

[37] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[38] "Nehalem," https://en.wikichip.org/wiki/intel/microarchitectures/nehalem_(client).

[39] Y. Xu, Y. Solihin, and X. Shen, "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*, 2020, pp. 987–1000.

[40] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016, pp. 1–12.

[41] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, pp. 1–67, 2014.

[42] O. Krzikalla and I. Gaztanaga, "Boost intrusive containers." https://www.boost.org/doc/libs/1_70_0/libs/histogram/doc/html/histogram/overview.html, online; accessed August, 2020.

[43] "Persistent key-value store from pmdk," https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/map.

[44] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[45] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang, "mlpack 3: a fast, flexible machine learning library," *Journal of Open Source Software*, vol. 3, p. 726, 2018.

[46] C. Sanderson and R. Curtin, "Armadillo." http://arma.sourceforge.net/speed.html, online; accessed August, 2020.

[47] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[48] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, 2014.

[49] T. Wang, S. Sambasivam, and J. Tuck, "Hardware supported permission checks on persistent objects for performance and programmability," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 466–478.

[50] "LLVM test-suite 11.0," https://llvm.org/docs/TestSuiteGuide.html.

[51] "Gcc torture," https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite/gcc.c-torture.

[52] "Intel libvmmalloc," https://pmem.io/vmem/libvmmalloc/.

[53] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *Proceedings of the USENIX Annual Technical Conference*, 2020, pp. 813–827.

[54] M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank, "Nvtraverse: In NVRAM data structures, the destination is more important than the journey," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 377–392.

[55] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam, "go-pmem: Native support for programming persistent memory in GO," in *Proceedings of the USENIX Annual Technical Conference*, 2020, pp. 859–872.

[56] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[57] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 258–270.

[58] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 91—-104.

[59] D. Bittman, P. Alvaro, D. D. Long, and E. L. Miller, "A tale of two abstractions: the case for object space," in *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, 2019, pp. 11–11.

[60] *persist_ptr from PMDK*, https://pmem.io/libpmemobj-cpp/master/doxygen/classpmem_1_1obj_1_1persistent__ptr.html.

[61] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," *ACM Sigplan notices*, vol. 19, no. 5, pp. 157–167, 1984.

[62] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, "An orthogonally persistent Java," *ACM Sigmod Record*, vol. 25, no. 4, pp. 68–75, 1996.

[63] A. Dearle, G. N. Kirby, and R. Morrison, "Orthogonal persistence revisited," in *Proceedings of the International Conference on Object Databases*. Springer, 2009, pp. 1–22.

[64] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, "Barriers reconsidered, friendlier still!" in *Proceedings of the 2012 International Symposium on Memory Management*, 2012, pp. 37–48.