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
{yec,xfliao,hyjin}@hust.edu.cn, {yxu47,xshen5}@ncsu.edu, Yan.Solihin@ncsu.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.

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

...
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 mentioned 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.
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](image)

**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):
    if bit 63 of addr is 1
        return NVM
    else
        if bit 47  is 1
            return Relative
        else  IJ *to = va2ra(p)
    else
        if determineY(p) GH Relative IJ *to = ra2va(p)
        else  IJ *to = p

determineY(p):
    if bit 47 is 1
        return Relative
    else  IJ *to = p

pointerAssignment(char** to, char* p):
    if determineX(to) = NVM
        if determineY(p) = Relative
            return *to = p
        else  *to = va2ra(p)
    else  
        if determineY(p) = Relative
            return *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;

<table>
<thead>
<tr>
<th>operation</th>
<th>semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>cast operator</td>
<td></td>
</tr>
<tr>
<td>(T*)p</td>
<td>$$.val = p.val</td>
</tr>
<tr>
<td>(T*)i</td>
<td>$$.val = i.val</td>
</tr>
<tr>
<td>(I)pxv</td>
<td>$$.val = pxv.val</td>
</tr>
<tr>
<td>(I)pxr</td>
<td>$$.val = ra2va(pxr.val)</td>
</tr>
<tr>
<td>unary operator</td>
<td></td>
</tr>
<tr>
<td>++p, --p, !p, ~p</td>
<td>$$.val = op p.val</td>
</tr>
<tr>
<td>&amp;p</td>
<td>$$.val = p.addr</td>
</tr>
<tr>
<td>*pxv</td>
<td>$$ = *(pxv.val)</td>
</tr>
<tr>
<td>*pxr</td>
<td>$$ = *(ra2va(pxr.val))</td>
</tr>
<tr>
<td>sizeof p</td>
<td>$$.val = op p.type</td>
</tr>
<tr>
<td>alignof p</td>
<td>$$.val = op p.type</td>
</tr>
<tr>
<td>assignment operator</td>
<td></td>
</tr>
<tr>
<td>pny = pxv</td>
<td>$$.val = va2ra(pxv.val)</td>
</tr>
<tr>
<td>pny = pxr</td>
<td>$$.val = pxr.val</td>
</tr>
<tr>
<td>pdy = pxv</td>
<td>$$.val = pxv.val</td>
</tr>
<tr>
<td>pdy = pxr</td>
<td>$$.val = ra2va(pxr.val)</td>
</tr>
<tr>
<td>p = NULL</td>
<td>$$.val = NULL.val</td>
</tr>
<tr>
<td>pxy += i, pxy -= i</td>
<td>$$.val = pxy.val op i</td>
</tr>
<tr>
<td>logical operator: &amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>p op i, i op p</td>
<td>$$.val = (I)p op i; $$.type = I</td>
</tr>
<tr>
<td>p op p'</td>
<td>$$.val = (I)p op (I)p'; $$.type = I</td>
</tr>
<tr>
<td>conditional operator</td>
<td></td>
</tr>
<tr>
<td>p ? expr : expr</td>
<td>$$.val = (I)p ? expr : expr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operation</th>
<th>semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>additive operator: +, -</td>
<td></td>
</tr>
<tr>
<td>pxy op i</td>
<td>$$.val = pxv.val op i; $$.type = pxv.type</td>
</tr>
<tr>
<td>i + pxy</td>
<td>$$.val = i + pxv.val; $$.type = pxv.type</td>
</tr>
<tr>
<td>pxv - pxv'</td>
<td>$$.val = pxv.val op pxv'.val; $$.type = I</td>
</tr>
<tr>
<td>pxr - pxv</td>
<td>$$.val = ra2va(pxv.val) op pxv.val; $$.type = I</td>
</tr>
<tr>
<td>pxv - pxr</td>
<td>$$.val = ra2va(pxv.val) op ra2va(pxr.val)</td>
</tr>
<tr>
<td>pxr - pxr'</td>
<td>$$.val = pxr.val - pxr'.val; $$.type = I</td>
</tr>
<tr>
<td>postfix operator</td>
<td></td>
</tr>
<tr>
<td>p++, p--</td>
<td>$$.val = p.val op</td>
</tr>
<tr>
<td>p[i],</td>
<td>$$.val = *(p+i)</td>
</tr>
<tr>
<td>p.identifier, p-&gt;identifier</td>
<td>$$.val = *(p + OffsetOf(identifier))</td>
</tr>
<tr>
<td>pxv(argument list)</td>
<td>$$.val = pxv.val (argument list)</td>
</tr>
<tr>
<td>pxr(argument list)</td>
<td>$$.val = ra2va(pxr.val) (argument list)</td>
</tr>
<tr>
<td>relational and equality operator: &lt;, &gt;, &lt;=, &gt;=, =, !=</td>
<td>$$.val = op p.val</td>
</tr>
<tr>
<td>pxv op pxv'</td>
<td>$$.val = pxv.val op pxv'.val</td>
</tr>
<tr>
<td>pxr op pxr'</td>
<td>$$.val = ra2va(pxr.val) op ra2va(pxr'.val)</td>
</tr>
<tr>
<td>pxr op pxv</td>
<td>$$.val = pxv.val op ra2va(pxr.val)</td>
</tr>
<tr>
<td>pxv op i</td>
<td>$$.val = pxv.val op i</td>
</tr>
<tr>
<td>i op pxv</td>
<td>$$.val = ra2va(pxv.val) op i</td>
</tr>
<tr>
<td>p op NULL</td>
<td>$$.val op NULL</td>
</tr>
</tbody>
</table>

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](image)

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 = 1.

