Hardware Supported Persistent Object Address Translation

Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, James Tuck
North Carolina State University
{twang14,ssambas,solihin,jtuck}@ncsu.edu

ABSTRACT

Emerging non-volatile main memory technologies create a new opportunity for writing programs with a large, byte-addressable persistent storage that can be accessed through regular memory instructions. These new memory-as-storage technologies impose significant challenges to current programming models. In particular, some emerging persistent programming frameworks, like the NVM Library (NVML), implement relocatable persistent objects that can be mapped anywhere in the virtual address space. To make this work, persistent objects are referenced using object identifiers (ObjectID), rather than pointers, that need to be translated to an address before the object can be read or written. Frequent translation from ObjectID to address incurs significant overhead.

We propose treating ObjectIDs as a new persistent memory address space and provide hardware support for efficiently translating ObjectIDs to virtual addresses. With our design, a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. We also describe several possible microarchitectural designs and evaluate them.

We evaluate our design on Sniper modeling both in-order and out-of-order processors with 6 micro-benchmarks and the TPC-C application. The results show our design can give significant speedup over the baseline system using software translation. We demonstrate for the *Pipelined* implementation that our design has an average speedup of 1.96× and 1.58× on an in-order and out-of-order processor, respectively, over the baseline system on microbenchmarks that place persistent data randomly into persistent pools. For the same in-order and out-of-order microarchitectures, we measure a speedup of 1.17× and 1.12×, respectively, on the TPC-C application when B+Trees are put in different pools and rewritten to use our new hardware.

CCS CONCEPTS

• **Computer systems organization** → *Superscalar architectures*;
• **Hardware** → *Memory and dense storage*;

KEYWORDS

Non-volatile Memory, Persistent Memory, NVM Library

MICRO'17, October 14-18, 2017, Boston, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123981

ACM Reference format:
Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, James Tuck. 2017. Hardware Supported Persistent Object Address Translation. In *Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Boston, MA, USA, October 14-18, 2017 (MICRO'17),* 13 pages.
https://doi.org/10.1145/3123939.3123981

1 INTRODUCTION

Non-volatile memory technologies are under development and some are expected to compete with DRAM for use as a future main memory. For instance, Intel and Micron announced that their 3D Xpoint memory will be in the market in 2017 [13]. These new *non-volatile main memory* (NVMM) technologies are byte-addressable and have reasonably fast access latencies [1, 13, 15, 17, 18, 24]. Because NVMMs are accessed using regular loads and stores, programmers can store important data in *persistent* data structures in memory instead of serializing it to the file system. This raises many important questions about how programmers will access persistent regions of memory.

One area that needs more study is the mechanism that programmers will use to manipulate persistent objects. Ideally, programmers would access persistent objects the same way they access volatile memory, using pointers and references. In fact, many prior works assume precisely this with no modification. For example, Mnemosyne [27] proposes adding a new keyword to the C language, namely `persistent`, that allows some data to be placed in non-volatile memory regions. The assumption is that a segment of the virtual address space is reserved for persistent objects. However, assuming a fixed location within the address space prevents or restricts the use of Address Space Layout Randomization (ASLR) [2], an affordable and prevalent security mechanism in most current systems.

Alternatively, NVHeaps [7] and NVM Library (NVML) [25], two recently proposed libraries for persistent memory, allow objects to be relocated within the virtual address space. In these environments, programmers will open persistent regions or pools, much the way that files are opened on systems today, and they will be mapped into the virtual address space arbitrarily. Objects within the pools will not be referenced using addresses. Instead, object identifiers (ObjectIDs), consisting of a pool identifier and a byte offset within the pool, are used to identify and locate objects. ObjectIDs are general enough to access objects within the same pools or different pools, and can serve as the basis for building linked structures within and across pools. However, ObjectIDs require translation to a virtual address before they can be used to access a persistent object.

ObjectIDs enable persistent memory programming on current systems with ASLR. However, they pose significant challenges too. Translation of ObjectIDs is an onerous burden to place on programmers [21]. Anywhere an object is dereferenced, a translation is

needed. Given the already challenging problem of ensuring failure-safety [8, 14, 16, 20, 23, 25], this translation requirement adds a significant additional complexity to the code (see example in Section 2). Furthermore, translation can be a significant performance overhead. At a minimum, we need to perform a translation of ObjectID to base address by looking up the ObjectID in a table. Even if this translation is cached in a fast data structure, like in NVML, the overhead for frequently accessing this structure could be significant because it would require many instructions per translation.

We observe that the translation of ObjectIDs to virtual addresses is a problem that bears similarity to the problem of translating virtual addresses to physical addresses. We propose adding hardware and software support to enable ObjectID translation directly in the microarchitecture. In our design, programmers directly reference memory using ObjectIDs. We interpret ObjectIDs as a new address space layer that sits on top of virtual memory, and we provide new load and store instructions that directly access non-volatile memory using ObjectIDs.

We have studied the design of this architecture and make the following contributions. (1) We describe novel architectural and system-level extensions that enable translation of ObjectIDs to addresses, eliminating the need for programmers to directly translate ObjectIDs and the performance overhead of translation. (2) We design and evaluate critical microarchitectural choices as it relates to translation and disambiguation of addresses for persistent objects. We consider two different designs for Persistent Object Look-aside Buffer (POLB), one that translates to the virtual address and relies on a conventional Translation Look-aside Buffer (TLB) for conversion to the physical address, and another that works in parallel with the TLB and directly translates to the physical address. We also consider the design implications of these POLBs on an out-of-order superscalar processor's memory disambiguation logic. (3) We evaluate our design on micro-benchmarks and one database application, TPC-C, using a cycle-accurate microarchitectural simulator based on Sniper that is extended to support our microarchitecture. We demonstrate that the *Pipelined* design has 1.96× and 1.58× speedup, on average, over the baseline system on in-order and out-of-order processors, respectively, when persistent data are randomly placed in 32 persistent pools. We also attain a 1.17× and 1.12× speedup on the TPC-C application when different B+Trees are put in different pools. Finally, hardware translation reduces the dynamic instruction count by 43.9%, on average, compared to the baseline that uses software translation.

The rest of the paper is organized as follows. Section 2 gives an example of programming with an NVML like interface and demonstrates the challenges and sources of performance loss. Section 3 describes our overall architecture, software interface, and system-level assumptions. Section 4 describes the microarchitectural implementation of the POLB and how it interacts with other key structures. Sections 5 and 6 present our methodology and evaluation. Section 7 discusses related work, and Section 8 concludes.

2 MOTIVATION

We demonstrate the programming overheads and challenges of persistent memory programming using the programming interface shown in Table 1. We first describe the interface and how to use it.

Function Calls	Descriptions
Pool Management	
`pool* pool_create(name, size, mode)`	Create a pool with the specified `size` and associate it with a `name`.
`pool* pool_open(name)`	Reopen a pool that was previously created. Permissions will be checked.
`pool_close()`	Close a pool.
`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.
Durability	
`persist(oid, size)`	Make the region durable that begins at the ObjectID and ends `size` bytes later.
Failure Safety	
`tx_begin(pool* p)`	Start a new transaction related to the pool `p`.
`tx_add_range (oid,size)`	Take a "snapshot" of the current persistent data region referenced by the ObjectID and save it into the undo log.
`OID tx_pmalloc (size)`	Atomically allocate persistent data. Perform the functionality of `pmalloc()` but record it in the undo log in case the transaction fails.
`tx_pfree(oid)`	Free persistent data referenced by ObjectID within a transaction.
`tx_end()`	End the current transaction. If the transaction successfully ends, all the persistent data logged by `tx_add_range()` and `tx_pmalloc()` will be made durable.

