Hardware Supported Permission Checks On Persistent Objects for Performance and Programmability

Tiancong Wang, Sakthikumaran Sambasivam, James Tuck
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC, USA
{*twang14,ssambas,jtuck*}*@ncsu.edu*

***Abstract*—Non-Volatile Memory technologies are advancing rapidly and may augment or replace DRAM in future systems. However, a key question is how programmers will use them to construct and manipulate persistent data. One possible approach gives programmers direct access to persistent memory using relocatable persistent pools that hold persistent objects which can be accessed using persistent pointers, called ObjectIDs. Prior work has shown that hardware-supported address translation for ObjectIDs provides significant performance improvement and simplifies programming, however these works did not consider the large overheads incurred to check permissions before accessing persistent objects.**

In this paper, we identify permission checking in hardware as a critical mechanism that must be included when translating ObjectIDs to addresses in order to simplify programming and fully benefit from hardware translation. To support it, we add a System Persistent Object Table (SPOT) to support translation and permissions checks on ObjectIDs. The SPOT holds all known pools, their physical address, and their permissions information in memory. When a program attempts to access a persistent object, the SPOT is consulted and permissions are verified without trapping to the operating system. We have implemented our new design in a cycle accurate simulator and compared it with software only approaches and prior work. We find that our design offers a compelling 2.9x speedup on average for microbenchmarks that access pools with the RANDOM pattern and 1.4x and 1.8x speedup on TPC-C and vacation, respectively, for the SEPARATE pattern.

***Keywords*-non-volatile memory; persistent memory programming; persistent data permission check**

I. Introduction

Non-volatile memory technologies are advancing rapidly and may augment or supplant DRAM as main memory given their higher capacities, lower stand-by power, and reasonably fast access latencies [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Products based on Intel's and Micron's 3D Xpoint memory technology are already in the market [1]. Non-volatile Main Memory (NVMM) means that programmers can keep important data in *persistent* data structures in main memory rather than serializing them to disk.

One area of significant interest is the mechanism that programmers will use to manipulate persistent objects. We share a view consistent with many prior works [11], [12], [13], [14], [15], [16] that programmers will access persistent objects the same way they access volatile memory, using pointers, references, or objects supported by common low-level programming abstractions. Pointers stored within a persistent object to other persistent objects are especially challenging because they need to be permanent. Virtual addresses, while they may seem ideal, cannot be reserved across program executions (e.g. ASLR) or guaranteed to be the same for multiple processes that may want to share the object and, hence, they are unsuitable as permanent pointers.

Recent works [17] have proposed using object identifiers (ObjectIDs) as permanent pointers to persistent objects. ObjectIDs consist of a pool identifier and a byte offset within the pool to identify and locate objects. They are general enough to access objects within the same pool or across pools, and they can serve as the basis for building linked structures within and across pools. Furthermore, ObjectIDs require translation to an address before they can be used to access a persistent object.

While ObjectIDs are sufficient to enable persistent memory programming on current systems, they pose significant challenges too. Manual translation of ObjectIDs is an onerous burden to place on programmers [18]. Even if this translation is somehow cached in a fast data structure, like in PMDK[17], the overhead for frequently accessing this structure could be significant because it would require many instructions per translation. Hardware or software support can reduce some of these overheads [13], [15]. Chen *et al.* [13] evaluated various low-level software mechanisms to reduce the overhead of manipulating ObjectIDs. Wang *et al.* [15] proposed hardware support for translating ObjectIDs to virtual addresses using a mechanism similar to existing hardware for virtual memory. ObjectIDs are treated as a persistent address space, and loads and stores access persistent objects directly using the ObjectID as the address. This approach provided a large performance advantage on the workloads studied.

Even though hardware-supported translation of ObjectIDs makes it easier to program and use persistent objects, they are not as simple to use as normal pointers. Consider a linked data structure traversal that spans multiple pools: after opening the first pool, we find an ObjectID pointing to another pool that needs to be accessed. Barring the presence of additional information, it is unknown if the next pool has already been opened or not. The programmer must reason about whether the persistent pool has already been opened and mapped into the process' address space or build in appropriate checks. Similarly, permissions (i.e. like file permissions in a conventional file system) also must be checked. Without these additional checks to ensure the pool is opened and with appropriate permissions, the programmer may incur memory protection errors.

Interestingly, this is quite different from programming with conventional pointers or references. Typically, if a pointer is known and non-NULL, it can be assumed legal in a correctly implemented program. However, ObjectIDs do not come with this guarantee because they persist across runs of the program, creating an additional step in reasoning about their correctness. In current designs, the burden to perform these correctness checks rests fully on the programmer to implement them in software.

Instead, we propose hardware support for validating ObjectIDs and checking their permissions on demand. We borrow from well known architecture techniques used to support virtual memory, and we extend them to support common file system operations to make using persistent objects simpler and higher performing. In particular, we extend and modify the Persistent Object Look-aside Buffer (POLB) and Persistent Object Table (POT) proposed by Wang *et al.* [15]. We replace the per process POT with a System Persistent Object Table (SPOT) that holds all known pools in the system, their physical address, and their permissions information. When a program suffers a POLB miss, a privileged hardware SPOT walk traverses the SPOT to find the relevant entry, check permissions, and copy it into the POLB. If the persistent ObjectID is known and the permissions check out, the requesting process obtains the mapping and continues execution without trapping to the operating system.

With this design, we can improve a program's performance significantly. Part of the performance improvement comes from our new hardware. Permission checking is performed by hardware along with fast hardware-supported address translation. Also, the hardware SPOT walk avoids system calls for mapping pools. Another part of the performance comes from simplifying the software. Programmers can omit frequent and costly library calls that check the permission of pools they wish to reference. This streamlines the code and reduces overhead, pushing some of the complexity into low latency hardware and system code.

Now, we briefly summarize our main contributions. 1) We point out that ObjectIDs held in persistent objects can point to unmapped persistent regions, thereby requiring the programmer to reason about the permissions and mapped status of those objects. We remove this burden from the programmer by performing permissions checks in hardware and by bypassing the requirement that objects be manually opened and mapped before being accessed. 2) We explore and compare multiple design choices to provide permission checks with both native software translations and our newly proposed hardware support. 3) We have implemented our new design in a cycle accurate simulator and compared it with software only approaches and prior work [15]. We use workloads consistent with prior work [11], [13], [15], [19], and we add two applications, TPC-C and vacation [20]. We modified the applications to include permissions checking before each ObjectID translation. We find that our design offers 2.9x speedup on average for the microbenchmarks we studied and 1.4x and 1.8x speedup on TPC-C and vacation respectively.

The rest of the paper is organized as follows. Section II presents important background on programming for persistent memory. Section III discusses how mapping and permissions checks are handled in current systems and the possibility for hardware support. Section IV presents our design, and the implementation is described in Section V. The methodology and results are presented in Sections VI and VII. We discuss some important related work in Section VIII, and Section IX concludes.

II. Background

A. Pools and ObjectIDs

Persistent regions of memory are organized as pools [11], [17] and given a unique identifier. Pools can be organized as a collection. Typically, there is a root object from which all other objects in the pool can be reached.