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)](image)

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 looksaside 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.

![Fig. 6: Design of storeP function unit and dataflow of operation psv = ppx](image)

PolB: Persistent Object Look-aside Buffer, from ObjectID to virtual address; VALB: Virtual Address Look-side Buffer, from virtual address to ObjectID; POW: Persistent Object Walker; VAW: Virtual Address Walker; PW: Page Table Walker;
Since virtual to relative address conversion may incur a variable latency, we use a buffer to hold outstanding \textit{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 \textit{memory management unit} (MMU) is added two lookaside buffers to implement \textit{ru2va} and \textit{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 \textit{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, \textit{storeP} may incur a fault for errors listed in Table I.

c) Hardware complexity and storage overheads: The \textit{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>
<thead>
<tr>
<th>Structure</th>
<th>Entry Size (Bytes)</th>
<th>Num Entries</th>
<th>Total (Bytes)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>16</td>
<td>32</td>
<td>512</td>
<td>0.0205</td>
</tr>
<tr>
<td>POLB</td>
<td>12</td>
<td>32</td>
<td>384</td>
<td>0.0137</td>
</tr>
<tr>
<td>VALB</td>
<td>12</td>
<td>32</td>
<td>384</td>
<td>0.0137</td>
</tr>
</tbody>
</table>

Total size: 1,280 bytes; Total area: 0.0479 mm²

The first two \textit{storeP} instructions store pointer \texttt{p} and \texttt{n} into temporary variables \texttt{tmp_p_0} and \texttt{tmp_n_0}. Since stack variables are in volatile memory, according to the semantic for pointer assignment \texttt{pdy = pxr}, the \textit{storeP} instructions translate \texttt{p} and \texttt{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 \texttt{void* p = malloc(1024)}, the compiler can determine that: (1) \texttt{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 \texttt{malloc} and \texttt{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 \texttt{pvr} pointer to a stack variable makes the variable a \texttt{pvr} pointer.

![Fig. 8: Components and workflow of the compiler-based approach](image)

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 \texttt{p != n}, the compiler inserts \texttt{determineY} for the pointers on both sides and converts them to virtual addresses accordingly; for assignment operation, the compiler inserts \texttt{pointerAssignment} function call.

```c
void Append(Node* p, Node* n){
  if(p != n)
    p.next = n;
  else return;
}
struct Node{
  int value;
  Node* ... Relative ? ra2va(p) : p;
  Node** to = &(tmp_p_1.next);
  pointerAssignment(to, n);
}
```

![Fig. 9: Linked list appending function generated by the LLVM pass](image)

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 reccompilation (or binary rewriting) such that the store instructions in the code can be replaced with either \texttt{storeD} or \texttt{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

motion, or common subexpression elimination. For example, for \texttt{p != q && p != o}, the software user-transparent persistent reference compiler may generate a \texttt{ra2va(p)} function call for each \texttt{p} of the two statements. Trivial value numbering can easily remove the redundant \texttt{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 \texttt{p} detached \cite{39} during program execution. The code generated for the software method interrupts at the second \texttt{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 \cite{40}, they work as usual as our proposal does not change data placement in the physical address space. Virtual address based stride prefetchers \cite{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 \cite{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.

![Fig. 10: Generated code for p != q && p != 0. Pool detach causes inconsistent program semantics.](image)


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 workload (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.

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

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 × 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>dynamic checks</th>
<th>abs. to rel.</th>
<th>rel. to abs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>8,200,020</td>
<td>2944</td>
<td>14</td>
</tr>
<tr>
<td>Hash</td>
<td>2,577,030</td>
<td>668</td>
<td>668</td>
</tr>
<tr>
<td>RB</td>
<td>14,518,568</td>
<td>18.6</td>
<td>15.2</td>
</tr>
<tr>
<td>Splay</td>
<td>25,572,050</td>
<td>15.2</td>
<td>15.2</td>
</tr>
<tr>
<td>AVL</td>
<td>14,407,532</td>
<td>7.93</td>
<td>7.93</td>
</tr>
<tr>
<td>SG</td>
<td>18,137,435</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

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.

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.
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.

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

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 pointers incur 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 offset-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


Persistent key-value store from pmdk,” https://github.com/pmem/pmdk/tree/master/ucexamples/libpmemobj/map.


M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrakov, “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.


Persist_ptr from PMDK,” https://pmem.io/libpmemobj-cpp/master/doxygen/classpmem::1_lobj_1_1persistent_ptr.html.