Table 1: Summary of library calls required to support persistent memory programming.

Then, we present a code example and point out key overheads. We also discuss other overheads that are commonly presented in the context of logging for failure-safety.

2.1 The Persistent Programming Interface

Many designs for this interface are possible and the community has not settled on any design yet. We base our interface on the

NVM Library (NVML), an open-source library developed by the PMEM team at Intel that was inspired by prior works [7, 27]. We select it because it has garnered significant support from industry. However, because of its size and complexity, we have reduced it to an essential set of functions in order to give it C language semantics and to narrow the focus to specific aspects of the interface that are important to our work. We expect that the findings of our work extend to NVML and other systems with a design based on ObjectIDs.

Table 1 contains a number of functions that support a few keys tasks: pool management, object management, translation, support for durability, and support for failure-safety.

2.1.1 Pool Management. As in most prior works [7, 25, 27], we assume that persistent objects are grouped into file-like entities, called pools, that are mapped in their entirety into a process' address space. This is similar in concept to *mmap* on Linux. Pools are then created from scratch (`pool_create`), opened (`pool_open`), or closed (`pool_close`) just like files. Like files, opening or creating a pool would require an OS-level system call that verifies permissions and maps the pool into the address space. We do not further consider these mechanisms since they are similar, in principle, to ones that already exist for managing files.

However, this is where the analogy to files ends, because pools are not byte serialized and they contain objects. We assume, like prior work, that all pools must contain a root object, which serves as an entry point for the pool and aggregates key information for the pool. The last function shown under Pool Management is `pool_root`, and it provides access to the root object of a given pool.

2.1.2 Object Management. All objects within a pool are persistent, and we can allocate and deallocate them within the pool the same way that the heap works. We provide `pmalloc` and `pfree` functions. Unlike malloc, `pmalloc` needs to know the pool the object is being created within, and then it returns an ObjectID to the newly allocated object.

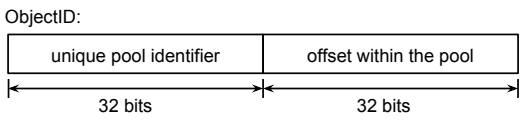

Figure 1: Components of an ObjectID.

As depicted by Figure 1, the ObjectID is the concatenation of a unique identifier given to the pool and an offset within the pool where the object is located. We assume that the ObjectID is 64 bits to make it possible to hold it in one register. We devote the upper 32 bits to the pool identifier, a unique system-wide number assigned to a pool when it is created, and the lower 32 bits for the offset. This can be interpreted as a segmented address space, where each segment is 4GB. However, it can also be interpreted as a flat address space since an object in one pool can reference any other pool using a legitimate ObjectID.

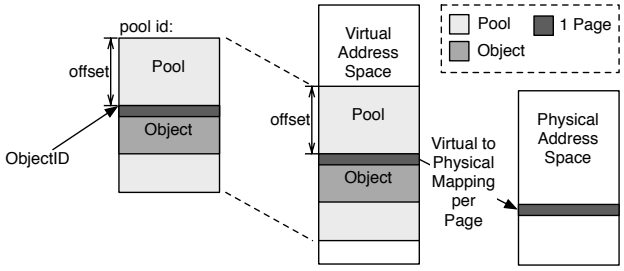

Figure 2: Relationship of pool, object, virtual, and physical address space.

2.1.3 Translation. As shown in Figure 2, each pool is mapped into the virtual address space, and each page of the pool is individually mapped to a physical address using conventional approaches for virtual memory. It's worth noting that the same pool might map to different virtual addresses in different processes. Furthermore, the mapping to physical address is handled by the virtual memory manager for the system in the conventional way.

An ObjectID can be converted to a virtual address as long as we know the location of the pool. In our API, such a translation is provided by the `oid_direct` function. Once a pool is mapped into an address space, it has a base address, and the relation of pool id to base address is tracked in an efficient data structure, like a hash table. For any given ObjectID, the translated address is obtained by searching the data structure for the unique pool id, returning the address where the pool is mapped, and then adding the offset.

NVML uses a combination of a hash table and a last value predictor to translate ObjectIDs. The most recent pool id look-up is held in a variable. If the next translation request is for the same pool, the base address is quickly determined without searching the hash table. Otherwise, if there is no match, the hash table is searched. Such an approach is efficient when translations tend to occur for the same pool, but it loses efficiency when ObjectIDs are spread across many pools.

We adopt this same strategy in our `oid_direct` implementation, illustrated as pseudo-code in Figure 3. We use a global variable pair, `most_recent_pool_id` and `most_recent_base_addr`, to hold the most recent translations. If the `pool_id` part of the ObjectID matches, the translation can be calculated with the most recently translated pool id. Otherwise, a search is performed on the hash table (`OIDTranlationMap` in the pseudo-code) to find the translation, and it also updates the most recently searched pair for use by subsequent translations.

If a translation is requested for a pool that has not been opened, it is considered an error. Ideally, the API should be extended with error codes to allow recovery or handling by the programmer, but we omit this support and further discussion for the sake of keeping the API small.

2.1.4 Durability and Failure-Safety. Durability and failure-safety are two major aspects of persistent programming, and many prior works have dealt with these topics [8, 14, 16, 20, 23, 25].

Durability refers to the process of ensuring that modified data has been written back to non-volatile storage. Intel's NVML extensions

```
void* oid_direct(OID oid)
{
    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);
    return most_recent_base_addr + oid.offset;
}
```

Figure 3: Pseudo-code of `oid_direct`. `most_recent_pool_id` and `most_recent_base_address` holds the most recent translated pool identifier and base address. `OIDTranslationMap` is a hash map that holds all the translations from pool id to base address.

describe new instructions, like CLWB and CLFLUSHOPT [12, 25], that force data to write back to memory. These instructions, in conjunction with SFENCE, allow the programmer to guarantee that data is written back to persistent memory. The `persist` function guarantees that all data starting at a given ObjectID and extending `size` additional bytes are written back to persistent storage.

Failure-safety deals with the larger problem of how to update a persistent data structure so that it is always in a consistent state or recoverable to one. Prior works have described how to use write-ahead undo or redo logging to provide transactional semantics when persisting data. We add support for write-ahead undo logging.

Transactions begin and end using the `tx_begin` and `tx_end` functions. Any persist object can be logged using the `tx_add_range` function which copies the specified range of addresses (ObjectID to ObjectID+size) to the log and makes them persistent. `tx_pmalloc` and `tx_free` are special versions of `pmalloc` and `pfree` that can be undone.

2.2 Persistent Linked List Example

Figure 4 shows an example of inserting a node and searching for a node in a linked list. First, consider the struct definition. Each node of the list contains a value and an ObjectID to the next node in the list. This design allows the list to be fully contained in a single pool or span multiple pools.

The `insert` routine takes a pool, head, and value as arguments. Next, it creates a new node in the specified pool. `pmalloc` returns an ObjectID so we have to convert it to an address before we can initialize the new node. Next, we set the next pointer to the head and return the new node as the head of the list. We have to be careful to set the next field using an ObjectID and to return the ObjectID rather than the translated address.

The `find` routine traverses a list looking for a node with a matching value. During each iteration a translation is required in order to access the elements of the node. Note that since the list could span multiple pools, we are assuming that all such pools have already been mapped using `pool_open` or `pool_create`.