To simplify finding an object, a 64-bit pointer, called an ObjectID (Figure 1), is used to refer to persistent objects. The top 32-bits refers to a pool identifier, and the bottom 32-bits indicate the byte offset within the pool. The ObjectID precisely defines the location of an object within a persistent pool.

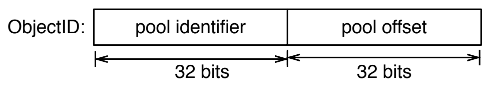


Figure 1. Structure of ObjectID [13], [15].

B. Persistent Pool and Object API

PMDK [17] and other prior works [15], [11] have described interfaces for manipulating pools and objects. We adopt the interface proposed by Wang *et al.* [15]. It supports functions for creating pools (analogous to files), objects within pools, support for persisting objects, and failure-safety through durable transactions. Table I shows a sub-set of their interface. We do not repeat a full explanation of each function since it has been described in prior work.

We point out that `pool_open` will look for a pool of the given name. If it exists and if the calling process has permission to access the pool, the pool is mapped into the

Function	Description
Pool Management	
`pool* pool_create(name, size, mode)`	Create a pool with the specified `size` and associate it with a `name`. Running process is the owner.
`pool* pool_open(name, mode)`	Reopen a pool using `name` that was previously created. Permissions will be checked.
`pool_close(pool* p)`	Close a pool `p`.
`OID pool_root(pool* p, size)`	Return the root object of the pool `p` with specific `size`. The root object is intended for programmers to design as a directory of the contents in the pool.
Object Management	
`OID pmalloc (pool* p, size)`	Allocate a chunk of persistent data with the given `size` on pool `p` and return the ObjectID of the first byte.
`pfree(oid)`	Free persistent data pointed to by the ObjectID.
Translation	
`void* oid_direct(oid)`	Translate an ObjectID to a virtual address. Used when there's no hardware translation.

Table I
POOL AND OBJECTID API AS DESCRIBED IN PRIOR WORK[15], [17].

process' address space. Later in this paper, we may refer to opening a pool or mapping a pool synonymously.

C. Hardware-Supported Translation

To reduce the overhead of frequent translations when traversing linked data structures in persistent memory, Wang *et al.* [15] proposed hardware-supported translation of ObjectID to virtual address. They added a Persistent Object Look-aside Buffer (POLB) to the processor and placed a Persistent Object Table (POT) in memory. The POLB is analagous to the TLB and the POT is analagous to the page table. Also, two new instructions, nvld and nvst, are added to the processor that use ObjectIDs to directly access persistent memory.

When pools are opened, their information is added to the POT. Later, when an nvld or nvst is used to access a persistent object, the POLB is consulted to translate the ObjectID into an address. If the translation is not present, hardware checks the POT for the translation and moves it to the POLB to satisfy the request. In the event that the translation is not present in either the POLB or POT, and exception is raised for OS level handling. Wang *et al.* ([15]) do not explain in detail how this is handled by the OS.

Wang *et al.* ([15]) also assume that the POT is located in user space, and as such cannot hold information about the physical address of persistent objects. It may only hold the translation to virtual address for each persistent pool.

1) Pipelined *versus* Parallel*:* However, they do evaluate two different designs for the POLB. In the *Pipelined* design, the POLB holds virtual address translations and is pipelined with the TLB. Hence, the ObjectID is first translated to a virtual address and then translated to a physical address. While this adds an extra cycle to the execution of an nvld, it only needs a small POLB due to the large granularity of the persistent pools.

In the *Parallel* design, the POLB holds physical address translations and is accessed in parallel with the TLB to perform translations before accessing the cache. Even though this avoids an extra cycle to compute the virtual address, this design puts more pressure on the POLB since it must hold translations at a page granularity.

III. MOTIVATION

A. Example

We consider an example to motivate the additional problem of permissions checking on persistent objects. Figure 2 shows a persistent list. Each pool contains one object. After an initial partial traversal of the list, some of the pools have been opened and some have not. The red dotted-line indicates an ObjectID pointing to an object in an unopened pool. Before a dereference to the object in the unopened pool occurs, the programmer must open the pool to map it into memory and check that the access is permissible.

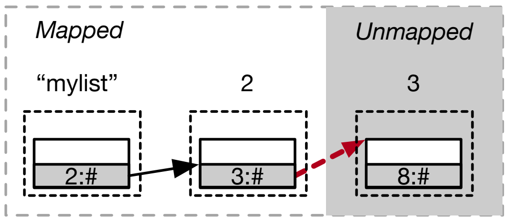


Figure 2. Persistent data structure spread across pools. Some pools are mapped and others are not.

This leads to new challenges that are different from typical linked data structures. Normally, if an address is not NULL and a program is free of memory related bugs, it is legal to dereference that address. In this case, persistent objects were created in a previous run of the program and persist until the next run. The ObjectID remains valid across program runs, however, the ObjectID would no longer be legal to access in the second run unless it were first opened. This leads to the challenge that the programmer must somehow know which parts of a linked data structure are mapped and which are not. If it is not easy to know, it is better to be conservative and check to make sure the pool is opened before accessing it.

Moreover, between two runs of the program, other programs may access this data structure and update it. New objects may be added. Hence, the programmer may not assume that the same ObjectIDs are present and remain valid from one run to the next. Hence, the legitimacy of the ObjectID must be verified again.

Function Calls	Descriptions
Pool Management	
`pool* oid_open(oid)`	Reopen a pool using an ObjectID rather than a name. Permissions will be checked.
Permissions Management	
`int oid_check(oid,mode)`	Check if corresponding pool is open. Return an error code if not.
`void* oid_check_direct(oid, mode)`	Translate but perform premissions checks first.

Table II
POOL AND OBJECTID EXTENDED API TO SUPPORT PERMISSIONS CHECKING.

```
typedef struct {
  int value;
  OID next;
} node;
// find the first node that matches data
OID find (OID head, int data){
   OID tmp = head;
   while (tmp != OID_NULL) {
     if (!oid_check(tmp,"rw"))
        oid_open(tmp,"rw");
     node *x = (node*)oid_direct(tmp);
     if (x->value == data)
        return tmp;
     tmp = x->next;
   }
   return OID_NULL;
}
```

Figure 3. Example of a persistent linked list traversal with permission check.

B. Permissions Checking Support

Inspired by the linked list example, we add a few new functions to the API to support permissions checking, as shown in Table II.

First, consider `oid_check`, which checks the mapping and permissions of a pool using a known ObjectID. This function can be implemented fully in library code by remembering which pools have already been opened. If the pool has been opened once before, we can assume that the system previously granted access. This is the approach used by Persistent Memory Development Kit (PMDK, formerly known as NVML) [17]. This approach can be efficient because it can leverage a fast data structure to perform the check, like a hash table or binary tree, and avoid any overhead of trapping to the operating system. It operates under the implicit assumption that once access to a persistent region is granted it is never revoked.