2.2.1 Overheads. The persistent linked list incurs more overhead compared with a normal linked list. Translations from ObjectID to address are needed as compared to a conventional design. In

```
typedef struct {
  int value;
  OID next;
} node;
// insert a node that contains value
OID insert (pool* p, OID head, int value){
  OID new_oid;
  node* temp;
  assert(new_oid = pmalloc(p,sizeof(node)));
  temp = (node*)oid_direct(new_oid);
  temp->value = value;
  // link to OID of old head
  temp->next = head;
  head = new_oid;
  // return new OID
  return head;
}
// find the first node that matches data
OID find (OID head, int data){
   OID tmp = head;
   while (tmp != OID_NULL) {
     node *x = (node*)oid_direct(tmp);
     if (x->value == data)
        return tmp;
     tmp = x->next;
   }
   return OID_NULL;
}
```

Figure 4: Example of a persistent linked list insertion and search using our programming interface.

the event that a list is fully contained within the same pool, some translation overhead can be eliminated within the `oid_direct` function because it remembers the last one and can avoid a costlier search operation. Nonetheless, up to 100s of instructions could be needed to perform a translation depending on implementation and number of pools used by a program at any given time. Table 2 shows that the `oid_direct` function requires approximately 17 instructions when the last translation is reused (second column), but it increases to approximately 97 instructions, on average, when most translations require the cost of a full look-up (third column). Also, as translation must precede execution, these overheads fall on the critical path and have a major impact on performance.

Bench.	Insns on ALL	Insn on EACH	Miss on recent
LL	17.0	99.3	99.7%
BT	16.8	77.8	62.2%
RBT	17.0	104.7	91.0%
B+T	17.0	95.0	80.3%
BST	16.8	107.3	96.7%
SPS	17.0	102.7	99.9%
GeoMean	17.0	97.3	87.2%

Table 2: Average instructions executed in the `oid_direct` function on ALL and EACH benchmarks, as described in Section 5. Last column shows the miss rate for the most recent translation predictor on the EACH pattern.

These translation overheads also bring many indirect inefficiencies. For example, they increase the working set of the program,

adding to increased pressure on the cache, and the compiler may be less efficient since it cannot interpret ObjectIDs as addresses, limiting the success of many common optimizations.

2.2.2 Programmability Concerns. NVML [25] targets server class applications that primarily require system programming experts to explicitly manage persistent data. Hence, it makes little effort to simplify persistent programming. However, we argue that NVMM has the potential to be powerful on a wide range of systems, from client applications to embedded domains. Prior work has pointed out many of the other challenges of persistent programming [7, 21, 25, 27], like providing failure-safety and ensuring persistent objects are not leaked. Hence, given the already complex nature of programming for persistence, it is important to ease the burden on the programmer when possible.

Unfortunately, in the current API, implementing a simple data structure is somewhat more complex [21]. In the case of the linked list, rather than keeping track of the address for a node, it is imperative to track both the ObjectID and the translated address. Depending on the operation, one or the other may be required. Programming with pointers is already challenging and error prone, and ObjectIDs add another kind of reference that must be properly used by the programmer. Hence, rather than reasoning about one pointer, the programmer must reason about two, the normal address and the ObjectID.

Our hardware support can reduce this complexity by allowing a single reference, the ObjectID, to serve both purposes. To fully exploit the hardware, languages and compilers may need to be modified in order to expose this simplicity to the programmer. We do not consider these additional efforts in this paper, but we believe that such extensions are feasible.

3 DESIGN

We identify that translation from ObjectIDs to virtual address places a burden on the programmer and incurs significant performance overheads. We propose new instructions and architectural support to accelerate ObjectID translation.

3.1 New Instructions

In our design, we interpret the space of all ObjectIDs as a new address space and provide hardware support for loading and storing directly to this address space. Rather than requiring translation in software from ObjectID to virtual address, two new instructions will support translation in hardware, as shown in Table 3. `nvld` allows a load directly from an ObjectID. New hardware will look-up the ObjectID, convert it to a virtual address, and perform the load all in one instruction. Likewise, `nvst` will write to a location specified by an ObjectID.

While we do not show them explicitly, we assume that each instruction comes with variants for accessing data of varying widths (e.g. bytes, words, double words, and so on). Ultimately, the language and compiler would need to facilitate selection of the appropriate width operation, but we do not consider this issue more.

3.2 Overall Architecture

Figure 5 provides a depiction of our architecture. To support translation from ObjectID to virtual address when an nvld or nvst executes,

Instruction	Description
`nvld rd,rs1,imm`	rd = MEM[Lookup(rs1) + imm]
`nvst rs1,rs2,imm`	MEM[Lookup(rs2) + imm] = rs1

Table 3: New instructions to support translation directly in hardware.

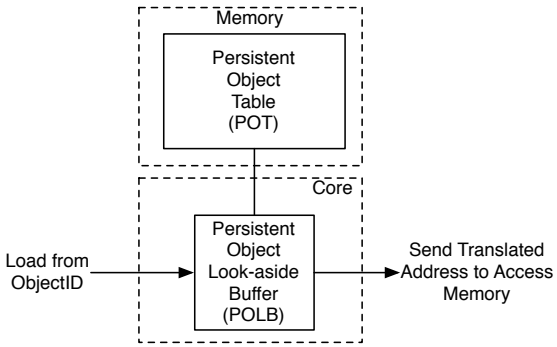

Figure 5: Overall architecture depicting Persistent Object Look-aside Buffer and Persistent Object Table.

we need two structures, the Persistent Object Look-aside Buffer (POLB) and Persistent Object Table (POT).

The POLB is similar to the TLB. It contains entries for the pools that have been mapped into a process's address space. The entry allows for translation from an ObjectID to an actual address, possibly virtual or physical (Section 3.2.1 will cover more details). The POLB is located inside the core and is meant to be accessed with low latency so as to incur very little overhead on nvld and nvst operations. If an entry is present, hardware is allowed to perform the translation.

The POT serves a similar role as a page table. The POT tracks the current pool mappings for a process. In our design, the POT converts a pool id to a virtual address.

The POLB and POT work together to provide high performance translation. If a translation is requested for an entry that is not present, the POT is checked to determine if the pool is mapped into the address space already. If a matching entry is present in the POT, it is simply moved to the POLB by hardware. Note that the POT serves as a backing store for all such information, whereas the POLB acts as a cache, holding only the most recent translations used by the core. However, if a requested translation is unknown to both the POLB and POT, it is treated like a page fault and a trap to the operating system is required to handle it, which may abort the program or invoke some form of signal handling to allow the application to recover.

3.2.1 Microarchitectural Considerations. While the design of the POLB and POT bear similarity to the TLB and Page Table, they are different. There are a variety of microarchitectural design choices. For example, should the POLB translate ObjectIDs to virtual addresses or physical addresses. If the POLB produces virtual addresses, that implies an additional *pipelined* access to the TLB. On

the other hand, if the POLB produces physical addresses, a cache access can proceed in *parallel* with the POLB look-up. However, this leads to other complexities.

Another aspect is the problem of aliasing among normal loads and stores and nvld and nvst. It is possible for code to simultaneously access a persistent object through both kinds of instructions. This implies that the architecture must disambiguate virtual addresses and references based on ObjectIDs at the same time. For high performance, memory disambiguation and load-store forwarding should operate correctly in the presence of aliased operations. For example, a store using an ObjectID should be able to forward data directly to a regular load.

Furthermore, these design choices are inter-dependent. We consider two overall design strategies for the POLB and POT, one that pipelines POLB accesses with the TLB and another that operates in parallel with the cache. Because these choices interact with memory disambiguation, we also consider these designs in the context of an out-of-order and in-order processor pipeline.

Their complete implementations are described in Section 4.

3.3 Supporting System and Library Modifications

We modify our programming interface from Section 2 with support for our new instructions and hardware.

In `pool_create()` or `pool_open()`, after the pool is created or verified, the persistent data region will be mapped into the process' address space. Also, the POT must be updated with a new entry for the pool.

Now, `oid_direct()` is no longer required. A programmer can dereference an ObjectID to read or write persistent data directly. Several of the library functions can also benefit from hardware acceleration. For example, tx_add_range() can use these more efficienc instructions to when copying data into undo logs. We modify all of the library calls that access persistent data to use faster nvld and nvst instructions.

4 MICROARCHITECTURE

We present the microarchitectural details of our new structures and how they integrate into the pipeline.

4.1 Persistent Object Look-aside Buffer

The POLB is implemented as a small cache-like structure. Each entry in the POLB contains a tag, a valid bit, and a data value. The tag array is implemented as a Content Addressable Memory (CAM) for efficient searching.

4.1.1 Pipelined. In the case of the *Pipelined* design, the POLB stores the translation from the pool identifier to the base address. In this case, the tag will hold a pool identifier, and the data field holds a 64-bit virtual address. On a lookup, the pool identifier from an ObjectID is checked against the tag array. If a valid entry is found, the corresponding base address is obtained and added to the offset field of the ObjectID. The resulting virtual address is sent to the TLB and L1 data cache to access the memory hierarchy. Figure 6(a) shows the process of translation for *Pipelined*.

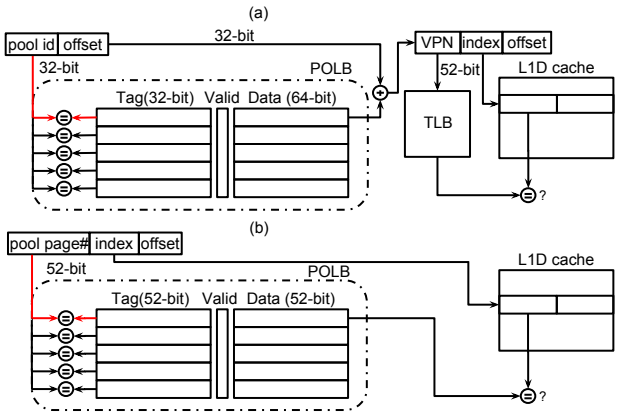

Figure 6: Two designs for the Persistent Object Look-aside Buffer.

4.1.2 Parallel. For *Parallel* the fields of the POLB have a different purpose because the goal is direct conversion to a physical address. In this case, we need to know both the pool identifier and the page within the pool that is being referenced. This requires taking some of the bits from both the pool identifier and the offset of the ObjectID. We assume 4 KB pages and a virtually-indexed physically-tagged cache, so the lower 12 bits can be forwarded directly to the L1 data cache. The upper 52 bits of the ObjectID are used to search the POLB for a match. If a valid entry is found, a 52 bit physical address is obtained and used for a tag comparison with the L1 data cache. Figure 6(b) shows the process of translation for *Parallel*.

4.1.3 Comparison. *Pipelined* imposes a delay for translation from ObjectID to virtual address before accessing the cache. It's unlikely the POLB delay can be added with no impact on load latency. *Parallel* can avoid this delay by operating in parallel. This appears to lean in favor of *Parallel* in the absence of other microarchitectural considerations.

On the other hand, the size of the POLB in *Pipelined* only needs to be big enough to track all pools currently mapped in the process's address space, whereas *Parallel* needs to track all active pages in the pool. Some of these pages may be redundantly tracked in the TLB, since we allow regular loads and stores to the same virtual addresses that nvlds and nvsts are accessing through ObjectIDs.

If the added cost of the *Pipelined* design can be hidden or otherwise avoided, it may be preferable given its smaller size and that it avoids duplicate mappings with the TLB.

4.2 Persistent Object Table

We design the POT under a different set of assumptions than a typical page table. If we assume that pools are like files, then it is reasonable that hundreds to thousands of files may be the limit that is needed. Many Linux systems limit the number of open files by a single process to 1024 or less, which implies that a relatively small table would suffice.

The POT is accessed on a POLB miss, as described in Section 3, to look-up entries that need to be placed in the POLB. The POT

must contain the information needed to determine if the access is legal and to fill in an entry in the POLB. Each entry in the POT contains a pool identifier and a corresponding virtual base address for the pool. We reserve a pool id of 0 to indicate a NULL pool which cannot exist. This allows us to initialize all entries as invalid.

For both *Pipelined* and *Parallel*, we design the look-up of the POT based on the X86 architecture's hardware page table walk. Figure 7 depicts the POT walk. We presume a new architectural register can be used to hold the base virtual address of the POT in memory. The pool identifier from the referenced ObjectID will be taken and calculated through a hash function to get the index within the POT. A legal translation occurs when the POT entry is valid and the pool id in the entry matches the one in the ObjectID.

If a legal entry is found that does not match, linear probing is used to find a match. If an invalid entry is encountered, it means the translation is missing in the POT, and an exception is raised. The OS may abort the program due to an illegal access or allow the program to recover in some way. (For example, a signal handler may establish a legal translation by proper use of the pool creation/opening interface.)

For *Pipelined*, the process is complete after finding a match in the POT. The pool identifier and virtual base address are copied to the POLB and the valid bit in the POLB is set.

In the case of *Parallel*, the process is not yet complete because a physical address has not yet been determined. We could redesign the POT for the sake of *Parallel* so that it could provide the physical page. However, given the assumption of few pools, we choose not to replicate the page table. We opt instead to use the same POT design for both *Parallel* and *Pipelined*. Hence, after the POT walk, the base address found in POT and the offset in ObjectID will be used to perform a page table walk to find the physical frame number (PFN). Then, a new entry can be created in the POLB with the pool identifier and the PFN. Given the need to walk both the POT and the page table, we expect the POLB miss for *Parallel* to take longer to resolve than one in *Pipelined*. The lower part of Figure 7 shows the additional page table walk required for *Parallel*.

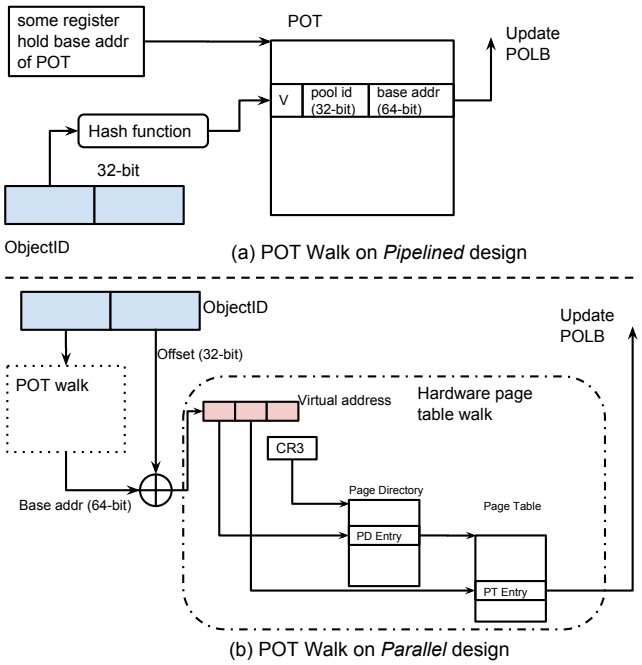

Figure 7: Hardware-based POT walk.