This function is helpful if an ObjectID is obtained in a linked persistent data structure, but whether or not it refers to a pool that has already been opened is unknown.

Figure 3 shows a linked list traversal modified to properly check permissions. Before attempting to convert the ObjectID to virtual address, the status of the ObjectID is determined by calling `oid_check`. If the pool it is contained within is not already open, `oid_open` is called to force the mapping of the pool and the permission check. The repeated calls to `oid_check` incur a significant and necessary overhead.

C. Optimizing Checks in Software

We can further optimize the linked list traversal specifically, and permissions checking in general, by embedding the check into the translation function. We provide a function, similar to one in PMDK [17], called `oid_check_direct` to perform software translation and permissions checking at the same time. By merging these two operations, we can leverage the fact that translations have significant locality. If the last translation was for the same pool, there is no need for any additional checks.

The pseudo-code in Figure 4 describes the procedures in `oid_check_direct` and differentiates the additions to the baseline `oid_direct` function. The new function will try to translate the ObjectID using the most recent translation, but if that fails, it will trap to the OS to check for permission to access the pool and will map the pool if permission is granted.

```
void* oid_check_direct(pool_id, offset):
    if (most_recent_info_valid && oid.pool_id ==
    most_recent_pool_id)
        return most_recent_base_addr + oid.offset;
    most_recent_info_valid = 1;
    most_recent_pool_id = oid.pool_id;
    most_recent_base_addr = OIDTranslationMap->
    find(oid.pool_id);
    if most_recent_base_addr != NULL
        return most_recent_base_addr + oid.offset;
    // Different from oid_direct
    else
        //Trap to kernel-level
        valid = check_user_permission(user ID,
    pool_id);
        if valid
            base_addr = map_pool(pool_id);
            OIDTranslationMap->insert(pool_id,
    base_addr);
        else
            abort program
        //Return to user-level
        most_recent_base_addr = OIDTranslationMap
    ->find(oid.pool_id);
        return most_recent_base_addr + oid.offset;
```

Figure 4. Pseudo-code of `oid_check_direct`. The if-else part is added to handle the situation where a translation is missing (i.e. pool not mapped).

During the translation, the pool ID will be searched in the hash table for translation as normal. But when the pool ID is not found in the table, instead of aborting the program, we provide a series of kernel-level actions that are equivalent to `pool_open`. We will perform a system check to see if the programmer has permission[1] to access the pool. If the permission is granted, the pool will be mapped into the program's address space and a translation entry will be added to the table. Thus the translation can proceed without problem and further translations will also proceed without operating system involvement. If the permission is denied, the program will be aborted.

The action taken when the pool is not already open is analogous to opening a file and the associated permissions checks therein. It is also similar to the method for handling a page fault, at least in principle. The physical page of the persistent object was once legal to access, but the mapping must be renewed on subsequent runs of the program. Interestingly, the data may still reside in the same location in NVMM. Unlike a page fault that moves the data into memory from disk, in our case, we simply need to restore the mapping. However, the OS must take action to create the mapping.

We can optimize the code in Figure 3 to use this new function. We remove lines 9 and 10 and replace the call to `oid_direct` with `oid_check_direct`.

However, it is worth noting that `oid_check_direct` cannot be replaced with a single `nvld` or `nvst` instruction because they do not include permissions checks.

D. Permissions Checks with Hardware Supported Translation

Software translation imposes significant overheads [15], [13] and permission checks in software add even more overheads, so it is advantageous to integrate these permissions checks with hardware that performs translation.

Without adding any additional hardware support, we can implement an approach that relies on hardware translation with software level checking of permissions. For example, this amounts to the same code as in Figure 3 with `oid_direct` replaced with nvld. However, the added software permission checks bring a significant overhead (shown in Section VII). Instead, these checks would ideally occur as part of the nvld not as a separate functionality.

With modest changes, the *Pipelined* design (Section II-C1) can support streamlined permissions checks. Suppose an nvld or nvst accesses an ObjectID for a pool that has not yet been opened. In this case, the POT will miss. We modify the design to raise an exception on a POT miss, and the OS runs an exception handler, like a page fault handler, that will find the corresponding pool for the ObjectID. If such a pool exists and if the running process has permission, it can be immediately mapped into the POT. This mechanism allows the programmer to avoid checks on ObjectIDs that are known to be legitimate. This extension adds no additional cost to the *Pipelined* design proposed in [15]. However, it does incur a significant performance penalty on a POT miss. An OS exception must be raised and a handler invoked. This may add significant overhead when traversing data structures that spread across many different pools.

[1]We will grant maximum permission level to the programmer. For example, if the programmer has read and write permission to the pool, we will grant both permissions.

We argue for an even more streamlined design. Persistent objects are expected to be permanently resident in memory. Unlike pages that have been swapped to disk or files that need to be copied into memory from disk, few actions are needed to make a persistent object accessible, since it still resides in memory. In reality, all that is needed is knowledge of its location in memory. A system-level table that works like a page table could provide this information at relatively low hardware complexity. With such a design, we do not need system calls (in software) or a trap to an OS exception handler (with hardware support) to map pools into the program.

Such integration is non-trivial. The hardware would need access to the privileged system-level data structure that would identify the physical location and permissions information of persistent pools. It's worth noting that this is substantially different from the information provided by the page table. In the next section, we build on this idea, and we describe our hardware support for performing permissions checks directly in hardware.

IV. Design

A. System Persistent Object Table

Once created, persistent objects are always in memory. Rather than relying on the operating system to re-map them into a process' address space, we provide hardware support to carry-out this action and to check permissions. We add a system data structure called the *System Persistent Object Table* (SPOT) to keep a record of all the pools created in the system. When a pool is first created, an entry is added to the SPOT that stores the physical address of the pool and important information, like the owner, group, and permission information.

The SPOT serves as a backing store for the POLB and replaces the POT in the prior work [15], as shown in Figure 5. In the event of a miss in the POLB, a hardware walk over the SPOT, similar to a page table walk, finds the corresponding pool's entry in the table and adds it to the POLB if allowed. As long as the desired object is found in the SPOT and permissions check out, this entire process occurs without trapping to the operating system. However, if the SPOT indicates that the object is not valid or there are insufficient privileges to access the pool, an exception is raised for OS handling. If the OS cannot resolve the request, a memory protection violation is signaled.

Note, the SPOT walk is different from a typical page table walk that occurs on a TLB miss. A page table only contains entries for pages that have been mapped into the virtual address space. It does not contain pages for files or other objects that are not yet known to the program. Furthermore, for entries in the page table to be updated, an OS-level routine must take an action. On the other hand, the SPOT

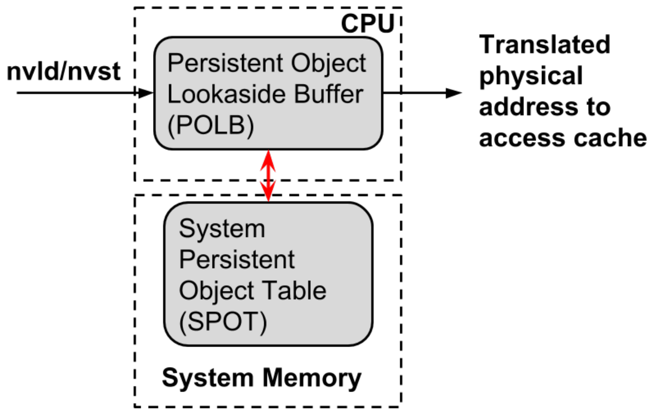

CPU
nvld/nvst
Persistent Object Lookaside Buffer (POLB)
Translated physical address to access cache
System Persistent Object Table (SPOT)
System Memory

Figure 5. Overview of the design of automatic permission checks with System Persistent Object Table (SPOT).

does contain persistent objects that may not be legal for a program to access, and hardware helps decide whether or not it is legal for the program to access them.