4.3 Memory Disambiguation

nvld and nvst introduce new challenges for enforcing memory dependencies and for load-store forwarding. These instructions reference memory using an ObjectID but regular loads and stores use virtual addresses. Conventional memory disambiguation logic in a processor core would not be able to determine whether an ObjectID and regular address alias. Only after converting the ObjectID to a virtual address would such a dependence enforcement be possible.

For *Pipelined*, it is reasonable to convert an ObjectID using the POLB before filling the nvld's or nvst's address in the Load Store Queue (LSQ). This would require placing the POLB in the address generation pipeline stage. This stage occurs between wake-up and before execution so as to give load instructions a cycle to compute their effective address. It would also ensure that the LSQ can perform memory disambiguation and forwarding correctly because it would only ever see virtual addresses.

For *Parallel*, the POLB must be placed in parallel with the cache access since it produces a physical address. However, we cannot allow the LSQ to hold ObjectIDs for pending nvld and nvst operations, without some modifications. An ObjectID could happen to have the same value as an address currently operated on by the program, leading to a false dependence edge. In the case of a store address that happens to match an nvld's ObjectID (or vice versa), we might perform erroneous store-load forwarding. To prevent this, at a minimum, we would need to mark each entry in the LSQ with a bit indicating whether the entry's address is an ObjectID or not, to properly enforce dependencies and forward data. However, this still does not cover the case of loads bypassing aliased stores. To detect when ObjectID's alias with other normal loads or stores, we would need additional hardware.

Given the design complexity of the *Parallel* design, we choose not to consider it further for out-of-order execution. We do compare *Parallel* and *Pipelined* in the context of an in-order core (Section 4.5). However, it's worth pointing out that techniques for aggressive load speculation may be helpful for implementing *Parallel* on an out-of-order processor.

4.4 Out-of-Order Pipeline

We integrated the *Pipelined* design for the POLB and POT into an aggressive out-of-order superscalar pipeline. nvld and nvst flow through the pipeline like regular load and store instructions. When they are decoded, they are reserved an entry in the LSQ in program order. However, at this time, their source operands are not yet known. For a load, that is only the ObjectID, but for a store it includes both the ObjectID and the data operand.

The nvld and nvst instructions wait in the Issue Queue (IQ) until their source operands are ready. When woken up, they read their source registers and then subsequently their address is calculated

through an address generation (AGEN) stage. In this stage, the POLB is accessed, the ObjectID is translated to a virtual address, and the virtual address is placed in the LSQ entry for the instruction. The LSQ enforces memory dependencies and ordering with these virtual addresses in the usual way.

If the lookup on the POLB misses, the Address Generation Unit stalls for the POT walk to complete. If it finds a matching entry, the entry is added to the POLB and the ObjectID translation is forwarded to the LSQ. In the event of a POT miss, the Re-order Buffer entry for the instruction is marked as raising an exception. Once it reaches the head of the re-order buffer, the exception is raised, flushing the pipeline, and the control is handed over to the OS for handling.

Figure 8 illustrates an example of how nvld and nvst instructions are executed on out-of-order pipeline. Assume the instructions are already decoded, renamed and dispatched to the IQ. In the example, the nvld instruction is ready to execute with its operand `p70` ready. It flows into Register Read (RR) stage to read register file which stores an ObjectID of `(1234,10)`. Then in Address Generation (AGEN) stage, the ObjectID is calculated first and it's still `(1234,10)` because of the immediate being 0. It then goes through POLB to obtain base address of a pool (`5678`) and then calculates the actual virtual address of the access (`5688`). Then the virtual address is placed in LSQ for memory disambiguation.

4.5 In-order Pipeline

We also implement on a traditional five-stage pipelined processor: instruction fetch (IF), instruction decode (ID), execution (EX), memory accesses (MEM) and write back (WB).

We assume the processor is updated to support our new instructions. In the EX stage, the ObjectID may go through additional arithmetic logic to perform offset related calculations to determine the final ObjectID, like addresses in memory access instructions. In the MEM stage, they access memory. As mentioned in Section 3, for *Pipelined*, the ObjectID first goes through the POLB to calculate the virtual address and then accesses the TLB and data cache in parallel. For *Parallel*, the POLB is accessed in parallel with the cache.

In the case of a POLB miss, the in-order pipeline stalls until the POT walk is completed. If an POT miss occurs, the instruction raises an exception, flushes the pipeline, and hands control to the OS for handling.

5 METHODOLOGY

5.1 Simulation

We extend Sniper 6.1 [3], a cycle-accurate X86 simulator, to model our microarchitecture. We adopt Sniper's most recent timing model for the Instruction Window-Centric core model (ROB core model) to perform both in-order and out-of-order simulations. The simulator models a QuadCore Intel Xeon X5550 Gainestown (Nehalem-EP) processor shown in Table 4. We model the in-order processor using the same frequency and architecture.

We use Pin as the front-end for Sniper. We did not modify the x86 ISA or compilers to compile new instructions. Instead, to emulate nvld and nvst, we use normal loads and stores but assign ObjectIDs in such a way that they are clearly distinguishable from the rest of the program's address space. When we identify loads and stores to persistent memory, we mark them and execute them according to our microarchitectural description. This strategy is beneficial because it allows the compiler to heavily optimize the code, demonstrating the full benefit of programming directly with ObjectIDs rather than translating them in software.

Component	Configuration
In-order core	4 cores, X86-64, in-order, 2.66GHz, Branch predictor: Pentium M, Branch misp.: 8 cycles, page size: 4KB, cache block size: 64 Bytes
Out-of-order core	4 cores, X86-64, out-of-order, 2.66GHz, Issue width: 4, LQ: 48, SQ: 32, ROB: 128, Branch predictor: Pentium M, Branch misp.: 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)
POLB	POLB access: 3 cycles (1ns), POT walk: 30 cycles (11ns)
clwb latency	100 cycles (38ns)
DRAM	1GB, 120 cycles (45 ns)
NVMM	Battery-backed DRAM, 120 cycles

Table 4: Architecture Configuration.

The POLB size we choose, by default, is a 32-entry CAM according to our sensitivity analysis in Section 6.3. It results in a 32*32/8=128 bytes tag array and 32*64/8=256 bytes data array in *Pipelined* design, and *Parallel* processors, and 32*52/8=208 bytes tag and data array respectively in *Parallel* design. We model the tag look-up and virtual address translation to be 1 ns, i.e. 3 cycles. Sniper does not contain a model for a detailed page table walk, instead it charges 30 cycles for the TLB miss penalty. We also model a fixed 30-cycle POLB miss penalty on *Pipelined*, since it would require, on average, one access to memory. We charge a 60-cycle miss penalty on *Parallel* to estimate the combined POT walk and page table walk. For our design, we believe this is fairly pessimistic since the POT entries would likely hit in the cache and be serviced in a short period of time. However, we perform a sensitivity study to determine the impact of much longer POT walk latencies in Section 6.3.

We set the POT size per process to 16384 entries, because we do not expect programs in the near future to leverage a large number of pools. Such a POT requires 256 KB of memory.

We pessimistically model the CLWB instruction for making data durable with a fixed 100 cycle latency, estimated by the CLFLUSH latency on a similar X86 architecture [11].