B. Pool Open Using SPOT

When an entry from the SPOT is moved to the POLB by hardware, this is equivalent to the `pool_open` function in the API. It legalizes the ObjectID by creating a translation. In prior work, it was assumed that a persistent pool was first mapped into the address space of the process before accessing it [17], [15]. However, we are skipping that step by establishing a direct translation from ObjectID to physical address.

We design our POLB to translate directly from ObjectID to physical address. This implies building on top of the *Parallel* design (Section II-C1).

C. Permission Checking Using SPOT

We also add hardware that is equivalent to the `pool_check` function. Each entry in the SPOT stores the physical address of a pool and important information, like the owner, group, and permission list. We borrow our design from standard Unix file system implementations that specify owner, group, and access settings (e.g. read, write, and execute privilege) for each file.

In order to support this check in hardware, we add protected registers to the core that specify the current user and the groups to which the user belongs. These registers can only be set during privileged execution.

Before an entry is copied from the SPOT, the user and group are validated against the SPOT entry. For example, if the current user matches the owner of the persistent pool, then access is permitted. Or, if the user is different from the owner but the user is a member of the same group, some level of shared access may be permitted.

We discuss the permission checking logic more in Section V-B.

D. Organization of the SPOT

The SPOT will hold a record of the persistent pools in the system. The ObjectID format allows up to 2^{32} unique pool identifiers, so we design the SPOT for scalability up to a large number of persistent pools. X86 page tables use a multi-level design. Such designs provide low memory overhead in the common case, allow for a full address space, and work well even with small page sizes. For these reasons, we also propose a multi-level design for the SPOT.

The organization of the table is dependent on the layout and management of persistent pools. If pools are always laid out contiguously in the physical address space, then only the top 32-bits of the ObjectID matter for translation. We can design fewer levels of table by translating at large granularities, and this results in less memory overhead and fewer accesses to memory to look up a SPOT entry.

On the other hand, it is more likely that persistent pools are not contiguous, in order to allow the system to flexibly manage the NVMM device with pools of varying sizes. This implies translation at the granularity of the page size. In addition to incurring more overhead from additional tables, there is also potentially more redundant information. For example, the permissions information is only needed for each pool, not for each page. To optimize for this case, we design a multi-level SPOT that translates at page granularity but only stores permission information at pool granularity. These designs are presented in Section V.

V. System Implementation

We now describe the hardware implementation for the SPOT and translation and permission checks.

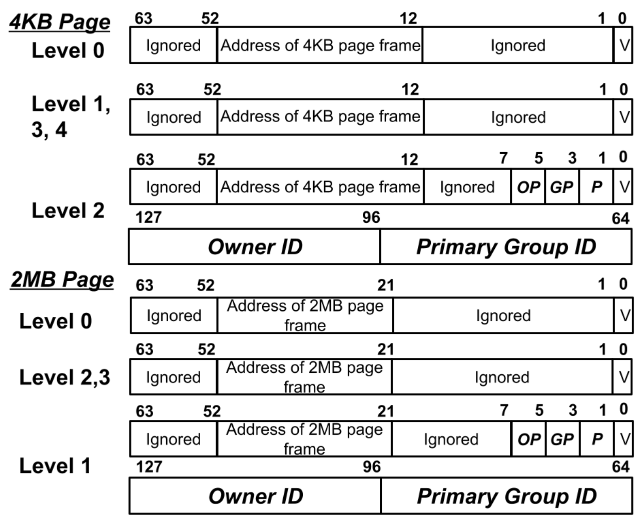
A. System Persistent Object Table

When a translation is not found in the POLB, a privileged hardware SPOT walk traverses the SPOT looking for the corresponding pool and object information. The starting address of the SPOT is stored in a control register so that hardware can perform an SPOT walk, similar to a page table walk.

We adopt a multi-level design for the SPOT, and we consider multiple designs to account for different page sizes. Page size is a key architectural parameter for the design of the SPOT. Larger pages mean a smaller table and fewer look-up steps. However, larger pages have the downside of greater internal fragmentation.

Figure 6 shows the design of the SPOT entry depending on the page size and for each level of the table. We assume the same page sizes as the Intel 64 architecture, where page size can vary among 4KB, 2MB or 1GB [21]. Each entry either stores the address of the next level of the table, or entries in the last level table store the physical page frame number for the page.

Another functionality of the SPOT is to perform the permission check. One challenge is that we only need the



V: Valid bit. 1: Present. 0: Not Present
OP (2 bits): Owner Permission bits. 1x: Write. 10:Read-only. 0: No permission.
GP (2 bits): Group Permission bits. Permission of owner's primary group. Same bit representation.
P (2 bits): Other Permission bits, for all the other users. Same bit representation.
Owner ID (32 bits): Set by O/S. Represent the owner of this page.
Primary Group ID (32 bits): Set by O/S. Represent the primary group of this page.

Figure 6. SPOT entry details on systems with 4KB or 2MB page size.

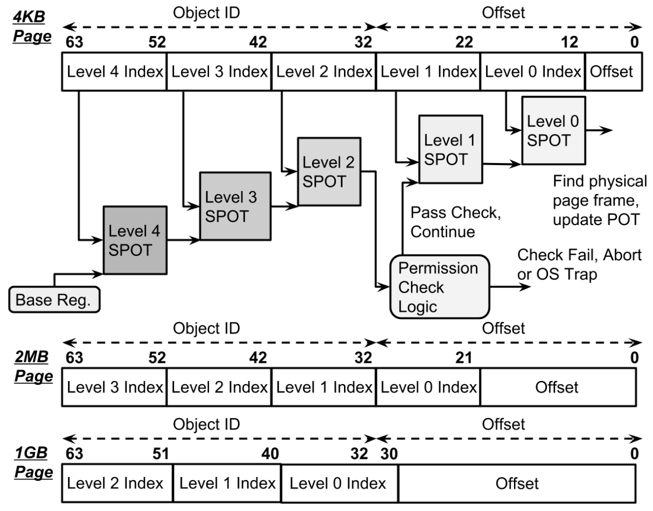


Figure 7. Hardware SPOT walk on 4KB page size system. 2MB and 1GB have similar procedure with different number of SPOT levels.

owner, group, and permissions bits per pool, but we need translations per page. If we repeat the owner, group, and permissions bits in every entry of the last level of the SPOT, it would add a significant amount of redundant information for all pages in a pool. Instead, we place the permissions information at the granularity of a pool, and we only store translations at the granularity per page. To save the owner ID and group ID, the mid-level tables need an extra 64 bits compared to the other table entries. The permission check is depicted in Figure 7 with the Permission Check Logic reading the entry from Level 2 of the SPOT.

The overall procedure for the SPOT walk is demonstrated in Figure 7 with a 4KB page example. The starting address is stored in a special control register and preset by privileged operations. The 64-bit ObjectID is used to walk the SPOT. Each level takes part of the ObjectID to calculate the index for its table. A look-up at that index yields the starting physical address for the next level of the table. We use the level 2 table to perform the permission check logic (described in Section V-B) to verify if the user has permission or not. Lower levels will continue to be accessed only after the permissions are validated. After the lowest level entry is read out, the physical page frame is obtained and used to provide an entry to the POLB along with the correct read and write permissions from level 2 (more details in Section V-B).

B. Permission Checking Logic

To support permission checking logic, we add permissions information to the mid-level SPOT tables, and we add protected control registers to the core to indicate the user ID and primary group ID. In reality, a user may be part of more than one group. The number of registers devoted to group IDs could be adjusted to match the common case, provided it is not too large[2]. Each user is assigned to a primary group and we store the primary group ID to check permissions. These registers would be set as part of the context switch code in the OS. We assume 32-bits is sufficient for the user ID and primary group ID. So, a 64-bit register can hold both of the IDs.

Figure 8 illustrates the logic added to perform permission checking. First, bits [127:96] will be read out to compare the current user ID against the owner ID. If the IDs match, meaning the current user is the owner, the POLB will be updated with the physical page frame number in bits [51:12] from the last level entry of the SPOT, provided the read/write permission checks out. The operation of nvst will fail if the Owner Permission bits are `01` because that means the entry is read-only. Both nvld and nvst will succeed if the Owner Permission bits are `11` or `10`.

If the IDs do not match, primary group IDs will be checked to see if the current user is in the primary group of the owner. If so and the group is granted access (Group Permission is not `00`), the POLB will be updated with physical page frame number if the read/write permission bits match the nvld/nvst operation. If the group matches but has no access, an exception is raised for handling by the OS or user program. If the current user is not in the group, the Other Permission

[2]An informal inspection of Linux workstations suggests that most users are only a member of a handful of groups at most.

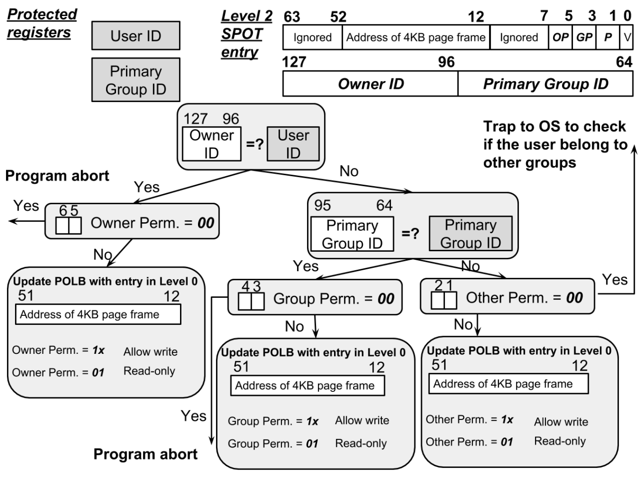

Protected registers
User ID
Primary Group ID
Level 2 SPOT entry
63 52 12 7 5 3 1 0
Ignored
Address of 4KB page frame
Ignored
OP
GP
P
V
127 96 64
Owner ID
Primary Group ID
127 96
Owner ID
=?
User ID
Trap to OS to check if the user belong to other groups
Program abort
Yes
No
Yes
6 5
Owner Perm. = 00
95 64
Primary Group ID
=?
Primary Group ID
No
Update POLB with entry in Level 0
51 12
Address of 4KB page frame
Owner Perm. = 1x Allow write
Owner Perm. = 01 Read-only
Yes
No
4 3
Group Perm. = 00
2 1
Other Perm. = 00
Yes
No
No
Update POLB with entry in Level 0
51 12
Address of 4KB page frame
Group Perm. = 1x Allow write
Group Perm. = 01 Read-only
Update POLB with entry in Level 0
51 12
Address of 4KB page frame
Other Perm. = 1x Allow write
Other Perm. = 01 Read-only
Yes
Program abort

Figure 8. Flowchart of the added hardware logic for permission check used during SPOT walk.

bits will be used to check permission. If the Other Permission bits are `00`, an OS exception will be raised and checks if the user belongs to other groups of the owner. Otherwise, the permission can be granted and the POLB is updated with the physical page frame number if read/write permission checks out.

C. Miscellaneous

1) Load-Store Disambiguation in Parallel*:* As reported in [15], the *Parallel* design adds complexity to load-store disambiguation in out-of-order processors. We assume oracle support for load-store speculation, so that we can evaluate the *Parallel* design on out-of-order processors. Many prior works have considered how to support aggressive load and store speculation [22], [23], and these designs need to be re-considered in the context of ObjectIDs. We leave such considerations to future work.

2) Getting Unique Pool Identifiers: Pools need unique identifiers. One way to obtain them is using a utility like the the Universally Unique Identifier [24] in Linux. Although the probability of a duplicate in UUID is not zero, it's very close to zero in practice. In our implementation, the Pool ID is a 32-bit number, so we need some function to map the 128-bit number into a 32-bit pool ID which increases the likelihood of collision. However, if a collision happens, we can linearly search for the next available pool ID by linear probing on the SPOT.

3) Concurrency and Updates: Since the SPOT is shared, concurrent updates will occur. These updates are made by the operating system and must be synchronized.

VI. Methodology

A. Simulation Environment

We simulate our hardware designs on a cycle-accurate X86 simulator, Sniper [25]. Sniper is based on the interval core model and the Graphite [26] simulation infrastructure. In our simulation, we use its instruction-window centric (IW-centric) core model that supports both in-order and out-of-order processor simulation, provided in the latest 6.1 version. We use the simulator's default architectural model, Intel Xeon X5550 Gainestown (Nehalem-EP) processor, and other parameters used in the simulator are described in Table III.

Component	**Configuration**
Out-of-order core	4 cores, 2.66GHz Issue width: 4 LQ: 48, SQ: 32, ROB: 128 Branch predictor: Pentium M, Branch mispredication penalty: 8 cycles page size: 4KB, cache block size: 64 Bytes
Cache	D-TLB: 64, I-TLB: 128 L1D: 8-way 32KB, 3 cycles (1ns) L1I: 4-way 32KB, 3 cycles (1ns) L2: 8-way 256KB, 8 cycles (3ns) L3: 16-way 8MB, 27 cycles (10ns)
clwb latency	100 cycles (38ns)
DRAM	1GB, 120 cycles (45 ns)
NVMM	Battery-backed DRAM, 120 cycles
Context Switch	266 cycles (100 ns)

Table III
Simulator Configuration.