As for the library extension, we implement all the APIs summarized in Table 1. We implement two versions of the library, one that exploits our new instructions and one that does not. Whenever

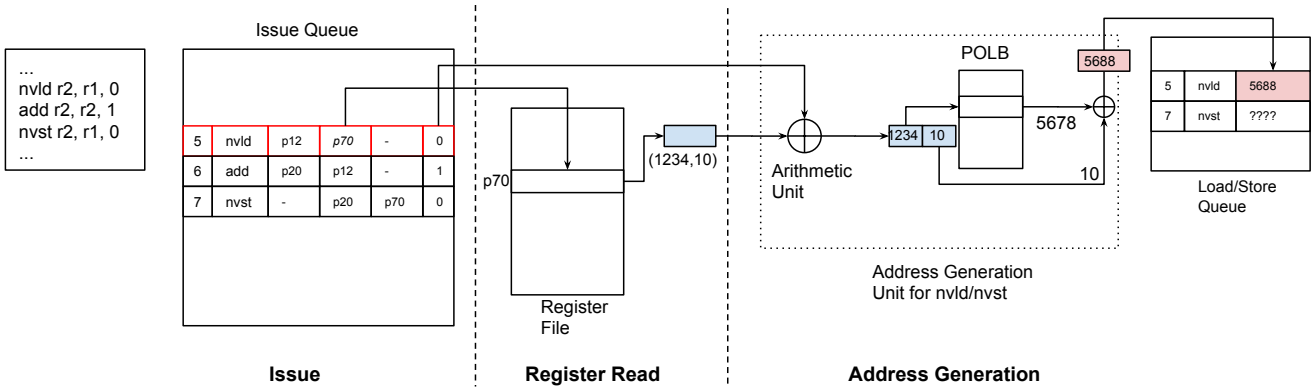

Figure 8: An example of hardware supported translation on an out-of-order processor with *Pipelined* design. The code snippet disassembles a simple C expression, a++, but the variable a is held in a persistent pool. The data field of the LSQ is not shown.

Name	Abbr.	Description
Linked-list	LL	Search 700 random integers in the linked-list. If the number is found, remove it. Otherwise, insert a new node with the number as key
Binary Search Tree	BST	Search 5000 random integers in the binary tree. If the number is found, remove it and replace the node with maximum key on its left sub-tree. Otherwise, insert a new node with the number as key.
String Position Swap	SPS	Randomly swap a pair of strings in a 32KB string array and repeat 10000 times.
Red-black Tree	RBT	Search 3000 random integers in the RB-Tree. If the number is found, remove it and re-balance the tree according to the red-black rule. Otherwise, insert a new node with the number as key and also re-balance the tree.
B-Tree (order=7)	BT	Search 5000 random integers in the B-Tree. If the number is missing, insert a new node with the number as key and the tree will be re-balanced.
B+ Tree (order=7)	B+T	Search 5000 random integers in the B+ Tree. If the number is found, remove it. Otherwise, insert a new node with the number as key. Both insertion and deletion need to re-balance the tree.
TPC-C	TPCC	Generate 1 warehouse according the parameters in TPC-C spec [26] and perform 1000 transactions.

Table 5: Summary of Workloads.

we run a workload on our proposed hardware, we use the optimized version of the library. Otherwise, we use the software-only implementation.

5.2 Workloads

We evaluate our design with six micro-benchmarks and the TPC-C[26] application listed in Table 5. Four of the micro-benchmarks (LL, SPS, BT and RBT) are similar to prior works [7], but they use our library calls to allocate persistent data structures and maintain durability and failure-safety. The B+ Tree benchmark is derived from the core structure of the TPC-C application. In TPC-C, we move the data structures in the form of a B+ Tree to persistent pools. Also we retain TPC-C's own failure-safe logging implementation without modification for persistence.

We develop three different pool usage patterns for each microbenchmark, namely ALL, EACH and RANDOM, summarized in Table 6. This approach allows us to evaluate a variety of access patterns on our microbenchmarks and understand their implications on our proposed hardware.

We run each benchmark on four different configurations: BASE, OPT, BASE_NTX, OPT_NTX, summarized in Table 7. BASE is fully software without our hardware support. It allocates the persistent data structures using the APIs, and it provides failure-safety through the logging and `persist()` API. OPT also allocates persistent data but instead of using `oid_direct()` to translate from ObjectIDs, all reads and writes will be directly performed with ObjectIDs. It also guarantees atomicity and durability of updates. We also compare to configurations without failure-safety and durability support. These are named BASE_NTX and OPT_NTX. They do not provide failure-safety but help determine the performance impact from atomicity and failure-safety. This is important to consider because logging code also uses ObjectID translation and extracts some benefit from our new hardware, but it also introduces significant overheads in the form of logging code and persists.

Each benchmark used in the paper may have any combination of pool usage pattern and architectural configuration, e.g. executing the Linkedlist benchmark with the EACH pattern and OPT configuration means that all nodes to be inserted in the linkedlist will be allocated a new pool and hardware translations are used to access persistent objects.

6 EVALUATION

6.1 Benchmark Performance

First, we look at the overall performance improvement provided by our proposed architectures. We start with the results for the

Patterns	Description
ALL	All persistent data are in one pool.
EACH	Each structure created by a program is put in separate pools.
RANDOM	The number of pools fixed to be 32 and newly created structure will be allocated in a pool indexed by the key of structure modulo by 32.
TPCC_ALL	All B+-Tree-based structures in TPC-C benchmark are allocated in one pool.
TPCC_EACH	Each B+-Tree-based structure is put in separate pools.

Table 6: Pool usage access patterns.

Configuration	Description
BASE	No hardware support for translation. Includes support for failure-safety and durability.
OPT	BASE optimized to use our hardware support.
BASE_NTX	BASE without support for failure-safety and durability.
OPT_NTX	OPT without support for failure-safety and durability.

Table 7: Combined architecture and benchmark configurations.

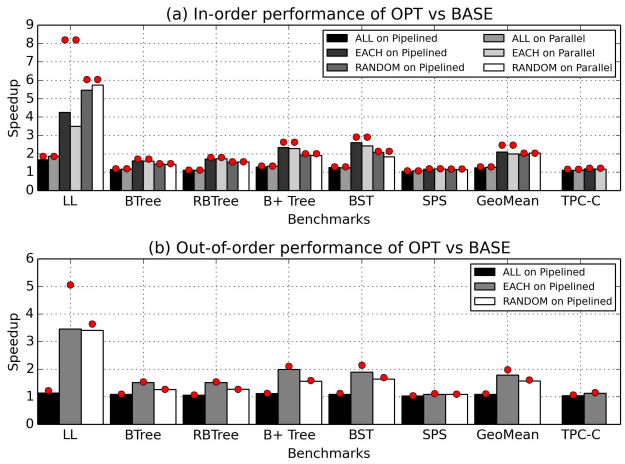

Figure 9: Speedup of OPT over BASE for each usage pattern on in-order (a) and out-of-order (b) designs. The red dots show the speedup on an ideal architecture where there's no penalty to perform a hardware translation.

in-order processor shown in Figure 9(a). Here, we show the speedup of OPT over BASE for each benchmark. This comparison showcases the advantages of hardware translation. For each usage pattern, we evaluate both *Pipelined* and *Parallel* to see the impact of each architecture. We also place a red dot over each bar to show best possible performance gain if using an architecture with no hardware translation overhead.