We extend Sniper to simulate hardware translation of ObjectIDs proposed by Wang *et al.* [15]. We also extend Sniper to simulate system behaviors like the page table walk [3]. We have extended it to fully emulate the page table walk. We read the base register of the highest level of the page table and access each entry in all levels until we find the page table entry. The latency simulated depends on which level of the cache hierarchy each entry is found within. We use a similar mechanism to simulate the SPOT walk.

B. Designs

We implement permission checks in software as described in Section III-B and Section III-C, referred to as SwT-Base and SwT-Opt respectively. SwT-Base is a naive implementation, and SwT-Opt is similar to PMDK [17]. We use the C++ Standard Template Library (STL) map (tree-based implementation) to implement the software translation table and the system-level data structure that holds all of the persistent pools (i.e. equivalent to the SPOT but in software).

We also implement two designs that use hardware-supported translation. One has hardware support for translation that doesn't integrate the permission check (i.e. no SPOT). In this design, the programmer must call `oid_check` manually, as described in Section III-D. This design will be referred to as HwT.

Finally, we implement our proposed design with the SPOT. It is called HwT+SPOT, and we compare its performance

[3]Sniper doesn't have a model for the page table walk, by default. Instead, it charges a fixed 30 ns (80 cycles) for a TLB miss.

Design	Description
SwT-Base	Naive implementation. It requires programmers to call `oid_check` to determine the status of a pool and manually open a pool. Described in Section III-B.
SwT-Opt	Programmers can use `oid_check_direct` to translate an ObjectID and to perform software translation while checking permissions. Described in Section III-C.
HwT	Baseline hardware translation support, where programmers additionally call the `oid_check` function to check permissions. Described in Section III-D.
HwT+SPOT	Hardware translation support with automatic permission checking using the SPOT.

Table IV
SUMMARY OF THE DESIGNS USED IN THE EVALUATION.

Workload	Description
Linked-list	Generate 700 random integers and search them in a linked-list. If a number is found, remove the node from the list. Otherwise, insert a new node to the list with the number as key.
Binary Search Tree	Generate 5000 random integers and search them in a binary tree. If a number is found, remove the node from the tree and replace it with a node with a maximum key on its left sub-tree. Otherwise, insert a new node to the tree with the number as key.
Red-black Tree	Generate 3000 random integers and search them in the RB-Tree. If a number is found, remove the node and re-balance the tree according to the red-black tree rule. Otherwise, insert a new node with the number as key and also re-balance.
B-Tree (order=7)	Generate 5000 random integers and search them in the B-Tree. If a number is missing, insert a new node with the number as key to the tree and re-balance.
B+ Tree (order=7)	Generate 5000 random integers and search them in the B+ Tree. If a number is found, remove the node. Otherwise, insert a new node with the number as key. Both insertion and deletion need to re-balance the tree.
String Position Swap	Randomly swap a pair of strings in a 32KB string array and repeat 10000 times.
TPC-C	Generate 1 warehouse according the parameters in TPC-C spec [27] and perform 1000 transactions.
Vacation	Travel reservation system with 4096 tasks and 4 queries per each task. Used the suggested high contention configuration (-n4 -q60 -u90 -r16384 -t4096) [28].

Table V
SUMMARY OF WORKLOADS.

with the other three designs. All designs are summarized in Table IV.

C. Workloads

We evaluate our designs with six micro-benchmarks similar to prior works [15], [11], [29] and applications TPC-C [27] and Vacation from the STAMP benchmark suite [28], which is also evaluated in [20]. The detailed description of each workload is listed in Table V. All the benchmarks allocate their core data structure (B+ Tree in TPC-C and linked list and red black tree in Vacation) in persistent storage with the APIs in Table I and II. For microbenchmarks, we use the transaction support in the APIs, and for the applications we retain their existing failure-safety support.

Patterns	Description
ALL	All persistent data are in one pool.
EACH	Each node in the structure is put in separate pools.
RANDOM	The data structure is allocated in 32 pools by randomly selecting one pool when creating a new node.
COMBINED	All persistent data structures (B+ Tree in TPC-C, Linkedlist and RB-Tree in Vacation) are allocated in one pool.
SEPARATE	Each data structure is allocated in separate pools. There are 7 B+ Trees used in TPC-C. And there are 16426 linked-lists and 4 red black trees allocated in the system.

Table VI
POOL ACCESS PATTERNS USED IN THE EVALUATION FOR MICRO-BENCHMARKS (ALL, EACH AND RANDOM) AND APPLICATIONS (COMBINED, SEPARATE).

We develop several pool usage patterns for each microbenchmark, namely ALL, EACH and RANDOM, and COMBINED or SEPARATE for each application. All are summarized in Table VI. This approach allows us to evaluate a variety of access patterns on our microbenchmarks and applications and understand their implications on our proposed hardware.

In our implementations, we assume programmers do not have knowledge of all the relevant pools to open at the beginning of a program. Instead, permission checking is embedded in the traversal code of each data structure, through either software interfaces (SwT-Base and HwT) or translation procedures (SwT-Opt and HwT+SPOT).

VII. EVALUATION

A. Overall Performance

First, we look at the overall performance improvement provided by our proposed architectures. Figure 9 shows the results on the out-of-order processor normalized to SwT-Base. We compare the performance of all four designs. We show the results for ALL, EACH and RANDOM patterns on microbenchmarks and COMBINED and SEPARATE patterns on applications.

For the case that only one pool is used in the system (ALL/-COMBINED), Figure 9(a) shows a speedup for HwT+SPOT over SwT-Base of 1.1x on average for the microbenchmarks, 1.3x on TPC-C, and 1.3x on vacation. Overall, no design is significantly better or worse because only a single pool needs to be translated and have its permission checked.

EACH, shown in Figure 9(b), puts every node in the data structure in separate pools, which results in hundreds to

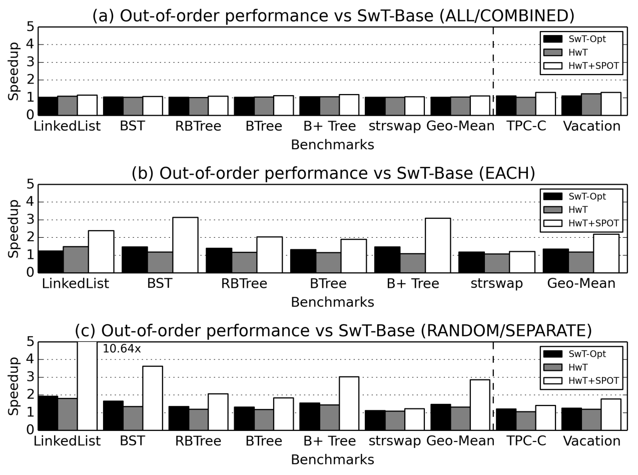


Figure 9. Speedup of SwT-Opt, HwT and HwT+SPOT over SwT-Base for each usage pattern on an out-of-order processor. (a) Shows the results of microbenchmarks with ALL and applications with COMBINED. (b) Shows only microbenchmarks with EACH. (c) Shows microbenchmarks with RANDOM and applications with SEPARATE.

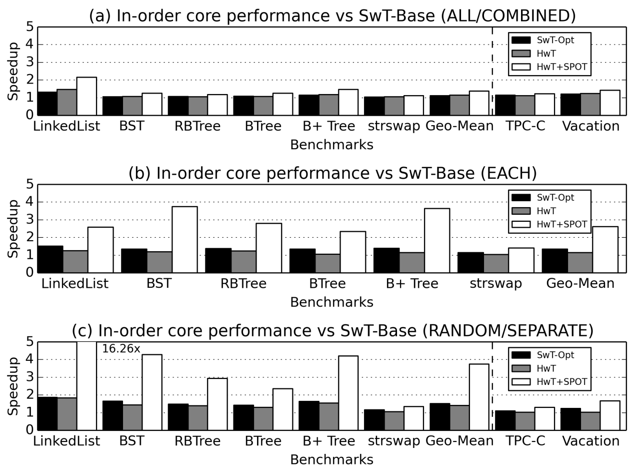


Figure 10. Performance of different designs on the in-order processor.

thousands of pools used in each program. This case shows a clear advantage for HwT+SPOT, since it is able to accelerate mapping and permissions checking operations. HwT+SPOT shows a speedup of 2.2x on average for the microbenchmarks and 1.4x and 1.8x on TPC-C and vacation, respectively. Interestingly, SwT-Opt is slightly better than HwT (1.3x vs 1.1x speedup on average of microbenchmarks). SwT-Opt has significant overhead when performing translations, and that is not present for HwT. However, HwT has overhead performing permissions checks that negate much of the benefits of hardware translation, since a permissions check is triggered for each node visited. These checks are costly, involving a trap to the OS.

The RANDOM/SEPARATE pattern is shown in Figure 9(c). For HwT+SPOT, RANDOM/SEPARATE has the biggest performance speedup across all the workloads and designs. Since fewer pools need to be mapped[4], there are more POLB hits, and less time is spent walking the SPOT. HwT+SPOT has 2.9x speedup on average for the microbenchmarks and 1.4x and 1.8x speedup on TPC-C and vacation, respectively.

B. In-order vs Out-of-order

We also show the performance of all designs on an in-order processor in Figure 10. The speedup is normalized to SwT-Base on the in-order processor. From the results we see that ALL/COMBINED has similar speedup as an out-of-order processor. EACH and RANDOM/SEPARATE both show larger speedup on the in-order processor than out-of-order processor, especially HwT+SPOT. The permission check performance is more critical to overall performance because the slow software checks can't be hidden by out-of-order execution.

[4]Most of the pools created in vacation are outside of simulation region, only 42 linked lists are created during the simulation.

C. Impact of Page Size

The page size has a significant impact on the performance of HwT+SPOT because the POLB and SPOT store pool information at the page granularity. To understand this effect, we vary the page size among 4KB, 2MB and 1GB for SwT-Base and HwT+SPOT and show the performance in Figure 11.

Bench.	4KB Page	2MB Page	1GB Page
Linkedlist	0.45%	0.00%	0.00%
BST	3.80%	0.01%	0.01%
RBTree	2.56%	0.00%	0.00%
B-Tree	1.61%	0.00%	0.00%
B+Tree	1.36%	0.00%	0.00%
SPS	1.22%	0.00%	0.00%
TPC-C	3.35%	0.00%	0.00%
Vacation	6.08%	0.17%	0.17%

Table VII
POLB MISS RATE FOR DIFFERENT PAGE SIZE WITH RANDOM/SEPARATE ON HWT+SPOT ON OUT-OF-ORDER PROCESSOR

Since larger page size can also benefit the other data in the program, the speedup shown in Figure 11 is normalized to SwT-Base running with the same page size. Overall, a 2MB page size has a large improvement on the performance over the 4KB page size because it reduces the POLB miss rate and number of SPOT walks, as shown in Table VII. Larger page size can reduce the POLB miss rate and SPOT walk latency because most of the workloads can fit all of their persistent data in one large page leading to fewer misses on the POLB. On the other hand, the performance improvement from 2MB to 1GB is smaller because 2 MBs is enough to

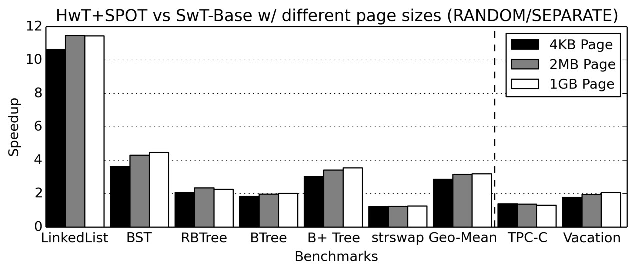


Figure 11. Performance of HwT+SPOT normalized to SwT-Base on architectures with different page sizes for the RANDOM/SEPARATE pattern.

hold the data in each pool, and in most cases, the POLB miss rate and number of SPOT walks does not drop any further. Thus we conclude that a 2MB page size is large enough to get most of the benefit from HwT+SPOT for our workloads.

D. Storage Overhead of SPOT

The memory size of SPOT is dominated by the last level tables. For the workloads used in our simulation with EACH/SEPARATE usage pattern, the total size of the SPOT used for each benchmark is summarized in Table VIII. The number is calculated by summing the size of all levels of the SPOT created by a workload. Since the Pool IDs are randomly generated and a workload tends to use a few pages in a pool, the lower-level tables have a similar number as other levels. With larger page size, the number of levels of table in the SPOT are reduced so the storage overhead decreases.

Bench.	4KB Page	2MB Page	1GB Page
Linkedlist	24 MB	24 MB	19 MB
BST	161 MB	161 MB	117 MB
RBTree	100 MB	100 MB	76 MB
btree	52 MB	52 MB	41 MB
bplustree	41 MB	41 MB	33 MB
strswap	37 MB	37 MB	29 MB
TPC-C	312 KB	312 KB	288 KB
Vacation	113 MB	113 MB	85 MB

Table VIII
THE SIZE OF TABLES ON ALL LEVELS OF SPOT USED IN OUR WORKLOADS, FOR 4KB, 2MB AND 1GB PAGE SIZE.

VIII. RELATED WORK

A. Permission Checks

Prior works pay little attention to the overhead of permission checks on relocatable pools. Mnemosyne [12] does not use the concept of relocatable pools so the persistent data can never be shared between programs and their kernel extension can make sure all the data owned by a program is mapped to the address space *a priori*. NV-heaps [11] are similar in concept to pools, and require programmers to open them with names before accessing them, potentially incurring the large overheads we observed. PMDK [17] also leaves the responsibility to programmers to open pools in advance. If a programmer fails to open the pool, translation will cause a program failure. The pool open interface performs permission checks and denies any illegal requests to open a pool.