First, we consider the performance of the different usage patterns. For the RANDOM pattern, the BASE design will incur many slow translations using the look-up table since pool accesses occur randomly and are unpredictable, and this means our architectures can offer significant speedups. *Pipelined* attains a 1.96 × average speedup, and *Parallel* attains a 1.92 × average speedup. ALL shows the minimal speedup among the three patterns because it uses only one pool. The BASE works well because it is able to leverage its last translation to avoid a look-up, and that reduces the benefit of hardware translation. ALL results in the minimum number of dynamic instructions among the three.

In the EACH pattern, all objects are placed in different pools. One might conclude that this pattern would attain the best speedup on our architecture. However, that is not always the case. There is a performance trade-off between the number of translations saved and the number of misses in the POLB. EACH and RANDOM benchmarks require significantly more translations, which gives more speedup than ALL. Comparing EACH and RANDOM, the one that achieves better performance depends on the predictor miss-rate in the BASE version versus POLB contention caused by an increasing number of pools. According to Table 8, if a benchmark doesn't create large contention on a 32-entry sized POLB in EACH, it should have a bigger speedup for EACH than RANDOM. On the other hand, LL has a large miss rate on the POLB compared to RANDOM, so the miss penalty of the POLB negates some of the benefits of eliminating software look-ups. Thus RANDOM is better than EACH for LL.

Now consider the differences between *Pipelined* and *Parallel*. *Pipelined* performs better than *Parallel* in all benchmarks. This suggests that the extra cycles spent on POLB accesses are a smaller overhead than the additional POLB miss penalty for *Parallel*. Table 8 shows the POLB miss rate for both *Pipelined* and *Parallel*. *Pipelined* has much lower miss rate than *Parallel*, and only has 1 and 32 misses (0.00%) on ALL and RANDOM benchmarks respectively. From this analysis, the *Pipelined* appears to be the better choice given its simplicity and performance advantage. Overall, both are reasonable approaches.

The TPC-C application gives additional perspective on the importance of our hardware, because the translation overheads are incurred within a larger application, not a kernel with frequent translations. We measure a 1.10 × speedup on TPCC_ALL and a 1.17 × speedup on TPCC_EACH, respectively. Even in this application, the speedups are significant.

We also evaluate the ideal performance attained where there's no penalty to access the POLB and no miss penalty on the hardware POT walk, thereby measuring a theoretical upper-bound for our hardware. These results are shown as the dot above each bar. The results show that our *Pipelined* and *Parallel* design, although they introduce overhead, can compete with the ideal for most usage patterns and workloads. The notable exception is LL on the EACH pattern. In this case, even though the total number of pools is not the largest among our workloads, each operation on the linked list involves a traveral starting from the head of list, which leads to poor temporal locality and high POLB contention.

Figure 9(b) shows the speedup of the *Pipelined* design on out-of-order processor. Overall, it shows a similar trend across the usage patterns as the in-order architecture. A key difference is that it

Bench.	Parallel			Pipelined
	ALL	RANDOM	EACH	EACH
LL	0.0%	0.5%	32.5%	32.4%
BST	0.6%	2.8%	8.1%	7.3%
RBT	0.6%	2.5%	4.1%	3.1%
BT	0.3%	1.4%	2.5%	1.7%
B+T	0.0%	1.0%	2.5%	1.5%
SPS	0.0%	1.2%	2.4%	1.2%
TPCC	2.3%	-	2.5%	0.0%

Table 8: POLB miss rate of OPT benchmark pattern on *Pipelined* and *Parallel* design respectively. *Pipelined* only shows the results of EACH here, because ALL and RANDOM with *Pipelined* only misses 1 and 32 times (0.000%) during POLB warm-up respectively.

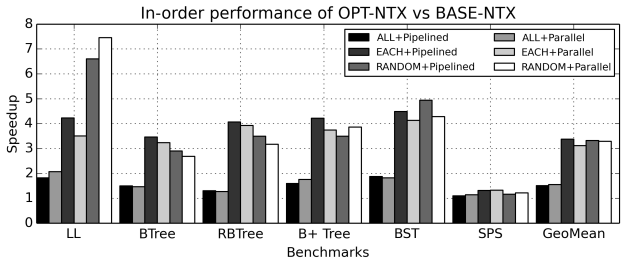

Figure 10: Performance of all OPT_NTX benchmarks without persistence and transaction supports on both Pipelined and Parallel designs. The results are normalized to the BASE_NTX benchmark running on baseline system.

shows less speedup than in-order because out-of-order execution hides some of the overhead of slow software translation because it extracts more instruction level parallelism.

There are a couple of workload specific behaviors worth pointing out. LL has more speedup than the tree structures because of more translations that are on the critical path, and LL has longer chains of accesses than tree structures. SPS, on the other hand, is array-based and does not need a translation per element in the BASE case. Thus it has the least speed up on both in-order and out-of-order implementations.

6.2 Overhead of Durability and Atomicity

We also evaluate our workloads without persistence and atomicity support. The OPT_NTX and BASE_NTX are evaluated on both the *Pipelined* and *Parallel* designs. The results on the in-order processor are shown in Figure 10 normalized to BASE_NTX. The speedup on both designs are higher than the prior case with persistence and atomicity support. One reason for this difference is that without the added overhead of logging which requires additional translations the working set of pools is small enough that each pool can fit within just one page. Hence, the *Parallel* design only needs one entry per pool in the POLB, which reduces the contention and makes *Parallel* faster than *Pipelined* on EACH. However on the RANDOM pattern, *Pipelined* still has more speedup than *Parallel*.

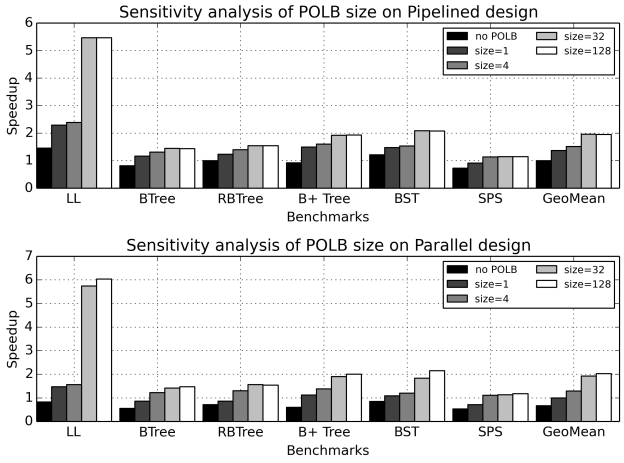

Figure 11: Sensitivity analysis to POLB size. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on Pipelined and Parallel design on in-order processor, with no POLB and POLB size=1, 4, 32, and 128, respectively.

Bench.	Pipelined				Parallel			
	1	4	32	128	1	4	32	128
LL	31.7%	28.3%	0.0%	0.0%	32.1%	28.4%	0.5%	0.0%
BST	19.7%	16.4%	0.0%	0.0%	25.9%	17.6%	2.8%	0.0%
RBT	36.9%	8.1%	0.0%	0.0%	58.7%	11.5%	2.5%	0.0%
BT	17.4%	4.7%	0.0%	0.0%	29.4%	6.4%	1.4%	0.0%
B+T	8.7%	3.8%	0.0%	0.0%	18.7%	5.0%	1.0%	0.0%
SPS	40.8%	1.1%	0.0%	0.0%	48.1%	2.3%	1.2%	0.0%

Table 9: POLB miss rates on OPT_NTX using the RANDOM usage pattern on *Pipelined* and *Parallel* designs while increasing the size of the POLB.