Recent work [13], [15] has sought to reduce ObjectID translation overheads in PMDK through software and hardware support. Chen *et al.* [13] evaluated various low-level software mechanisms while Wang *et al.* [15] proposed to treat ObjectID as a persistent address space and provided hardware support for translating ObjectIDs. Both works assume programmers need to open the pools manually to ensure the pool is valid to access.

Manual calls to `pool_open` in these prior works can perform permission checks, but we argue that it increases the programming burden to call `pool_open` before any dereference to an object in an un-opened pool. Instead, our paper proposes hardware support to automatically perform permission checks while translating ObjectIDs. Programmers do not need to worry about opening pools in advance. At the same time, our design ensures permissions are obeyed.

Capabilities. Our notion of permissions checks are similar to concepts championed by capability architectures [30], [31]. Capabilities can not only enforce permissions across users but also between objects created by the same program. Capabilities [30], in general, are stronger than the permissions checks provided by our design.

B. NVMM File Systems and Persistent Stores

Prior works including BPFS [32], PMFS [33], SCMFS [34], Aerie [35], NOVA [36] and its variant NOVA-Fortis [37] propose persistent file systems on non-volatile memory. Persistent file systems provide direct access to NVMM with atomicity support and write-ordering primitives to ensure correctness and they serve as the foundation of existing persistent programming models [12], [11], [38]. The file-based abstraction is adopted in persistent programming models because the file API provides natural ways to create, delete, resize and rename persistent regions [39]. Our work also relies on persistent file systems to manage NVMM. However, we focus on providing hardware support for permissions checks. The SPOT design does not replace any file system data structure. It merely serves as a permission check and physical address mapping table that can be added to facilitate fast operations on persistent objects in the file system.

C. Virtual Memory and Page Table Design

The translation of ObjectID to physical address builds on prior work in virtual memory and page table design. Thus many of the ideas in the area of virtual memory and page table design should be reconsidered in the design and optimization of the SPOT [40], [41], [42].

IX. Conclusion

ObjectIDs may point to unmapped persistent regions, thereby requiring the programmer to reason about the permissions of objects and whether or not they are mapped. We remove this burden from the programmer by performing translation and permissions checks in hardware and by removing the requirement that objects be manually opened and mapped before being accessed.

To support it, we add a System Persistent Object Table (SPOT) that holds all known pools in the system, their physical address, and their permissions information. When a program attempts to access an unmapped or unchecked object, a privileged hardware SPOT walk finds the relevant entry, checks its permissions, and copies its entry into the POLB to allow translation and access all without trapping to the operating system. We have implemented our new design in a cycle accurate simulator and compared it with software only approaches and prior work [15]. We find that our design offers a compelling 2.9x speedup on average for microbenchmarks that access pools with the RANDOM pattern and 1.4x and 1.8x speedup on TPC-C and vacation, respectively, using the SEPARATE pattern.

Acknowledgements

We thank the ISCA 2018 reviewers for helping us improve this paper. This work was supported in part by NC State University and by the National Science Foundation (CNS-1717486). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Intel and Micron, "Intel and micron produce breakthrough memory technology," Jul. 2015.

[2] R. Rajachandrasekar, S. Potluri, A. Venkatesh, K. Hamidouche, M. W. ur Rahman, and D. K. D. Panda, "Mic-check: A distributed check pointing framework for the intel many integrated cores architecture," in *International symposium on High-performance parallel and distributed computing*, Jun. 2014.

[3] B. C. Lee, "Phase change technology and the future of main memory," in *IEEE Micro, Vol. 30, Issue: 1*, Jan. - Feb. 2010.

[4] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read," in *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, Feb. 2007.

[5] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[6] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," in *IEEE, Vol. 98, Issue: 12*, Oct. 2010.

[7] C. Wang, Q. Wei, J. Yang, C. Chen, and M. Xue, "How to Be Consistent with Persistent Memory? An Evaluation Approach," in *IEEE International Conference on Networking, Architecture and Storage (NAS'15)*, August 2015.

[8] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*, November 2013.

[9] M. H. Kryder and C. S. Kim, "After Hard DrivesWhat Comes Next?" September 2009, pp. 3406–3413.

[10] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[11] 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 *International conference on Architectural support for programming languages and operating systems*, Mar. 2011.

[12] 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*, ser. ASPLOS XVI, 2011.

[13] 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*. ACM, 2017.

[14] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *International symposium on Computer architecuture*, Jun. 2014.

[15] 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*. ACM, 2017.

[16] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017.

[17] N. L. T. at Intel, "Persistent memory programming," August 2016, http://pmem.io.

[18] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[19] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “Efficient persist barriers for multicores,” in *International Symposium on Microarchitecture*, Dec. 2015.

[20] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with whisper,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17, 2017.

[21] *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1*, Intel.

[22] A. Roth, “Store vulnerability window (svw): Re-execution filtering for enhanced load optimization,” in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’05, 2005.

[23] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, “Scalable load and store processing in latency tolerant processors,” in *International symposium on Computer architecture*, Jun. 2005.

[24] M. M. P. Leach and R. Salz, “A universally unique identifier (uuid) urn namespace,” July 2005, https://tools.ietf.org/html/rfc4122.

[25] 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 (TACO)*, 2014.

[26] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A distributed parallel simulator for multicores,” in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[27] “Tpc benchmark c,” Transaction Processing Performance Council (TPC), February 2010. [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[28] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “Stamp: Stanford transactional applications for multi-processing,” in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 35–46.

[29] Y. Lu, J. Shu, L. Sun, and O. Mutlu, “Loose-Ordering Consistency for persistent memory,” in *Computer Design, 2014 32nd IEEE International Conference on (ICCD’14)*, October 2014.

[30] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The cheri capability model: Revisiting risc in an age of risk,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14, 2014.

[31] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[32] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better i/o through byte-addressable, persistent memory,” in *ACM Symposium on Operating Systems Principles*, Oct. 2009.

[33] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys ’14, 2014.

[34] X. Wu and A. Reddy, “Scmfs: a file system for storage class memory,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 39.

[35] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, “Aerie: Flexible file-system interfaces to storage-class memory,” in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 14.

[36] J. Xu and S. Swanson, “Nova: A log-structured file system for hybrid volatile/non-volatile main memories.” in *FAST*, 2016, pp. 323–338.

[37] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, “Nova-fortis: A fault-tolerant non-volatile main memory file system,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, 2017.

[38] J. Zhou, Y. Shen, S. Li, and L. Huang, “Nvht: An efficient key-value storage library for non-volatile memory,” in *2016 IEEE/ACM 3rd International Conference on Big Data Computing Applications and Technologies (BDCAT)*, 2016.

[39] A. Rudoff, “Programming models for emerging non-volatile memory technologies,” *;login:*, vol. 38, no. 3, June 2013.

[40] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, “Hardware enforcement of application security policies using tagged memory,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08, 2008.

[41] A. Jaleel and B. Jacob, “In-line interrupt handling and lock-up free translation lookaside buffers (tlbs),” *IEEE Transactions on Computers*, 2006.

[42] I. Yaniv and D. Tsafrir, “Hash, don’t cache (the page table),” in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, ser. SIGMETRICS ’16, 2016.