Pipelined does not miss on the POLB (except during warm-up) when running a RANDOM pattern, but *Parallel* has more pages of data within one pool and that results in more POLB contention especially in comparison to the ALL pattern.

6.3 Sensitivity Analysis: POLB size

We select a fixed POLB size of 32 entries on the previous studies. In this section, we perform a sensitivity analysis on the POLB size for workloads with RANDOM patterns. By design, these workloads use 32 pools. We sweep the number of POLB entries from 1 to 128. We also include the results when there is no POLB in the system forcing all hardware translations to perform a POT walk.

The results are shown in 11. We see that without a POLB most workloads slow down. Also, the speedup saturates at a specific POLB size once the POLB is large enough to eliminate most misses. The *Parallel* design requires a larger POLB than *Pipelined* because it incurs more conflict misses, as shown in Table 9. We can surmise that a 32-entry POLB will be sufficient for many workloads, under the expectation that programmers will not operate on hundreds of pools in near future applications.

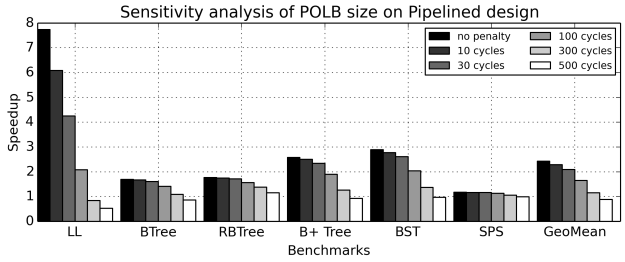

Figure 12: Sensitivity analysis to POT walk penalty. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on the in-order *Pipelined* design.

6.4 Impact of Hardware POT Walk

In our model, we assume that the hardware POT walk has a fixed 30-cycle latency. In this section, we estimate the impact of the POT walk by varying the fixed latency from 10 cycles, 30 cycles, 100 cycles, 300 cycles, and 500 cycles. We choose 10 cycles and 30 cycles because we expect caching to reduce the penalty of POT accesses. We sweep higher POT latencies to explore their impact on performance. We also evaluate the results with an ideal system that charges no POT walk penalty. The performance speedups are shown in Figure 12.

POT walk latency has a higher impact on workloads with a high POLB miss rate, namely LL. We can conclude that an actual hardware POT walk averaging around 30 cycles would have a small impact on the performance of the system, but an even smaller POT walk penalty will benefit benchmarks with higher POLB miss rates, namely LL.

7 RELATED WORK

Prior works have considered a variety of challenges related to programming for persistent memory [4, 5, 7–10, 14, 16, 20, 22, 23, 25, 28]. Many works have focused on the challenge of failure-safety, resulting in the description of a variety of persistency models [8, 14, 16, 20, 23, 25] and how to use them.

Other works have focused on how to create persistent regions of memory and update them safely. For example, Mnemosyne [27] proposes adding a new keyword to the C language, namely `persistent`, that allows some data to be placed in persistent memory regions or segments, and it uses software transactional memory to atomically update them. Furthermore, these persistent segments must be mapped to the same part of the virtual address space in all processes that use them. This is problematic, in the general case, since it may be difficult to know in advance all the segments that will be mapped in a program *a priori*. Furthermore, it hinders the use of Address Space Layout Randomization, an affordable and prevalent security mechanism.

NVHeaps [7] provides object level abstractions that allow programmers to construct non-volatile objects by deriving from a predefined base class, called `NVObject`, and then accessing them through special NV-pointers. NVML [25] builds on the advances of NVHeaps and Mnemosyne; a programmer translates persistent ObjectIDs to access data.

However, to our knowledge, no prior work has considered the overhead of supporting relocatable persistent objects nor proposed hardware or software optimizations to enhance performance of translation from ObjectIDs to virtual addresses. Concurrent with our work and highly related, Chen *et. al* [6] describe compiler and language level mechanisms to reduce the overhead of supporting position independence. However, their designs show higher overheads in the presence of many pools and may benefit from hardware translation to reduce that overhead.

Prior capability-based systems [19] and segmented-memory architectures, like the Burrough's 5500 and Intel iAPX 432, were also described as object-based systems, and they supported object-based operations and references. In such systems, all of memory was viewed as an object, and operations on objects were encapsulated and protected. That is not the case in our system. ObjectIDs are only used to reference persistent objects and hardware is added only to support efficient translation of persistent object addresses. All other operations on persistent objects are performed in software without kernel help or encapsulation. However, some of the features of capability-based systems may be worth revisiting in future systems with persistent memory.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose treating ObjectIDs as a new persistent memory address space and provide hardware support for efficiently translating ObjectIDs to virtual addresses. With our design, a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. We also describe several possible microarchitectural designs and evaluate them. The *Pipelined* approach is preferred for out-of-order processors because the address translation operation needs to occur before memory disambiguation. Fortunately, this design choice did not significantly hurt the performance potential of our approach, given the large savings of removing frequent software translation from the program. The *Pipelined* design offers a small advantage over *Parallel* on in-order cores due to its lower miss rate and lower miss penalty. On microbenchmarks that place persistent data randomly into persistent pools, the *Pipelined* implementation has an average speedup of 1.96× and 1.58× on an in-order and out-of-order processor, respectively, over the baseline system.

Future work should further consider the design of the POLB and POT as larger programs are written and as interfaces evolve. In particular, the size of the POT and its required coverage of the persistent memory address space will need to be analyzed on large applications once they are developed. Language and compiler level support also need further study.

ACKNOWLEDGEMENTS

We thank the MICRO-50 reviewers for helping us improve this paper. Solihin's work was supported by (while serving at) the National Science Foundation. 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] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. In *IEEE, Vol. 98, Issue: 12.*

[2] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1251353.1251361

[3] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677

[4] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014), 433–452.

[5] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.

[6] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *International conference on Architectural support for programming languages and operating systems*.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles*.

[9] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 15.

[10] Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Bridging the programming gap between persistent and volatile memory using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 30.

[11] InstLatX64. 2017. x86, x64 Instruction Latency, Memory Latency and CPUID dumps. (March 2017). http://users.atw.hu/instlatx64/

[12] Intel. 2016. *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[13] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology. (Jul. 2015).

[14] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *International Symposium on Microarchitecture*.

[15] 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. 2007. 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*.

[16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *International conference on Architectural support for programming languages and operating systems*.

[17] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *IEEE International Symposium on Performance Analysis of Systems and Software*.

[18] Benjamin C. Lee. 2010. Phase change technology and the future of main memory. In *IEEE Micro, Vol. 30, Issue: 1.*

[19] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA.

[20] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *Computer Design, 2014 32nd IEEE International Conference on (ICCD'14)*.

[21] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/marathe

[22] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. 2013. A case for efficient hardware/software cooperative management of storage and memory. (2013).

[23] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *International symposium on Computer architecuture*.

[24] Raghunath Rajachandrasekar, Sreeram Potluri, Akshay Venkatesh, Khaled Hamidouche, Md. Wasi ur Rahman, and Dhabaleswar K. (DK) Panda. 2014. MIC-Check: A distributed check pointing framework for the Intel many integrated cores architecture. In *International symposium on High-performance parallel and distributed computing*.

[25] NVM Library Team at Intel. 2016. Persistent Memory Programming. (August 2016). http://pmem.io.

[26] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C. (February 2010). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[27] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[28] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 421–432.